Automatic learning of state machines for fault detection systems in discrete event based distributed systems

OLIVER NEUNER

Master’s Degree Project
Stockholm, Sweden 2010

XR-EE-LCN 2011:002
Abstract

The electronic components in modern automobiles build up a distributed system with so called electronic control units connected via bus systems. As more safety- and security-relevant functions are implemented in such systems, the more important fault detection becomes.

A promising approach to fault detection is to build a system model from state machines and compare its predictions with properties observed in a real system. In the automobile, potential are communication characteristics between the distributed control units. Especially, the sequence of transmitted messages can be used as the basis for supervising the communication.

This thesis investigates if data gathered during system tests can be used to create state-machine system models. Such an automatically created model reflects the observed normal system behavior and can potentially be used for fault detection purposes. The task can be seen as learning a state machine from a single long message sequence.

Today’s automata learning algorithms are not designed for such single-message-sequence input data. Especially, learning without interaction between the original system and the learning algorithm is in general a NP-complete task. Additionally, if only positive data from the normal behaving system is available, the task is further complicated.

The well-known Angluin’s $L^*$ state-machine learning algorithm works in general independent from the type of input data. In order for this algorithm to be applicable, certain queries have to be answered. This work proposes a statistical approach to answer such queries.

The implemented adapted Angluin algorithm showed the potential of automatic model building in fault detection systems and, in particular, the possibility of learning state machines from a single positive data stream.
Acknowledgements

I would like to thank the Fraunhofer Institute for Communication Systems (ESK) for the possibility to conduct this work there. Special thanks go to my supervisors, Falk Langer (ESK) and Rolf Stadler (KTH) whose encouragement, guidance and support from the initial to the final level made this work possible.

I am also grateful for all my officemates and friends for the fruitful and exciting discussions, which helped me during all phases.

Finally special thanks go to my parents for their love and continuous support during all my studies.

Oliver Neuner
Contents

1 Introduction 6
   1.1 background and motivation 6
   1.2 aims 8
   1.3 a fault detection method 9
   1.4 contribution 11
   1.5 outline 12

2 Overview of fault detection theory and automata 13
   2.1 fault detection theory 13
      2.1.1 signal processing based fault detection 14
      2.1.2 predictive fault detection 15
      2.1.3 normal system model 16
      2.1.4 diagnosability 17
      2.1.5 observability 18
      2.1.6 model types 18
      2.1.7 Occam’s razor 19
   2.2 learning algorithms 19
   2.3 automata theory 22
      2.3.1 formal grammars 23
      2.3.2 finite-state machines 24

3 Selected works from model building 27
   3.1 artificial neural networks 27
   3.2 automata generating methods 28
      3.2.1 sk-string method 28
      3.2.2 genetic-algorithms 29
      3.2.3 L* method 30
      3.2.4 summary 33
4 State machine inference from a single positive data stream 34
  4.1 problem specification ........................................... 34
  4.1.1 required state machine properties ........................ 35
  4.1.2 deterministic vs. non-deterministic FSM ............... 36
  4.1.3 summary .......................................................... 37
  4.2 quality assessment of learning algorithms ................. 38
  4.2.1 reliability ....................................................... 40
  4.2.2 performance ..................................................... 41
  4.3 selecting a algorithm ............................................ 41
  4.4 adapting Aghalins L* algorithm ............................... 45
  4.4.1 data stream length variations ............................... 48
  4.4.2 learning long time dependencies ........................... 53
  4.4.3 probability considerations ................................ 54
  4.4.4 improving the learnability of long time dependencies . 55
  4.5 evaluation .......................................................... 59
  4.5.1 learnability .................................................... 59
  4.5.2 performance .................................................... 62
  4.5.3 automata sizes ............................................... 67
  4.5.4 alphabet length variation ................................... 68
  4.6 choice probability and timing .................................. 69
  5 Conclusions 71
  5.1 summary .......................................................... 71
  5.2 results ............................................................ 71
  5.3 future works ..................................................... 72
  5.3.1 special test systems .......................................... 73
  5.3.2 learning special long time dependencies ............... 73

Appendices 79
A L* example run 79
B Maximal canonical automata example 84
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Symbol streams created by communicating ECU’s</td>
<td>10</td>
</tr>
<tr>
<td>1.2</td>
<td>System overview of a possible future fault detection unit</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>Fault detection using thresholds</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>State prediction example</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Reber grammars</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>Learned Reber grammar, including initializing tree</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Learned embedded Reber grammars</td>
<td>51</td>
</tr>
<tr>
<td>4.3</td>
<td>Simulated recognizable long time dependency length vs. available data stream length</td>
<td>52</td>
</tr>
<tr>
<td>4.4</td>
<td>Performance comparison between the adapted ( L^* ) algorithm and its improved version</td>
<td>58</td>
</tr>
<tr>
<td>4.5</td>
<td>Learnability dependency on state, transitions and data stream length combinations</td>
<td>61</td>
</tr>
<tr>
<td>4.6</td>
<td>Performance evaluation, data stream length dependency for a simple FSM</td>
<td>64</td>
</tr>
<tr>
<td>4.7</td>
<td>Performance evaluation, data stream length dependency for a complex FSM</td>
<td>65</td>
</tr>
<tr>
<td>4.8</td>
<td>Execution time frequency distribution</td>
<td>66</td>
</tr>
<tr>
<td>4.9</td>
<td>Reference execution time frequency distribution</td>
<td>67</td>
</tr>
<tr>
<td>4.10</td>
<td>Execution time dependency on the number of states</td>
<td>68</td>
</tr>
<tr>
<td>4.11</td>
<td>Execution time dependency on the number of symbols</td>
<td>70</td>
</tr>
<tr>
<td>A.1</td>
<td>Example finite state machine</td>
<td>79</td>
</tr>
<tr>
<td>B.1</td>
<td>Example of a maximal canonical automata</td>
<td>84</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Comparison of non-deterministic and deterministic FSM . . . 37
4.2 Learning embedded Reber grammar with different data stream
sizes using the adapted $L^*$ algorithm . . . . . . . . . . . . . . . . 50
4.3 Learning embedded Reber grammar with different data stream
sizes using the improved $L^*$ algorithm . . . . . . . . . . . . . . . . 59
4.4 Functions fitted to the execution time dependency . . . . . . 69

A.1 Initial observation table and corresponding FSM . . . . . . . 80
A.2 Observation table after adding counterexample 1 . . . . . . . . 81
A.3 Observation table, closed and consistent . . . . . . . . . . . . 81
A.4 Observation table after adding counterexample 2 . . . . . . . . 82
A.5 Observation table, of the learned example FSM . . . . . . . 83
Chapter 1

Introduction

1.1 background and motivation

In highly complex and especially in distributed systems, fault detection is a crucial feature. In a modern automobile, diagnostic-systems for ensuring the functionality of safety relevant systems are necessary. Besides supervising the system’s functionality, fault detection and diagnostic-systems can support the car mechanic and inform the driver about occurred faults.

In general, two fault detection methods can be distinguished: Either detecting special patterns, which only occur in the error case, or detecting a discrepancy between the system and its expected behavior. To apply the first method, knowledge about possible occurring errors is needed. The second method uses knowledge about the correct behaving system.

The detection may be done by directly observing the system and using mathematical and statistic tools to decide if the system behaves correct. This approach is called signal processing based fault detection. Contrary, in model-based fault detection a model of the system is created and then tested against the system. In case of a deviation, an error is detected. Given the used model is correct, Isermann [13] presents different examples of model-based fault detection systems.

In huge digital and distributed systems, it is a hard to describe all error cases by hand and simple statistics are not sufficient for a reliable diagnosis. Therefore, a model based or a combined approach is almost inevitable. The use of a normal behavior model is advantages, since it is created from observing the system and does not explicitly relay on detailed system specifications.
1. Introduction

On the other hand, automatically creating a sufficient complex model is by far not an easy task. Depending on constrains of the model and on the available data, from which the model shall be created, different approaches are under investigation.

Lately, the number of electronic control units in automobiles is increasing. Reasons are the additional features which increase the safety and comfort [8]. Control units are communicating with each other and build up a distributed system with tasks executed on different control units. For example, sensor values are used in more than one control units.

With the number of electronic components the expenditure to ensure a safe and reliable system increase. The number of possible error sources and thus the complexity of fault detection units is increasing as well. The task of detecting faults in complex distributed systems is not solved satisfactory, yet. This becomes clear since huge fractions of faults in automobiles are caused by electronic defects. Common problems are electronic control units, deactivated due to incorrect working fault detection systems (cf. [8]).

In general, fault detection is an important function to ensure reliability in distributed systems. Especially in safety relevant systems like aircraft, power plants and automobiles, supervision and fault detection methods are crucial to indicate dangerous or not desired system states. Furthermore, actions in order to prevent damage may be triggered by such a fault detection system.

The classical fault detection approaches, like checking thresholds, do not give further insight into the system and occurring faults. As discussed above it is desirable to create a system model for fault detection. Unfortunately, from a designer’s point of view, it is a very hard or impossible task to predict system variables exactly and build an appropriate complex model by hand. Therefore, tools, which automatically create system models, are under investigation by scientists from many fields.

Nowadays, fault detection systems use signal based value checking or model based approaches. The used models or threshold values are deduced from the specifications by a Designer. Compared to the classical signal and model based fault detection methods (see 2.1.1), the proposed automatic model inference approach extracts dependency information from observations and not from specifications. Thus, even dependencies, which are not explicitly mentioned in the specifications, can be learned. The learned model only covers the domains under which the system was tested (where learning data is available). Therefore, the fault detection unit recognizes when the system leaves the tested domain, where no statement can be made if
the system works correct or not. This proposed automatic model creation approach is suggested in [15], where an artificial neural network is used as a system promising idea may be further improved by using state machines as models. State machine models can give insight into the working of the modeled system and thus beside fault detection, although fault isolation can eventually be realized. Fault isolation means finding the source or cause of the error. Because of their distributed information storage fault isolation is hardly possible using neural network models. For more details see 3.1.

From the vast variety of model types, models that predict the system’s behavior and give insight into the systems are preferred. The model ideally allows backtracking error sources, which is essential for fault correction mechanisms. Probably, the most common model type fulfilling the requirement of giving insight into a system is the state machine. This type of model is widely used by system designers to plan event based systems in the first place.

1.2 aims

Summing up the arguments, this thesis shall investigate possible automatic learning methods for state machines as system models from a given single data stream. Even if it is possible to apply those methods on different kind of data, the focus shall be on the discrete event based models. One possible application is using data streams recorded from communicating distributed controllers in a possible future automobile to learn a system model. This model can then be used to supervise the correctness of the controller’s communication.

More precise, a learning algorithm shall be suggested, which creates a state machine model from a single long data stream. The created state machine shall be able to verify if a message sequence obeys the same rules as the data stream, which has been used to create the machine. The created machine does not have to be equal to the system, which created the data stream.

Such an algorithm would enable the designer to integrate a fault detection system, which can detect errors in the communication of distributed systems. The manual creation of a state machine is rather inconvenient and complicated. An extremely accurate specification of the distributed system would be needed to create an appropriate state machine. This procedure is
1. Introduction

quit puzzling for systems with more than hundred states and uneconomic if an algorithm may solve this task within a few hours.

This thesis shall examine if it is possible to automatically create a state machine model from the type of available data. The used data are recorded message sequences from communicating electronic control units.

If one generalizes the problem to learning state machines from message sequences, a lot of applications for such a learning algorithm are possible. Voice and text recognition, intrusion detection as well as compression algorithms are examples.

1.3 a fault detection method

Modern automotive electronic systems consist up to 80 electronic control units (ECU’s), providing several thousand atomic functionalities ([4] and [15]). In such complex distributed systems, it is hardly practical to implement a fault detection mechanism for distributed errors in every single unit. This is especially clear under the assumption that functionalities are distributed between different control units. Some errors could arise in the system, even if all electronic control units (ECU’s) itself are working correctly. This motivates a central fault detection unit, which supervises the network-traffic -the communication- between the single ECU’s. On the other hand, it can’t be assured that all functions can be diagnosed/observed at a central point (see 2.1.5) and thus a combined approach, of a central fault detection unit and more basic fault detection systems in the single ECU’s, looks promising for future fault detection systems in automobiles or other complex systems.

For a distributed system to be diagnosable Langer [15] states the following assumptions:

Assumption 1 The behavior of a distributed system can be diagnosed by observing the network traffic.

Assumption 2 For detecting faults in a distributed system the important data to observe are events, which signal or cause changes in the system.

Assumption 3 An error occurs if the forecast of the network traffic matches not the observed behavior.
1. Introduction

Figure 1.2 gives an overview of a possible central fault detection unit. Figure 1.1 illustrates three communicating ECU’s. The abstracted communication packages are mapped to symbols (A, B and C). These symbols are sorted by their occurrence and called message data or symbol stream.

![Diagram showing symbol streams created by communicating ECU’s]

Figure 1.1: Symbol streams created by communicating ECU’s

As a first step, the network-traffic is observed and mapped to a symbol stream. This mapping reduces the information from the network-packages. Each packed is reduced to its relevant properties and substituted by a symbol. The so-created symbol stream is then separated into sub-streams. In this step, messages known to be independent are divided into separate streams and models.

After this step, two use cases are distinguished, the learning phase and the supervision phase. In the learning phase, a model of the system is created, which is then used to diagnose the system in the supervision phase. The learner in the diagram gets the reduced and generalized data of a substream and creates a system model, which is used by the predictor to forecast future data. Those predictions are then assessed by the comparator and in case of a discrepancy with the real data a fault signal is emitted. Optionally a diagnosis unit (examiner) backtracks and analyses the source of the fault within the system model.
1. Introduction

There are many problems to solve before a complete fault detection system based on the approach as described above can be realized. This work focuses on the learning algorithm of state machines from a data stream recorded from communicating electronic control units. The main goals are to investigate the feasibility and requirements of such a learning algorithm in the context of building models for fault detection purposes.

A concept based on the known Anghin state machine learning algorithm for automatically creating state machine models from a single positive data stream is proposed. This concept uses statistical tools to create the state machine, which most likely created the provided data stream.

As a proof of concept a possible algorithm is implemented and evaluated. This algorithm is evaluated considering the applicability for supervising the communication between the different control units in a future automobile as part of the car’s fault detection systems.

Figure 1.2: System overview of a possible future fault detection unit

1.4 contribution
1. Introduction

1.5 outline

In chapter 2, an overview about fault detection, automata and especially state machines is given. This covers predictive model based fault detection, possible model types, properties of learning algorithms as well as a detailed description of finite state machines and their relation with formal grammars.

In chapter 3, some related learning algorithms for state machines are presented. Besides this, neural networks as a possible model alternative are mentioned.

This knowledge is used in chapter 4 to identify needed properties of a suitable learning algorithm. Possibilities to evaluate those algorithms and metrics of their performance are defined. A known algorithm is adapted in order to meet all requirements and operate on the given data stream. Further improvements for a better learning of long time dependencies are presented. Finally, this improved algorithm is implemented and evaluated.

In the last chapter 5 the results are summarized and possible future improvements are proposed.
Chapter 2

Overview of fault detection theory and automata

This chapter introduces basic notation and the needed theoretical background to understand later parts. First, a general overview about fault detection is given. Then more topic specific background like automata theory and general methods of system identification are discussed. With the tools provided in this chapter, one should be able to abstract the identified properties algorithms evaluate solutions.

2.1 fault detection theory

Fault detection is an important topic in complex, safety or security relevant systems. It is not applicable to consider highly complex systems to be faultless. It can never be proven that system specifications are correct and especially in complex systems neither the hardware nor the software can be expected to be correct. Thus in mind it is essential to define a method to verify if a system behaves proper. The most intuitive approach is to compile rule-sets in the sense of thresholds for measurable parameters and if those rules are violated, the system is said to be misbehaving.

For example, if the measured temperature, speed or voltage exceeds a critical value. This approach is called analytical fault detection. Another possibility is to build a model of the system and compare the model’s prediction with the real system. This method is called model based or predictive fault detection.
Isermann and Ballé [12] summarize the historical development of model-based fault detection and introduce the basic terminology in this domain. Some definitions from [12]:

**Fault:** An unpermitted deviation of at least one characteristic property or parameter of the system from the acceptable/usual/standard condition.

**Failure:** A permanent interruption of a system’s ability to perform a required function under specified operating conditions.

**Error:** A deviation between a measured or computed value (of an output variable) and the true, specified or theoretically correct value.

**Fault detection:** Determination of the faults present in a system and the time of detection.

**Fault diagnosis:** Determination of the kind, size, location and time of detection of a fault. Follows fault detection. Includes fault isolation and identification.

**Diagnostic model:** A set of static or dynamic relations which link specific input variables - the symptoms - to specific output variables - the faults.

### 2.1.1 signal processing based fault detection

In signal processing based fault detection, measured values are processed and evaluated. Usual, thresholds are defined by the designer and if the measured values exceed those thresholds the fault detection system returns a failure signal. Figure 2.1.1 illustrates this process.

Threshold can be applied directly on measured values or on the processed signals. For example, there could be a maximum allowed slope of a measured value, and thus a threshold value is applied on values derivative.

Besides the use of thresholds, any direct mapping from the measured values to a decision if a failure occurred or not can be seen as signal processing based fault detection approach. For example, an ANN\(^1\) can be learned to decide if the measured values indicate a failure. In [6] Chow and Yee use neural networks for detecting faults in induction motors. They describe the procedure as mapping of motor parameters to motor conditions.

\(^1\)artificial neural network (see 3.1)
2. Overview of fault detection theory and automata

2.1.2 Predictive fault detection

The approach of predictive fault detection is to forecast future observations of a system, and if the real observation deviates significantly from the predictions the system is said to be faulty. The complete fault detection system consists at least of certain observers (e.g. sensors), a predictor and a comparator. The predictor uses the observations from some or all observers and forecasts future or other observations. By assessing the forecast and the observed values the comparator decides if the system behaves correctly and may suggest a point of failure in case of discrepancies.

Obviously, in the case of a deviation of the prediction and the real observations another possibility besides an erroneous system is a wrong prediction, which usually results from an incorrect system model.

Another issue is the observability of the system and the complexity of the predictor. Since in general not all states of the system can be or are observed, the output of the predictor might not be unique and correct all the time. This topic is closely related to observability (see 2.1.5). To overcome this problem probabilistic prediction might be used. That means the predictors outputs can be a set of possible future observations. Which prediction will occur does not depend on observed system states. One of those possible predictions is expected to be the real future observation.

In model based fault detection, the predictor uses a model of the system to estimate future measurements or identify inconsistent actual measurements. Neural networks, state machines or mathematical equations are possible model types.
In a discrete event based system with a limited number of events or states the fault detection system can try to predict possible future events or system states. Figure 2.2 illustrates this prediction. Every node in the picture represents a certain system state. The arcs indicate what successor states are possible. The arc labels are events, which appear during the corresponding state transition. For example, if the system is in state 1 only state 2 or 5 are allowed successor states. The state 2 is reach after the event A and state 5 with event B. If the system evolves into any other state the system behaves not correct and an error is detected. The graph in figure 2.2 can be seen as a model of the system, to predict future events or system states.

In the context of a distributed system, an event can be seen as if a certain message was communicated. The state describes the actual condition of the distributed system.

![Figure 2.2: state prediction example](image)

2.1.3 normal system model

In fault detection theory, one distinguishes between correct and normal behavior in such a way that the correct behavior is the specified desired behavior. The normal behavior describes the real system while operating faultless. Theoretically, specifications may be inaccurate, incomplete or defective and thus the "correct" model would directly incorporate those flaws as well. The difference becomes clear in the case of not complete or incorrect specifications, since a system may very well operate like the user wishes, but not
according to the specifications. For fault detection, it has several advantages to model a system’s normal behavior instead of its “correct” behavior. Practically, the normal system behavior may be directly observed from the system. Thus detailed datasets from the “normal” behaving system are available. In contrast, it is not directly possible to create datasets from abstract specifications.

Thus, a “correct” behavior model builds upon specifications, and a “normal” behavior model is created from observations.

### 2.1.4 diagnosability

Diagnosis originates from a medical term describing the process of recognizing a patient’s medical condition from his symptoms. Nowadays, diagnoses is widely used in engineering and science as the task of gaining knowledge about the conditions of a known system; often to determine if a system behaves correct and if not why or where the fault occurred. Diagnosis is although a field of Artificial Intelligence, which covers computational techniques and theoretical principles for monitoring, and testing of complex systems.

In [22] and [19] diagnosability is defined as following: "A system is diagnosable if, given a flow of observations, it will be possible in a finite delay to diagnose all the faults that have occurred on the monitored system". Moreover [19] describes a method to check if certain faults are diagnosable in distributed systems. That means a system could exhibit faults that are diagnosable and faults that are not.

Following this, diagnosability of an algorithm stands for the possibility of that algorithm to perform a good diagnosis. Classically, one could observe a system and checks if the observations exhibit some known faulty behavior. This faulty behavior has to be part of the specifications, and thus this approach is difficult to apply in real systems since usually not all fault cases are known/specified or the specified constrains are not observed or even observable. As a result, many diagnosis tools rely on verifying normal behavior of a system. The normal behavior of a system is defined as the absence of significant deviation from the average behavior, here it can be seen as an observed behavior which seems to be normal and not faulty; the observer does not see any conflict with the specifications. This kind of behavior is in practical cases often available from passed system tests.
2. Overview of fault detection theory and automata

2.1.5 observability

Observability is the property of a system, which describes the possibility to determine the state of a system by measuring its outputs. In relation to the problem of automatically building a system model, observability describes the fact that only the properties of a system can be modeled, which are observable. Whereas, diagnosability aims on the possibility of detecting faults or certain events. Observability describes the possibility to compile an exact representation of the system. If a system is observable an equivalent model can be built. If all possible faults are diagnosable, a model that can detect all faults may be inferred.

Moreover, it is not sufficient for properties to be, in principle, observable. The needed information has to be available in the data stream used to learn the model. A learned automata can only predict observable properties of a system. Therefore, this model is only reasonable in fault detection if the functionality of the system can be verified with observable properties only.

2.1.6 model types

In system theory, there are different types of models, describing the behavior of a system. One approach is to limit the description to the outer behavior, which means the inner structure or complexity of a system is neglected. The focus of this approach is the description of the systems interface to its environment. Any system, which produces the same behavior\(^2\), can be seen as equivalent. This kind of model is called black box.

In contrast, a so-called gray box model describes a system with previous knowledge about its inner structure. In most of the common fault detection approaches a designer, who designed the system and therefore has knowledge about the system’s internals, creates rules that shall not be broken, or models, which predict future system behavior. This leads to a rough fault detection based on that rules.

On the other hand, in complex systems a designer is hardly able to formulate rules with the good accuracy. In dynamic systems, those rules may alternate over time. For example, due to adaptive principles in the system or by design changes (updating the system) the need rule set may change. For the designer, it is not applicable to rewrite the system model by hand after every change. Therefore, the usage of designer knowledge in the form

\(^2\)the same input to output mapping
of a gray box model is not an easy or fast task in the case of a complex or alternating system.

Possible models are: differential equations, state machines or regular expressions.

### 2.1.7 Occam’s razor

The Occam’s razor describes an old and long discussed philosophical statement, which is very well described by Einstein:

Make everything as simple as possible, but not simpler.

In modern science theory, it states that from two theories explaining the same phenomena in the same detail the more complicated one shall be dropped. In the sense of system modeling, or in general building a predictor, it might be interpreted as the statement that the complexity of the predictor/model shall be as small/simple as possible but shall not contradict any observation.

This information is stated, since this thesis is about finding an automata modeling a system. Obviously, there are more possible automata, which can model a system correct. A concept of deciding which automata shall be chosen out of the set of possible models is needed. In other words, what are properties of a "good" model. Loosely speaking the simplest automata that can predict a sequence correct is desired.

In this context, the minimum description length (MDL) shall be mentioned. MDL is a useful tool in information and learning theory and although a basic principle of modern compression algorithm analysis. The idea of MDL is to build a minimum automata which can create a certain given string; the needed memory to store this automata is the minimum description length of the string. The MDL is closely related to the Shannon self-information theory and the Kolmogorov complexity.

A similar concept for estimating the goodness of inferred automata is the length of the automata’s description. Raman [21] proposes the use of minimum message length (MML) as a measure of goodness of an inferred automata.

### 2.2 learning algorithms

In this section different classes of learning algorithms shall be presented. These algorithms are distinguished depending on the needed input and output parameters.
passive - active learning

The class of learning algorithms can be classified as passive and active, correlating with passive and active learning. Active learning means that the algorithm can interact with the system it shall learn. In passive learning only previous known data sets are used.

For example, the task of identifying properties of a black box component could include the appliance of some test signals and listening to the output of the component, dependent of this output the input signal may be varied and adapted. By doing so hypotheses over the system can be tested easily and fast; the function space can be probed quickly. In case of passive learning all information has to be extracted from presented test data.

Raman et al. [21] summarizes the Angluin’ [1] findings about the inference of automata from sets of input data: "Finding the minimum deterministic finite automata with a set of positive and negative examples is NP-complete in the number of states". However, it is possible to identify the automaton in polynomial time if the identification procedure uses active experimentation. That means active interaction with a system reduces the complexity of identifying a black box system from non polynomial to polynomial. Learning algorithms using this interaction are called active, whereas algorithms using only given samples are called passive. Active learning algorithms possible exhibit polynomial complexity, passive algorithms do not.

positive - negative dataset

Another viewpoint on the task of system identifying is the type of available data. Imagine the task of black box analyses where one has sets of measurement data. Those sets can either describe the normal/working behavior of the box or the data could result from a not working system. The available positive (working) data sets describe what outputs a system can produce. Negative data identifies how the system shall not behave. Intuitive, it is much easier to identify a system if both types of data are available since one can test if the output of a suggested system model accepts positive samples and contradicts the negative ones.

Inferring the exact system is usually impossible, unless the available data set completely describes the system. In any other case, it is impossible to know if a system behaves correctly in a situation, which is not part of the available and classified data. For example, if one knows that all positive
datasets are given, the correct system can be created, but if all positive datasets are given but one does not know if there are more positive datasets the learning algorithm can’t know if the created machine is correct/complete or not. In the case one has only positive data available the inference task may find a minimum automata, which can only produce all positive data (one assumes a complete positive dataset) and rejects all other data sets. A practical case, since negative sets of data cannot always be recorded easily, just imagine how one should get data from a faulty behaving nuclear power plant. The tasks may be reformulated in the way that the minimal automata shall be able to produce all positive data and reject other possible data sets, depending on their similarity to the positive data. If negative datasets are available, all negative data sets shall be rejected. In case of active learning, obviously positive and negative datasets have to be allowed since the system is probed with data that may be positive or negative.

**single dataset learning**

Usually in automata theory an automata has a start state and at least one end state. In other words, an automata may accept or reject sets of different sequences. Therefore, to learn such an automata different datasets (messages streams) are needed and many published theories and algorithms focus on input data in the form of many distinguished message streams. These streams are classified as positive or negative and shall either be accepted or rejected by the automata. With the knowledge of the common start state, all of those message streams can be arranged to a tree like state machine (a maximal canonical automata, see B). The complicated task is how to simplify this machine. One of the common approaches for learning an automata from multiple datasets is the sk-method (see 3.2.1).

In contrast to this approach, the problem investigated in this thesis only uses a single, but long, dataset. This approach is chosen for practical reasons since in many applications no distinguished start and stop of a system is observable. In the fault detection case, the system may listen to a continuous stream of data where no obvious separation in sub-datasets (multiple datasets) is possible. Since the modeled system is continuously working it is assumed that there is no end- and start state in the learned machine. Furthermore, every state of the inferred automata has to be reachable from any

---

3 the system runs in a loop
other state. That means the working system has no deadlocks, it comprises only recurrent decisions.

The task will be called single dataset learning and shall result in an automata that can verify if a data stream exhibits the same features as the system, which created the data used to learn the automata.

**long time dependencies**

One big problem in many learning tasks is the ability to learn dependencies, which can be separated by long time periods. In the discrete case dependencies separated by lot of events. In the case of automata inference this has although a huge impact on the possible models. The classical hidden Markov-model includes only states that depend on the previous message, thus in order to cope with long time dependencies the Markov-model has to have a separate branch for each long-time dependent decision. This drastically increases the number of needed states.

In general learning long time dependencies is a complicated field of ongoing research, promising candidates are long-short time memory recurrent neural networks; investigated in [8]. Similar to the Markov-model a state machine, accepting long time dependencies, needs an extensive number of states but is, in principle, able to learn dependencies between distant events.

The ability to learn long time dependencies is an important feature of learning algorithms. Long time dependencies are certain transitions that are taken or not, depending on the appearance of other initiating messages. The more other messages are between the initiating messages and the dependent transitions the "longer" is the time dependency.

### 2.3 automata theory

Automata theory studies abstract machines and investigates which problems those machines can solve. Besides the well known state machine lot of different automata are known. The Turing machine, all different types of state machines, Petri-nets and many more are all special types of automata.

A task in automata theory is to investigate what type of automata is needed to model a certain system. This can be done by classifying systems by a formal language the system creates or can accept. The Chomsky hierarchy classifies formal languages by hierarchical restricting allowed production
2. Overview of fault detection theory and automata

rules (see 2.3.1). For example, this can be used to show what systems are completely describable by a state machine, and what systems can only be approximated.

2.3.1 formal grammars

A formal grammar describes the complete rule set which all sentences from a formal language have to fulfill. The grammar’s rules only describe the form of a sentence, not the meaning. Formally, a grammar is an ordered tuple \(< N, \Sigma, P, A >\). A grammar \(G\) consists of a finite set of symbols \(S\) and a finite set of production rules \(P\). The set of symbols consist of two disjoint subsets \(N\) and \(\Sigma\) \((S = N \cup \Sigma)\) with the set of non-terminal symbols \(N\) and the set of terminal symbols \(\Sigma\). One distinguished non-terminal symbol has to be the start symbol. A production rule \(t \in P\) describes the replacement of a left side by a right side. The left side has to include at least one non-terminal symbol \((A)\).

\[
t : S^*NS^* \rightarrow S^* \tag{2.1}
\]

For example, assume that \(N = \{ A \}\), \(\Sigma = \{ a, b \}\) and \(P = \{ t_1 = A \rightarrow aAa, t_2 = aA \rightarrow b \}\). Starting with the start state \(A\) and successive applies one of the possible producing rules until only terminal symbols \((a, b)\) are left.

The produced sequence of terminal symbols represents a possible accepted sentence from the formal language.

\[
t_1 : \quad A \rightarrow aAa
\]
\[
t_1 : \quad aAa \rightarrow aaAaa
\]
\[
t_2 : \quad aaAaa \rightarrow abaa
\]

One of the accepted sentences from the formal grammar above is "abaa". This string was created by applying production rules \(t_1, t_1\) and \(t_2\).

Chomsky hierarchy

Noam Chomsky classified formal grammars by restricting the grammars production rules in the so called Chomsky hierarchy. Two important classes are the context-free grammars and the regular grammars. In content-free grammars, the left side of a production rule is restricted to a single non-terminal

\(\text{the order of the language’s alphabet’s letters}\)
symbol. Regular grammars restrict the left side like content-free grammars and additionally the right side to either a single terminal symbol (including the empty symbol) or a terminal symbol followed by a non-terminal one.

Regular grammars can be represented by finite state machines. One identifies every no-terminal symbol with a state and a terminal symbol with a transition between states. With this knowledge, it is easy to see that since every state machine can be transformed into a regular grammar, state machines can only produce sentences, which can be identified by a regular grammar.

For example, the easy looking upper example produces sentences consisting of a b with one more a tailing then leading \((a^nba^{n+1} \forall n \in \mathbb{N})\).

This behavior is not creatable by a state machine. It is not reproducible by a regular grammar. This example contains a longtime dependency with a non finite depth - the number of leading a’s might be arbitrary- and thus a state machine, exhibiting the same behavior, would need an infinite number of states to produce all possible sentences.

### 2.3.2 finite-state machines

Finite state machines or short FSMs may be seen as graphs consisting of a set of states \(S\) and a set of transitions \(t\) between these states. In the modeling domain, the current state represents the history (the complete stored information) of the state machine. The current state may be changed by following a transition with origin in that state. One may distinguish FSMs between acceptors and transducers. Transducers map input symbols to output symbols. Acceptors only decide whether a sequence of input symbols is accepted or rejected.

In general, a FSM can be described as a tuple of an input alphabet \(\Sigma\), a set of states \(S\), a set of transitions \(t : S \times \Sigma \rightarrow S\), an initial state \(s_0 \in S\) and a possible empty set of final states \(f \in S\). In the case of a transducer FSM, there exist a set of output symbols \(\Gamma\) and a function \(w\) mapping either only the current state \(w : S \rightarrow \Gamma\) (Moore model) or the current state and the input symbol to an output symbol \(w : \Sigma \times S \rightarrow \Gamma\) (Mealy model).

Furthermore, a FSM is deterministic if the set of transitions is unique (there exists maximal one \(t : S \times \Sigma \rightarrow S\) for every \(S \times \Sigma\) combination; there are no two transitions from one state at the same input symbol). Otherwise a FSM is called non-deterministic.
canonical automata

Canonical automata are simple FSMs which do not require complex inference work to produce. There are two possible implementations of canonical automata: Minimal canonical automata and maximal canonical automata. A minimal canonical automata accepts every possible string using the alphabet $\Sigma$. It consists only of one state and for every input symbol $i \in \Sigma$ a transition to itself.

In contrast, the maximal canonical automata has a separate branch for every possible input string. The maximal canonical machine can only be produced easily if the available dataset contains delimited input strings. Every accepted input string is transformed into a sequence like automata with one state for each letter in the string. The treelike maximal canonical automata can then be created by merging the first state of all sequence automatas. In other words, the maximal canonical automata contains a separate branch for every possible input string. An example for a maximal canonical automata is shown in appendix B.1.

Reber grammar

A Reber grammar (RG) is a regular grammar, which is often used to test sequence predictors. Figure 2.3 (a) shows a state machine representation of the usual Reber grammar. The back nodes are said to be states, between which certain transitions are possible. Each transition is associated with a letter from a defined alphabet.

The Reber grammar can produce sentences by concatenating letters from valid transition. A possible sentence might be:

BTSSXXVPS

Since the Reber grammar only incorporates short time dependencies -two consecutive transitions are sufficient to locate the current state- one needs a more complex grammar to test the longtime dependencies. For this purpose, the embedded Reber grammar can be used. The embedded Reber grammar (ERG) incorporates two Reber grammars in parallel; see Figure 2.3 (b). This procedure can be repeated recursive in order to construct more and more longtime dependencies (more consecutive transitions are needed to know the current system state).
2. Overview of fault detection theory and automata

Figure 2.3: Reber grammars
Chapter 3

Selected works from model building

This chapter summarizes previous works in the area of automatic model building. The collected methods provide further insight to the field. All the presented algorithm can’t solve the investigated problem of single data stream FSM learning, but may serve well as a reference basis.

3.1 artificial neural networks

For prediction or modeling a system artificial neural networks (ANN) had been investigated by [15], [8] and [5]. Short time dependencies can be predicted by "simple" recurrent neural networks quite well, but they have problems forecasting long time dependencies. Long short time memory neural networks perform decent in this regime.

The big drawback of neural networks is the distributed storage of the model information, making it impossible for the designer to gain additional information about the system or backtrack the sources of discrepancies between the network’s prediction and the real system. Neural network models do not give insight into the system. They only predict future behavior depending on past messages. This implies that a neural network most likely creates wrong predictions in the beginning of the prediction phase since it has very little information available in the beginning. It is not obvious how to estimate a time after which the ANN predictions are more reliable. That this is a general problem with less information available at the beginning of
the prediction phase. On the other hand, using a state machine model not the next message has to be predicted directly. Moreover, the received data stream is verified against the model. Thus the probability of a false error detection depends only on the accuracy of the model itself and does not need many samples before a correct prediction can be made.

3.2 automata generating methods

In this section, an overview about published algorithms for automata inference shall be given. The presented methods can be classified into algorithms which:

1. start with a trivial solution and then transform that automata until some stop criteria are fulfilled.
2. formulate the task into an optimization problem and tries solving it, e.g. by using a genetic algorithm.
3. active learning using counterexamples

3.2.1 sk-string method

The sk-method belongs to the class of algorithms starting from a simple, obviously correct FSM (maximal canonical automata) and then successive refining the machine until a final criterion is fulfilled. The sk-method described in [21] learns FSM from sets of accepted and rejected strings.

In a first step, a treelike FSM is created with all sample strings starting at the same state and a single separate branch for every sample. In a next step, equal states in the treelike FSM are searched and merged. There are different heuristics for finding and merging states.

The sk-method and many other algorithms (e.g. Miclet’s tail-clustering method [17]) are based on the k-tails method by Biemann and Feldman [3].

The basic idea behind all this algorithms is to start with a correct state machine and then merge equal states.

These methods start with a maximal canonical automata and successive merge states when certain merge criteria are fulfilled. Potentially there are criteria that prevent merging e.g. presented by Maryanski in [16].
The k-tails method uses k successor states for determine which states are equal and should be merged. The sk-tail method extends this approach by statistically comparing the top s percent of the most probably, k long strings. The tail-clustering method uses clustering methods for finding mergeable states.

Since all this algorithms require a maximal canonical automata, which cannot easily be built from a single data stream, they are hardly to apply to the single data stream learning problem given in this thesis. This does not generally exclude merging based algorithms.

### 3.2.2 genetic-algorithms

A possible optimization algorithm, which can infer a state-machine, is from the class of genetic algorithms. Genetic algorithms are used to solve optimization problems by mimicking the evolutionary process. In order to use a genetic algorithm one has to map possible solutions of the optimization to a gene-sequence and a metric, measuring the quality of the solution. This metric is called fitness-function.

The algorithm starts with a set of possible gene-sequences (maybe randomly chosen) and applies mutation, crossover and selection operators. In a next step, possible solutions are produced from the set of gene-sequences. These solutions are then evaluated using the fitness-function and (with reference to Darwin’s ”Survival of the fittest”) the worst solutions are ”extinct”. The corresponding gene-sequences are removed from the set. This procedure is then repeated until a certain final criterion is fulfilled. Common final criterions are a fixed number of iterations or a certain ”fitness-level” is achieved within the set of gene-sequences.

Dupont [7] and others investigate the applicability of genetic algorithms for learning state machines. The problem is defining the fitness-function and the final criterion. Hingston [10] compared different genetic algorithms. S implementations used the MML\(^1\) in the definition of the fitness-function. Simple problems with less than ten states had been solved very successfully.

The genetic optimization is, in principle, applicable to the single data stream problem but does not guarantee to find a good solution and the performance in bigger systems is uncertain. In Hingston’s experiments the

---

\(^1\)minimum message length; see 2.1.7
algorithms often converged towards the minimal canonical automata instead of the correct FSM.

### 3.2.3 L* method

In contrast to the k-tail and similar methods, the following algorithm does not require a start automata, it stepwise builds a FSM by incorporating accepted or rejected sample strings. In comparison to optimization approaches no quality metric is needed. Moreover, the algorithm itself can be seen as an active learning algorithm.

As mentioned in the theory chapter 2, an active learning algorithm might execute in polynomial time, whereas a passive algorithm cannot. In 1987 Dana Angluin presents a learning algorithm which does not need to query the real system instead it interacts with a so called Teacher [1]. Thus in case the Teacher does not need to interact with the system the whole learning-mechanism may be regarded as passive, whereas the algorithm itself is active. The teacher encapsulates the actual learning algorithm.

For the presented algorithm (L*) the Teacher is required to answer two different queries:

**membership query**: Answers the question if a certain string can be created by the machine or not.

**conjecture query**: Answers whether a presented machine is identical with the system to learn. In case they are not identical, the teacher has to return a counterexample.\(^2\)

The L* algorithm itself works by maintaining a so-called observation table containing rows, representing candidates for states and transitions. The table’s columns are labeled by experiments for distinguishing states-rows. A state machine can be constructed from an observation table by using unique state rows as states and finding transitions by matching possible trailing sequences to consecutive states. Each element of the observation table is true if its row label concatenated by the columns label is accepted by the system. The mapping function \(T(S)\) returns true if \(S\) is accepted by the machine. \(T\) is called the membership query and \(S\) is a test string.

\(^2\)A string accepted by either the presented machine or the system but rejected by the other.
The row labels of the state candidates are donated by $S$, candidates for transitions are donated by $S \cdot A$ with $A$ as the set of possible letters and $\cdot$ as the concatenation operator. Therefore, all rows are donated by $(S \cup S \cdot A)$. The column labels are called $E$ with $E = A^*$ and might be a finite set of strings using letters from $A$.

The procedure of the algorithm is as following:

1. create initial observation table
2. check and eventually repair consistency of the observation table
3. check and eventually repair closeness of the observation table
4. repeat step 2 and 3 until the observation table is closed and consistent
5. create a state machine from the table and use the conjecture query
6. if the conjecture query returns a counterexample include the counterexample in the observation table and continue at step 2
7. if no counterexample is returned the correct state machine is created.

An observation table is closed if every transitions-rows $(S \cdot A)$ has a corresponding state. For every transition-row there has to exist a state-row $(S)$ which has the same values in every column. To fix a table which is not closed the transition with no corresponding state is moved into the state section and all possible transitions from this new state are added to the transition section.

An observation table is consistent if all states with the same values in every column have equal transition rows. That means two equal stat rows $a$ and $b$ have to have equal transitions. The condition for a consistent observation table is noted in equation 3.1.

$$T(a \cdot \tau) = T(b \cdot \tau) \forall \tau \in A$$  \hspace{1cm} (3.1)

To fix an inconsistent observation table, a new column is added. The two previously equal states $(a,b)$ are different in the new column. The new column is donated by $a \cdot e$ with $a \in A$ and $e \in E$.

Angluin argues that conjecture query might not be answered correct easily without more general knowledge of the system. If one assumes that the
algorithm shall only return an approximately correct machine with high probability a so called sampling oracle can be used to answer the conjecture query. This sampling oracle produces random strings from the alphabet \((A)\) and tests the system and the proposed machine for acceptance of that strings. In case the acceptance test returns different results, the string is assumed to be a counterexample. Algorithm 1 shows the pseudo code of Anghin’s \(L^*\) algorithm. Algorithm 2 is used within the \(L^*\) algorithm and shows how a state machine is created from an observation table.

**Algorithm 1 \(L^*\) algorithm**

create initial observation table \((table)\)

loop

while \(table\) is not closed and not consistent do

if \(table\) is not closed then \(\triangleright\) a new state is added

move all transition \(t\) into \(S\) with

\[ t \in \{ x \in (S \cdot A) \mid \text{row}(x) \neq \text{row}(s) \ \forall \ s \in S \} \]

query new elements of \(table\) using \([S \cup (S \cdot A)] \cdot E\)

end if

if \(table\) is not consisted then \(\triangleright\) a new columns is added

add column \(c\) to \(E\) with

\[ c \in (A \cdot E) \text{ s.t. } \text{row}(r_1) = \text{row}(r_2), T(r_1 \cdot c) \neq T(r_2 \cdot c), \ r_1, r_2 \in S \]

query new elements of \(table\) using \([S \cup (S \cdot A)] \cdot E\)

end if

end while

\(M = \text{create finite state machine from}(table)\)

if \(\text{CONJECTURE\_QUERY}(M)\) returns counterexample then

add counterexample and its prefixes to \(table\)

query new elements of \(table\) using \([S \cup (S \cdot A)] \cdot E\)

else \(\triangleright\) a correct machine \(M\) is found

output \(M\)

end.

end if

end loop

In appendix A an example run of the \(L^*\) algorithm, learning a very simple FSM, is given.
Algorithm 2 create finite state machine

Require: observation table (table)
create states (states) by copying the labels of all unique state rows.
for each state label $s_{from}$ in $S$ do
  for each letter $a \in A$ do
    find $s_{to}$ in states s.t. $row(s_{from} \cdot a) = row(s_{to})$
    create transition at $a$ from state $s_{from}$ to state $s_{to}$.
  end for
end for
return states and transitions

3.2.4 summary

This chapter discussed different, current approaches to automata learning. The presented list of approaches is not complete. Especially since many variants of the mentioned algorithms are known. For example, Muzammil [18] mentions three different $L^*$ implementations. One of this implementations aim is to minimize the needed queries in order to reduce the learning time.

The approaches starting with a maximal canonical machine need different sample strings and use heuristics to downsize this machine. The use of heuristics as well as the statistical optimization with genetic algorithms usually cannot create the minimum finite-state machine; they usually reach the correct FSM with a certain probability. The need of different sample strings although disqualify those approaches for the problem investigated here, since only a single "continuous" data stream is available.

In contrast the $L^*$ algorithms can create the correct minimum finite-state machine but uses active querying, which usually needs the presence of the real system or a detailed model of it. Furthermore, if the real system is involved, the learning time depends on responding times of the system. This time may be very long and therefore the whole execution time, too. Answering the conjecture query is not trivial and is often answered using a statistical oracle, which introduces a probability of creating an approximated FSM. Due to the active querying, $L^*$ algorithms are capable of learning FSMs in polynomial time. This makes them the first choice when learning large machines.
Chapter 4

State machine inference from a single positive data stream

4.1 problem specification

Now after the overall fault detection concept is introduced, this section intends to formulate the exact task of this thesis and properties of test systems. Test scenarios as well as metrics of quality, performance and reliability are defined in section 4.2. As already noted an algorithm that outputs a model, applicable in the proposed future fault detection unit from section 1.3 is searched. This learning algorithm has to cope with the available data streams in that scenario.

A passive learning mechanism has to be used since no interaction with the system is possible. As motivated in 1.1 a state machine will be used as model since a FSM gives insight into the system and potentially simplifies the diagnosis (finding possible error sources within the model and draw conclusions over the real system).

In the following, assumptions made about the available data stream are described.

Assumption 4 Each data stream can be represented by a single string, with a finite number of different letters (or symbols).

Assumption 5 All letters in the string are relevant for modeling the system.

Assumption 6 The data stream is faultless; only the correct system behavior is present.
Assumption 7 A sufficient long data stream is available; every state of the system is present in the data stream.

If Assumption 7 is not fulfilled an eventual resulting state machine is over- or under-generalized. This system model can be of practically use, but the prediction quality might be greatly reduced.

The presented fault detection system (in section 1.3) and the assumptions above imply certain constrains on the learning algorithms:

**single positive data stream learning** The complete FSM has to be learned from a single continues symbol stream. The stream does not contain faulty behavior.

**passive learning** The algorithm cannot interact with the system. The data stream is an observation of the system. This observation is not statistically optimized for identifying the system.

**unknown system parameters** It is not known how many states and transitions are needed to model the system. The system may contain dependencies between messages separated by many other messages.

In the literature, learning algorithms are known for passive learning from positive samples (e.g. the sk-tails method 3.2.1), but no algorithm for inferring automata from a single data stream could be found. Furthermore, if no information is available about the number of states or the maximum distance between dependent messages, it is impossible to know how long the data stream has to be in order to infer a correct automata.

Therefore, the algorithm shall optimally use the available information from the data stream to infer a FSM. This created state machine shall be as detailed as possible with a minimum number of states. The optimal FSM has the minimum possible uncertainty in predicting the original system.

### 4.1.1 required state machine properties

The constrains of the learning algorithm and the input data imply special properties of the created FSM. Since the system and its model are working continuous, that system model cannot contain "dead end" branches. In the produced FSM, all states shall be reachable from any other state. In other words, the model shall not contain deadlocks or states that can be reached
only once. It is assumed that the recording of the data stream is a snippet
from a random position of an infinite stream.

In the structure of a correct system model, of a continuous infinite stream,
a start state is not present. From an eventual start state, every letter from the
alphabet has to be accepted, since it is impossible to know in which state of
the created automata the data stream started. As in the definition of a state
machine exactly one initial state is forced, the learned automata are formally
no state machines. This can be mitigated by introducing a virtual initial
state with transitions for every letter branching to states from the actual
automata. In the following the actual learned automata, even without an
initial state, is referred to as a FSM or state machine.

The task of learning a “state machine” without a dedicated start state
was not treated in theory and practice, yet. The physical activation and
deactivation can be modeled as transitions from a state, in which the model
is when the real system is switched off. In Figure 4.1 on page 44 a possible
FSM is shown. The not solid nodes represent virtual states, which are not
part of the real systems, but are introduced to transform the automata back
to a finite state machine, in the original sense (with a dedicated start state).

4.1.2 deterministic vs. non-deterministic FSM

When learning state machines it is either possible to allow non deterministic
state machines as output or not. Sticking with deterministic state machines
has several advantages, especially their later uses (e.g. in the predictor)
is easier to implement. Most of the known FSM learning algorithms infer
deterministic finite automata (DFA). Deterministic finite state machines are
a subclass of this automata. Deterministic finite state machines, as argued
in section 2.3.2, do not allow several transitions with the same label outgoing
from the same state. Table 4.1 compares both state machine types in more
detail.

M. O. Rabin and D. Scott proof in [20] that all non deterministic FSM
can be converted in a deterministic one. This conversion eventually leads to
an exponential increase of states. From a descriptive point of view, deter-
nistic FSMs are a subset of non-deterministic FSMs. Non deterministic
FSMs allow more transitions but obey the other properties of deterministic
FSMs. There is no formal grammar, which can be accepted by a determin-
istic state machine and not by a non deterministic one, and the other way
around. With other words, the set of representable systems is equal for both
types of FSM.
4. State machine inference from a single positive data stream

<table>
<thead>
<tr>
<th></th>
<th>advantage</th>
<th>disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>deterministic</td>
<td>easy interpretation;</td>
<td>more states/ transitions are needed as in an equivalent</td>
</tr>
<tr>
<td></td>
<td>implementation of fault detection algorithms;</td>
<td>non-deterministic FSM</td>
</tr>
<tr>
<td></td>
<td>some known learning algorithms</td>
<td></td>
</tr>
<tr>
<td>non deterministic</td>
<td>less states and transitions as in equivalent</td>
<td>harder to interpret; few</td>
</tr>
<tr>
<td></td>
<td>deterministic FSM</td>
<td>known learning algorithms; more complicated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>interpretation</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of non-deterministic and deterministic FSM

One advantage of non-deterministic FSMs is their smaller size. Nevertheless, since deterministic FSMs are easier to interpret and more learning algorithms for DFAs are published and the fact that there is no system that can be more precisely modeled by a non deterministic FSM, in this thesis deterministic FSMs are chosen as desired system models.

4.1.3 summary

Summing up, a learning algorithm shall be proposed, which creates the minimal discrete finite-state machine accepting all data streams, creatable by an original machine. The created machine shall be "fitted" optimally to the system, which created a single observed data stream.

It is important not to confuse fitting to the original system, which created the data stream, and the fitting to the data stream itself. Using a FSM model for fault detection fitted to the data stream itself would lead to a detection indicating when the system leaves the tested regime. In contrast, a model fitted to the original system, optimally represents the system, and thus it is fitted to the regime that extends the tested domain to a regime, which most likely belongs to the original system. Since the fitting to the original system leads to be fewer wrongly detected errors, this concept is chosen.

The algorithm shall only use a single (but long) input stream from which the FSM is inferred. From every state from the FSM, a path has to exist to any other state.
4. State machine inference from a single positive data stream

This work uses the well known Angluin algorithm and extends it using statistical tools in order to cope with a single positive data stream.

When the problem of symbol mapping and stream separation is solved, the fault detection system in a future automobile may work as following. First, during test drives of a new automobile the complete bus traffic is recorded and mapped to a symbol stream. If the test engineers confirm that the car operated correct during the test rides, the symbol stream is expected to represent the normal behavior of the automobile. A FSM model is inferred from the symbol stream and then used in the fault detection unit to check if future symbol streams are accepted by the FSM. In case of a discrepancy an error is detected.

4.2 Quality assessment of learning algorithms

In order to compare different model types or different learning algorithms a common quality metric is needed. One may define two types of qualities, which answer the questions:

- How good are concrete results using this model/algorithm for predicting a system?
- How good is the algorithm in general?

The first question asks for the quality of the learned model in relation to the original system. This quality metric shall only judge how well the system is modeled. One could estimate this measurement by using a test system, apply the learning algorithm and compare the resulting state machine with the original system. The comparison can be done by directly inspecting the structure of both the system and the model. This method is only applicable if the original system is completely known and from the same type as the model. Therefore, a test state machine may be used to generate a long data string from which the learning algorithm generates a "learned" state machine. Both these state machines can be directly compared.

A more indirect approach, which can be applied, on any type of system is to define a metric of equivalence by the amount of equal results from a set of test strings. A given test string, accepted by one system but rejected by the other, may be categorized as following:
false accept: the learned machine accepts a test string that is rejected by the original one

false reject: the learned machine rejects a test string that is accepted by the original one

These definitions are similar to the common terms from hypothesis testing false positive and false negative but the terms accept and reject are more suitable in the state machine domain since a state machine either accepts or rejects certain strings (see 2.3.2).

This distinction can be very important since a model with false accept errors in the fault detection system would imply that some faulty behavior is part of the model, and thus some faults cannot be detected. On the other hand, false reject errors lead to wrongly detected faults, even if the system is working correctly. Depending on the requirements of the fault detection system one error type is worse than the other. Therefore, a separated treatment may be useful.

The model quality can be defined as in formula 4.1 with $P(f_a)$ the probability of false accepting a test string, and $P(f_r)$ as the probability of false rejecting a test string.

$$Q_m = 1 - P(f_a) - P(f_r)$$  (4.1)

For a practical test only a finite number of test strings can be evaluated but in general a system can create/accept an infinite number of test strings. In that situation, the model quality $Q_m$ can simply be estimated by formula 4.2 with $f_a, f_r$ the number of false accepted and false rejected strings and $n$ as the total number of tested strings.

$$Q_m = 1 - \frac{f_a + f_r}{n}$$  (4.2)

The model quality can obviously be split into a false accepted and a false rejected part. These parts shall be denoted $Q_{fa}$ and $Q_{fr}$. One practical problem with the above definition is the question what test strings shall be used for estimating the qualities, and especially how long these test strings shall be. Imagine the fault detection use case where a single very long data stream is verified against the FSM. Instead of testing many short strings a single long string may be used and the average length after which a discrepancy occurs could be used as quality metric. For practical reasons the
4. State machine inference from a single positive data stream

verification concept presented in the next section is used instead of estimating the model quality. Only one test string of the approximately same length as the data stream, from which the automata was learned, is used. The possible results are an over-fitted, an overgeneralized or an accurate (see section 4.2). The second question for quality, does not ask directly for the quality of the resulting model it asks for general properties of the algorithm. The most important properties are reliability, scalability and performance, which will be discussed separately in the following sections.

verification As a simple test of the learned FSM, one creates a data stream from the system (independent of the learning data streams) and verifies if that stream is accepted by the learned FSM. In the same manner, one let the learned FSM create a data stream and if the system is known checks if it accepts this stream. If the learned FSM accepts the learning data but does not accept the second data stream created by the system one may talk about an over-fitted FSM model. This usually occurs when using a too short learning data stream, which does not cover the complete system behavior. If the stream, created by the learned FSM, cannot be accepted by the original system but both the system and the learned FSM accept the learning data stream, the learned FSM is called overgeneralized. This is true because the learned FSM can create the systems original behavior and more. A FSM is called correct if it accepts the data stream, which was used to learn the FSM.

4.2.1 reliability

The reliability of a learning algorithm may be defined as the probability of the algorithm solving a given problem reproducible. For the actual case of learning FSM from a long data stream, this probability depends on the real system and the length of the data stream. For example, a given machine creates several data streams. A reliable learning algorithm should infer for each data stream the same state machine, preferable within the same time.

Another reliability aspect is the reproducibility for a constant input, which means the algorithm applied several times on the same data stream shall always produce the same result.
4.2.2 performance

Performance defines a measure of how fast the learning algorithm may solve a certain problem. This measurement may be the time needed on a common personal computer. The computer used for all experiments had 3.5GHz and 3.5 GB RAM.

Here the interesting question is how the performance scales depending on the algorithm’s inputs, especially the data stream length and the complexity of the learned automata. In computational complexity theory, the big O notation is used to describe the asymptotic behavior of a function. This can be used to describe the growth rate of the needed computation steps of an algorithm depending on certain parameters. With the needed computation steps the run time can be estimated as well. For example, $O(n, 2^m)$ describes a linear growth over the parameter $n$ and an exponential over $m$.

4.3 selecting a algorithm

Most of the algorithms explicitly rely on a set of finite samples from the system behavior. They start by building a minimum or maximum canonical machine (see section 2.3.2) and then try to transform this machine by expanding or merging states. Since, for building a maximum canonical automata several sample strings are needed, those algorithms are unsuitable for single data stream learning. The genetic algorithm approach seems to be promising, but has lots of difficulties like finding a mapping from a machine to a gene, a quality metric of a presented machine and a stop criterion.

The $L^*$ algorithm creates a minimum discrete state machine, which is unlikely to be happening in the case of a genetic algorithm. From all the examined algorithms, the $L^*$ algorithm seems to be the most suitable one for the described task, since it is not explicitly depending on any input data. Only two queries have to be answered. Due to the separation into the learning algorithm itself and a teacher it is theoretically possible to define a method creating a FSM in polynomial time, since the learning algorithm itself can be active. The teacher has to encapsulate the algorithm by answering active queries from the algorithm but is passive itself.

In order to apply the $L^*$ algorithm one needs to be able to answer two questions the membership query and the conjecture query approximately. How to answering these queries is the main subject of the following sections.
Correctly answering those queries is not trivial from the available data stream and thus a statistic approach is presented.

In the following, it shall be drafted how those two queries might be answered by a Teacher, using only the knowledge given from a single but long data stream. The principle questions behind the queries are presented in section 3.2.3.

**membership query:** This query may be answered trivially by checking if the queried string is a substring of the given data stream. This approximation requires, being correct, that every possible string accepted by the machine has to be in the data stream. In order to use this approximation one has to assume a maximum length of the query string, much shorter than the data stream. The maximum distance of a long time dependency to be learned has to be smaller than the maximum query length. On the other hand, a very long data stream can be assumed, and therefore, this is not a real limitation of this membership query approximation.

**conjecture query:** The conjecture query shall check if a proposed machine is the correct one and if not it shall return a sample string which is treated different in the correct and the proposed FSM. This query is trickier to answer but fortunately [1] already proposed to use a random sample oracle to answer this query (see 3.2.3). The basic idea is to create test strings and check if both, the correct and the proposed machines treat these strings equal. Since the correct FSM is not available, it may be assumed that a test string, accepted by the correct FSM, is a substring of the given data stream. Otherwise, the test string is rejected.

One remaining question is how the algorithm can learn a machine represented by a connected graph\(^1\) without a dedicated start or stop state. Since in the strict definition a state machine has one start node, at least one final node and eventually not final nodes the original \(L^*\) algorithm uses this features. Using the \(L^*\) algorithm directly, it will not learn the desired automata immediately, instead it will create a state machine with a start state from which every substring of the stream is accepted.

This implies that the desired automata is represented by a subgraph of the created graph (FSM). Since all possible paths in the desired graph shall be

---

\(^1\)each node can be reached by a path from any other node
accepted, all nodes (states) in the created graph are final nodes. Furthermore, since the start state has to accept every letter from the alphabet (which is in the data stream), the start state has to have an outward directed edge (transition) for every letter. Thus the start state cannot always be part of the desired graph.

In general, the created graph will consist of three subgraphs: the desired automata, a subgraph with no final state (collecting all not accepted strings) and a tree like subgraph containing the start state with branches leading to states of the other two subgraphs. The last tree like subgraph is optional in the case the start state is not part of the desired automata. This tree like graph leads to states from the desired graph via paths representing strings, which uniquely identify the state from the desired automata.

Figure 4.1 shows the result after learning a Reber-grammar (RG) with an adapted $L^*$ method; the solid nodes are states from the desired state machine; the not solid nodes represent the initializing subgraph which exhibits a treelike structure. A path in this subgraph from its start node to a node from the Reber-grammar represents an input string, which uniquely identifies this state. For example, the right most branch from the topmost node to the rightmost node from the RG has two edges labeled $P, X$ and $S$. Thus after the input strings "PS" or "XS" the RG state machine would be in the rightmost state. The input string "VV" (in the second right branch) would although lead to this state.

Algorithm 3 shows the pseudo code of the adapted $L^*$ algorithm. Besides the removal of the tree like structures in the end of this algorithm is exactly the $L^*$ algorithm. The main adaptions to the Angluin algorithm are made in the membership and conjecture query. Algorithm 2 on page 33 shows the implementations of the create finite state machine function. The membership query is abbreviated by $T(.)$. The row(s) command returns one row with label $s$ from to the observation table.

For the implementation of $L^*$ a few dynamic programming improvements are published, for example [11] shows how one could reduce the number of needed membership and conjecture queries. Since the focus of this thesis is to show the applicability of learning algorithms to the problem of single dataset learning, not to improve the $L^*$ performance, a strait forward implementation has been used. A good overview of different variations and improvements of the $L^*$ algorithm is given in the PhD. thesis from Muzammil [18] and Kern [14].
4. State machine inference from a single positive data stream

Figure 4.1: Learned Reber-grammar (solid nodes) including the initializing tree structure (un-solid nodes)
4. State machine inference from a single positive data stream

4.4 adapting Angluins $L^*$ algorithm

The most complex query, which has to be answered, when adapting the $L^*$ method, is the conjecture query, with the task of finding counterexamples\(^2\) for a proposed FSM, if that machine is not the correct one. First the conjecture query has to decide if the proposed FSM is correct, and if not return a string which represents the difference between the suggested machine and the correct one.

The idea is to let the proposed machine create test strings and verify if that strings are part of the given data stream. If a test-string is not part of the data stream the test-string should not been possible to create by the FSM and thus represents a counterexample.

Since every correct machine can produce an infinite number of correct strings and the length of the available data stream is practically finite, two problems occur:

- The correct machine is proposed but rejected because the data stream does not comprise a certain test string. This sort of mistake may be called false reject and look like a problem of a too short data stream, but it is a general feature of the task. It is never possible to be sure, that the real machine does not have more states than the proposed one. Those states may have never been reached in the available data stream. In fact, it is obvious that not all create able strings can be part of the data stream, and thus one can always find a counterexample (producible by the correct machine, but not part of the data stream). Therefore, the number of test strings has to be limited in such a way that the probability of a false reject becomes small.

- A wrong machine is proposed but accepted because no test string explored the difference between the proposed and the desired machine. This type of mistake may be called false accept and occur if the amount of test strings is too small or the length of the test examples is not sufficient to cover the behavior of the desired machine. This leads to the fact that it is impossible to infer the exact correct machine using a finite amount of tests since not all possible output strings can be tested.

\(^2\)strings, accepted by the proposed machine but not by the machine represented in the data stream
As a result, the length/number of test strings shall be as huge as possible to omit false accepts. On the other hand, to omit false rejects the test string length/number has to be very small compared to the finite data stream. Additionally, to save computation costs, the maximum amount of test strings is limited. This leads to the complex question how many and which test strings shall be used.

One approach is to test all producible test strings, shorter than a certain length \( \text{maxtestlen} \). This maximum length of test examples shall be chosen in a way that the probability of not finding a certain test string in the data stream is very low \( (\text{e.g. less than } \delta; \text{ maybe } 0.1\%) \).

This task is similar to the stochastic problem of rolling a \( n \) side dice \( m \) times, asking for the probability of rolling each side at least once. The number of dice sides \( n \) equates the number of possible test strings with the length \( \text{maxtestlen} \). Using \( m \) as the number of test-strings (with this length) in the data stream. The probability of finding all test-strings in the data stream (rolling each dice side at least once) is then given by 4.3 [9].

\[
P(n, m) = \sum_{i=0}^{n} (-1)^i \binom{n}{i} \left(1 - \frac{i}{n}\right)^m
\]

\[
P(n, m) > 1 - \delta
\]

In order to apply the criteria 4.4, \( n \) and \( m \) has to be estimated. The number of possible test strings \( n \) with length \( \text{maxtestlen} \) can be estimated by the cardinality of the alphabet\(^3\) to the power of \( \text{maxtestlen} \). However, this is a quite rough estimation since any complex automata accepts only a much smaller amount of strings. A much better estimate is the average number of possible decisions per state to the power of \( \text{maxtestlen} \) times the number of states. The average number of possible decisions can be calculated by the number of transitions divided by the number of states. In fact, the correct number \( n \) of possible test strings can be extracted from the machine but using the average number of decisions per state is more practical since the exact calculation is computationally much more complex and the exact value would neither improve the quality nor the performance of the criteria 4.4. To be on the "save" side the average number of possible decisions might be

---

\(^3\)the number of letters in the alphabet
rounded to the next higher integer. The number of test strings \( m \) within 
the data stream with length \( \text{streamlen} \) is estimated to be the number of 
possible sub-strings of length \( \text{maxtestlen} \).

\[
m = \text{streamlen} - \text{maxtestlen} + 1 \quad (4.5)
\]

In any practical application \( \text{streamlen} \gg \text{maxtestlen} > 1 \) and thus 
\( m = \text{streamlen} \) would be a suitable estimate as well.

\[
n \approx \text{number of states} \left( \frac{\text{number of transitions}}{\text{number of states}} \right)^{\text{maxtestlen}} \quad (4.6)
\]

Unfortunately, equation 4.3 is not easy to implement on a normal computer 
since the binomial \( \binom{n}{i} \) becomes a very large number and the \( (1 - \frac{1}{n})^m \) 
term becomes very small. The exponent \( m \) easily reaches very high values (the 
data stream may consist of more than hundred thousand symbols).

Therefore, a simple approximation for \( P(n,m) \) has to be used. This approximation 
should return a probability lower than 4.3 in order to make sure that \( \delta \) is the maximum probability of a false reject. As an approximation the 
complementary event of not finding a certain test string of length \( \text{maxtestlen} \), 
times the number of test strings of that length may be used. This approximation 
predicts always a lower probability as the exact calculation since the 
calculation of not finding a certain test string, multiplying with the number 
of tested strings and then getting the complementary event disregards the 
possibility of not finding more than one test string. Therefore, the approximation 
works quite well if the probability of finding all test-strings within 
the stream is high. It although can be seen as the first two summands from 
the correct formula 4.3 \((i = 0 \text{ and } i = 1)\).

\[
P(n,m) \approx 1 - n \left( \frac{n-1}{n} \right)^m \leq P(n,m) \quad (4.7)
\]

It is important to note that the approximation (4.7) as well as the exact 
formula (4.3) assumes that all possible test strings have the same probability. 
This is not necessarily true but in case one relaxes this assumption to an 
arbitrary probability for each test string the resulting probability of an false 
reject becomes arbitrary as well.
4. State machine inference from a single positive data stream

The approximation 4.7 and the condition 4.4 are used in the adapted $L^*$ algorithm (Algorithm 3) as a stop criterion. When the $L^*$ algorithm suggests a FSM, all producible strings, up to a maximal length, from that machine are tested against the data stream, in the conjecture query.

By limiting the test string length, the number of transitions between long time dependencies is limited as well. In order to represent long time dependencies, the test strings have to include the dependent transitions. That means the minimum test string length is the minimum length a string must have to distinguish all states. This length may be called the minimum long time dependence length.

4.4.1 data stream length variations

In order to test the algorithm’s performance dependency on the data stream length, different data streams were tested using the same automata. As automata the embedded Reber grammar (figure 4.2(b)) was used. The algorithm, as suggested above, tests all creatable strings up to a certain length. Since the number of creatable strings grows exponentially with this maximum test-example length ($\text{maxtestlen}$), the performance of the algorithm should be exponential in $\text{maxtestlen}$.

On the other hand, the data stream length has to increase exponentially for being able to test strings up to $\text{maxtestlen}$. Therefore, the algorithm’s performance should be linear in the data stream length. Figure 4.3 shows the needed data stream length vs. the maximum length ($\text{maxtestlen}$). The prediction is made for a false reject probability of ($\delta = 1\%$).

In Figure 4.3 was simulated using the same approximations as in the implemented conjecture query from the $L*$ algorithm (Equation 4.7). The prediction was especially made for the embedded Reber grammar (figure 4.2 (b)) but since only the number of transitions and states are important, any similar machine would result in the same prediction. Using equation 4.7 the stream length is successive increased for a fixed $\text{maxtestlen}$ until the probability becomes less than ($\delta = 1\%$) and the corresponding stream length is noted. This is repeated for different values of $\text{maxtestlen}$ between two and thirty to get the prediction in figure 4.3.
Algorithm 3 adapted $L^*$ algorithm

**Require:** input stream (stream)
- collect all symbols from stream into alphabet
- create initial observation table (table)
- set maximal test string length $\text{maxtestlen} = 2$
- set $\delta = 0.001$ for 0.1% probability of a false reject

**loop**
- ensure table is closed
- ensure table is consistent
- $M = \text{CREATE FINITE STATE MACHINE FROM}(\text{table})$ \Comment{like in $L^*$}
- if conjecture_query($M, \text{maxtestlen}$) returns counterexample then
  - add counterexample and its prefixes to table
  - query new elements of table
- else
  - if $M$ accepts stream then
    - save $M$
  - end if
  - estimate false_reject probability $P$ for $M$
  - with test-strings length $\text{maxtestlen} + 1$
  - if $P < \delta$ then
    - $\text{maxtestlen} = \text{maxtestlen} + 1$
  - else
    - exit loop
  - end if
- end if
end loop

reset $M$ to the last saved machine
remove all states from $M$ with no outgoing arc
remove all states from $M$ which are not reachable from every other state
output $M$
end.
Algorithm 4 conjecture_query

Require: proposed machine \((M)\)

Require: maxtestlen

Require: data stream

repeat
    create a string \((s)\) with length \(maxtestlen\) accepted by \(M\)
    if \(s\) is NOT substring of stream then
        return \(s\) \(\triangleright\) return \(s\) as counterexample
    end
end if

until all strings \(s\) with length \(maxtestlen\) creatable by \(M\) been substrings of stream

return true;

Algorithm 5 membership_query \(T(s)\)

Require: test string \((s)\)

Require: data stream

if \(s\) is substring of stream then
    return true
else
    return false
end if

<table>
<thead>
<tr>
<th>stream length</th>
<th>number of counterexamples</th>
<th>max. test-string length</th>
<th>learning time [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 000</td>
<td>12</td>
<td>5</td>
<td>0.7</td>
</tr>
<tr>
<td>5 000</td>
<td>12</td>
<td>5</td>
<td>0.6</td>
</tr>
<tr>
<td>50 000</td>
<td>13</td>
<td>7</td>
<td>5.9</td>
</tr>
<tr>
<td>200 000</td>
<td>13</td>
<td>9</td>
<td>22.7</td>
</tr>
<tr>
<td>1 000 000</td>
<td>13</td>
<td>11</td>
<td>114.0</td>
</tr>
<tr>
<td>2 000 000</td>
<td>13</td>
<td>12</td>
<td>239.0</td>
</tr>
<tr>
<td>9 800 000</td>
<td>15</td>
<td>14</td>
<td>1456.8</td>
</tr>
<tr>
<td>21 700 000</td>
<td>15</td>
<td>16</td>
<td>5297.6</td>
</tr>
</tbody>
</table>

Table 4.2: Learning embedded Reber grammar with different data stream sizes using the adapted \(L^*\) algorithm
4. State machine inference from a single positive data stream

(a) embedded Reber grammar (not recognized long-time dependency)

(b) correct inferred embedded Reber grammar

Figure 4.2: learning results for the embedded Reber grammar (b) with correct recognized long-time dependency
Figure 4.3: maximum distance of recognizable dependent messages ($maxtestlen$), depending on the available data stream length (simulation), used the number of states (17) and transitions (28) from the embedded Reber grammar (see figure 4.2 (b))
Table 4.2 shows the experimental results. With a data stream length of 1000 no correct machine besides the minimal canonical one could have been inferred. For a length of 5000 the machine shown in figure 4.2 (a) was created. One can see that the longtime dependency in the embedded Reber grammar could not be recognized correctly but the inferred machine is correct in the sense of accepting all possible strings created from the embedded Reber grammar. For all longer data streams, the correct original embedded Reber grammar (figure 4.2 (b)) was created. In addition, as predicted the learning time increased linear with the data stream length.

Overall one can say the performance regarding the stream length is quite good but the fact of a very steep exponential dependency of maxtestlen reduces the capability of learning long time dependencies. This problem will be addressed in section 4.4.2. The number of needed counterexamples may vary depending on the order of the found counterexamples. This order varies since -in the actual implementation- test strings are created dependent on the order the symbols occur in the data stream.

4.4.2 learning long time dependencies

From the theoretical analysis and the first tests (table 4.2) the suggested algorithm complexity is exponential to the maximum length of tested counterexamples. This length has to be longer than the minimum distance between depended transitions in order to recognize this dependency. Thus, in order to improve long time dependency learning, increasing the test string length is necessary.

The suggested algorithm tests all possible strings up to a length for which the probability of a false reject (returning a not correct counterexample) is small. If one would relax the constrain, from testing all possible strings up to a certain length, to statistically testing a certain amount of strings, the maximum length of those test strings may be greatly increased. Optimally, similar to the random sampling oracle from Anghin’s paper [1], one would test the samples which most likely occur in a data stream.

Again in the example of the embedded Reber grammar 4.2 a long sample, covering the transition branching into the two equal Reber grammars and the reunification, would give the algorithm the possibility to recognize the long-time dependency. This recognition is independent of the actual path the sample takes inside the inner Reber grammar.
4. State machine inference from a single positive data stream

4.4.3 probability considerations

The result, in figure 4.3, shows that the needed data stream length grows exponential with the minimal length, needed to recognize a long time dependency, is not very satisfying. Furthermore, the simple adapted algorithm limits the maximum length of all test strings since it assumes that all test-strings have the same likelihood of being accepted by the correct FSM. In the improved algorithm, the probability of a test-string being accepted by the correct FSM is calculated individually for every test-string candidate and such it is possible to test longer strings.

There are two types of counterexamples: Either the counterexample is part of the data stream but cannot be created by the proposed machine \(\text{positive counterexample}\) or the counterexample can be created by the machine but is not part of the data stream \(\text{negative counterexample}\). In the case, the algorithm finds a \text{positive counterexample} the question is, if the algorithm should add the counterexample to the machine or not. On a first look, this question seems obvious but since the algorithm while updating the \text{observation table}, has to query the counterexample concatenated by every letter from the alphabet and by every column label. The question is, if it is possible to answer all those additional queries correctly by checking if those new strings are part of the data stream. If all new strings are accepted by the modeled system, these strings have to be part of the data stream in order for the algorithm to infer a correct machine. If one assumes that the probability for all transitions is equal, one could conclude the counterexample has to occur at least \(n\) times in the data stream in order to be sure that the probability for correct additional queries \(x\) is greater than \(p\). With \(m\) as the number of possible successor strings, one may approximate the probability of correctly adding a string to the observation table by equation 4.8.

\[
p \approx 1 - \left(\frac{m-1}{m}\right)^n x
\]  

(4.8)

The number of possible queries \(m\) may be approximated as \(m \approx \text{tpsl}\) with \(l\) as the maximum length of the column’s labels and \(\text{tps}\) as the average outgoing transitions per state. If this probability, of answering all necessary adjacent queries correct, is higher than a threshold \(\delta\), the counter example is legal and can be added to the observation table.

The same principle can be applied for \text{negative counterexamples} but here the role of the correct and the proposed machine are switched. In practice,
the conjecture query creates test-strings from the proposed machine. If the probability of those test-strings for correctly answering all necessary adjacent query’s $p$ is higher than $\delta$, possible successors of those test strings can be tested vs. the data stream. In addition, the counterexample is added to the observation table. A successor of a test-string is a string, producible by the proposed FSM, starting with the test string.

### 4.4.4 improving the learnability of long time dependencies

The idea of the following improved algorithm is not testing all possible strings up to a certain length, rather testing only the most probably occurring strings. Thus, the maximum length of such test strings can eventually be increased. Instead of calculating the occurrence probability of every test string candidate and then deciding if that string shall be tested, the probability of falsely accepting this string is used. Here, the probability of a falsely accept can be seen as the probability of a test string candidate to result in wrong conclusions about the machine. This can happen because while processing a counterexample queries about possible successors of that counterexample have to be done. The available data from the data stream is eventually not sufficient for statistically answering this queries correct.

The new conjecture query allows creating test strings chosen depending on the available data. Only test strings, for which the approximated probability of correctly adding this string to the observation table (Equation 4.8) is higher than a certain probability $\delta_a$, are tested. Thus, the amount of unnecessary test strings shrinks and the length of the individual test strings may be much longer. This is depending on the available data and the proposed machine. The possible increased length of the test strings improves the learn ability of long time dependencies. The algorithm 6 shows the pseudo code of this improved algorithm.

Like in the classical L* algorithm after each “add to table” the table is ensured to be closed and consistent. This step is omitted in Algorithm 6 to highlight the important steps. Besides the main algorithm the conjecture query was changed.

To show the improvement, the exact same experiment from section 4.4.1, with results in table 4.2, is repeated with the improved algorithm. The new version returns a correct FSM for every tested input string length.
Algorithm 6 improved adapted \( L^* \) algorithm

**Require:** input stream (stream)
- set \( \delta_a = 0.001 \) for 0.1% probability of a false accept
- collect all symbols from stream into alphabet
- create initial observation table (table)
- \( M = \) CREATE FINITE STATE MACHINE FROM(table)

**loop**
- if CONJECTURE_QUERY(\( M \)) returns counterexample then
  - add counterexample to table
  - \( M = \) CREATE FINITE STATE MACHINE FROM(table)
  - found = true
- else
  - found = false
- end if
- correct = true
- while a substring \( s \) of stream is not accepted by \( M \) do
  - if \( \text{PROBABILITY OF FALSE ACCEPTING}(s) > \delta_a \) then
    - correct = false
    - leave while
  - found = true
  - add \( s \) to table
  - \( M = \) CREATE FINITE STATE MACHINE FROM(table)
- end while
- if correct then
  - save \( M \)
  - if found == false then leave loop
  - end if
- else if found == false then leave loop
- end if

**end loop**
- reset \( M \) to the last saved machine
- remove all states from \( M \) with no outgoing arc
- remove all states from \( M \) which are not reachable from every other state
- output \( M \)

end.
Algorithm 7 improved conjecture_query

Require: proposed machine \( (M) \)
Require: data stream
repeat
    create a string \( (s) \) with is accepted by \( M \)
    if Probability of false accepting \( (s) < \delta_a \) then
        if \( s \) is NOT substring of stream then
            return \( s \)
        end.
    end if
until all strings \( s \) creatable by \( M \) are substrings of stream or there probability of false accept is \( > \delta \)
return true;

In the case with 1000 symbols the longtime dependency is not recognized but the result is correct and equivalent to the 5000 symbols’ case from the section 4.4.1. With 5000 symbols the correct FSM, including the long time dependency, is created. Here the trade-off between run time and the ability to recognize long time dependencies becomes clear. The improved algorithm performs faster for short data streams and can recognize long time dependencies with less data but needs longer than the ”simple” adapted algorithm when processing long data streams. This learning time might be improved by limiting the number of test strings but is not implemented because, in this work, the maximum long time dependency length recognition shall be achieved. The maximum length of a test-string, the number of found counterexamples the used stream length and the total execution/learning time is shown in table 4.3. This is, in principle, the upper bound for the distance between two dependent transitions, which may be recognized. The longtime dependency in the embedded Reber grammar has a shorter distance, but in order to be recognized a corresponding string has to be statistically testable (see section 4.4.3).

Summing up the improvements, the new algorithm creates test strings depending on the available data and the actual guess for the correct machine. Therefore, all possible test strings, which can be tested correctly with a certain probability, are assessed. Furthermore, by checking the FSM vs. the whole data stream and eventually adding positive counterexamples the returned FSM is ensured to accept the complete data stream.
4. State machine inference from a single positive data stream

(a) the improved algorithm needs a much shorter messages stream for recognizing longtime dependencies

(b) the time needed for recognizing the same longtime dependency is much shorter when using the improved algorithm

Figure 4.4: Performance comparison between the adapted $L^*$ algorithm and its improved version (data from table 4.2 and table 4.2)
4. State machine inference from a single positive data stream

<table>
<thead>
<tr>
<th>stream length</th>
<th>number of counterexamples</th>
<th>max. test-string length</th>
<th>learning time [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 000</td>
<td>7</td>
<td>13</td>
<td>0.047</td>
</tr>
<tr>
<td>5 000</td>
<td>10</td>
<td>15</td>
<td>0.296</td>
</tr>
<tr>
<td>50 000</td>
<td>11</td>
<td>14</td>
<td>3.125</td>
</tr>
<tr>
<td>200 000</td>
<td>11</td>
<td>21</td>
<td>23.281</td>
</tr>
<tr>
<td>1 000 000</td>
<td>10</td>
<td>24</td>
<td>542.128</td>
</tr>
<tr>
<td>2 000 000</td>
<td>10</td>
<td>27</td>
<td>1789.19</td>
</tr>
<tr>
<td>9 800 000</td>
<td>10</td>
<td>32</td>
<td>37699.2</td>
</tr>
</tbody>
</table>

Table 4.3: Learning embedded Reber grammar with different data stream sizes using the improved $L^*$ algorithm

4.5 evaluation

For testing purposes random deterministic state machines are created. In order to estimate the algorithms performance depending on the length of the data stream, the numbers of states, transitions and unique messages, different deterministic state machines with those parameters are created. These machines are then used to create data streams, which are used as input for the learning algorithm.

In a first test-run, the randomly created FSM’s were not restricted to be deterministic. However, since the learning algorithm only outputs deterministic FSM’s and an eventual exponential increase of states might occur while converting non deterministic to deterministic FSMs the use of random FSM is not ideal. To investigate the algorithm’s performance dependencies deterministic state machines are used as original systems.

4.5.1 learnability

Before the performance in the form of execution time is evaluated, some tests showed that not from all test data streams a correct FSM could be inferred. Therefore, as a first step, it is investigated under what circumstances no correct FSM can be produced.

It has been proven by Angluin [1] that his algorithm always produces a correct output. In addition, the proposed modified version cancels execution
only if certain needed queries cannot be performed with satisfying probability. Therefore, all not learnable cases result from an insufficient amount of available data. In other words, the used data stream is too short to infer the correct machine.

One could argue that at least the minimal canonical\textsuperscript{4} FSM can be inferred in every case, but this is excluded by purpose, because this result would not be of practical use. Especially, the adapted algorithm may infer the canonical automata in case the provided data statistically suggests this. In general, the adapted algorithm tries to create a correct FSM, which completely accepts the provided data stream and then tries to specialize by reducing possible over-generalizations. In all failed learning attempts, the provided data statistically suggested that the correct machine is not the canonical one, but did not provide sufficient information to find another solution.

In order to identify cases where no correct machines are inferred random state machines with varying number of states transitions and number of symbols are created. These FSMs are learned using different long data streams.

In the figures 4.5 the number of states is drawn vs. the number of transitions per state. State machines with states from 10 to 100 were investigated, for every state the number of transitions is either equal, twice or three times the number of states. The diagrams 4.5 (a)-(f) show \times{} at positions where no correct FSM was inferred. In the experiments FSMs which used up to 50 distinguishable symbols were used. The limitation to less than 50 eventually decreases the probability of correctly inferring a FSM with a high number of transitions but was introduced to limit the number of tests and thus the time needed for executing all test-scenarios. If a correct state-machine could be inferred a circle is drawn. The different subgraphs show the development when the data stream length is decreased. At a length of 10 million symbols all tested parameter combinations resulted in a correct FSM. The simplest FSM, which consists of one transition per state and therefore, requires fewer symbols in the data stream that, even for only thousand symbols, in all tested cases a correct FSM could be inferred.

In the created test cases, the original system to learn is known and the correctness of a solution can be measured by comparing the learned FSM with the original one. See the quality section in the problem verification paragraph in 4.2.

\textsuperscript{4}canonical FSM is defined in section 2.3.2
4. State machine inference from a single positive data stream

Figure 4.5: Learn ability of discrete FSM with different state, transitions and data stream length combinations
The mentioned result verification has been implemented by creating a
different data stream from the original system and check if the learned one
accepts this data stream as well. The same has been done the other way
round (letting the learned FSM create the test string). Therefore, in case
the learned machine can accept the data stream used to learn this FSM but
does not accept the verifying stream one could say the inferred FSM is not
correct.

In the following, a correct FSM is used for a FSM, besides the minimal
canonical one, accepting the data stream, which was used for inferring. If
this correct FSM does not accept other data streams created by the original
machine the learned FSM is called over-fitted.

The question to be answered is how long a data stream should be to infer
a machine with a certain complexity. As a measure of complexity the number
of states, transition and distinguishable messages are investigated.

4.5.2 performance

The proposed algorithm is not optimized for a minimal execution time, firstly,
because published improvements for reducing execution time are not imple-
mented. For example, reducing the number of needed queries as suggested
by Hungar [11]. Secondly, the emphasis of this work is on the principal pos-
sibility of applying FSM learning algorithms in fault detection applications.
For instance, the conjecture query tries to test as many strings as statistically
possible, instead of using Angluin’s [1] proposed number of test strings in or-
der to verify the correctness of a proposed machine. The maximum number
of strings, as suggested by Angluin, depends on a confidence and accuracy
parameter, defining the probability of having at least a certain approximation
of the correct FSM.

Even when performance does not play the main role, the execution time
for learning is interesting. The presented times may be seen as an upper
bound which should not be exceeded in any future implementation. Espe-
cially since Angluin’s algorithm should be able to execute in polynomial or-
der [1]. However, since the conjecture query tests as many strings as possible
depending on the machine the number of possible strings growths exponential
with the available data stream.
According to [1] the maximum length of any string in the observation table is $|\Sigma| + 2|S| - 1$. In addition, the size of the observation table is in the order given in equation 4.9.

$$\mathcal{O}(|S|^2|\Sigma|^2 + |S|^3|\Sigma|)$$

(4.9)

Angluin further argues that if the membership and conjecture query can be executed in polynomial time the total algorithm runs in polynomial time. With the same arguments as in 4.4.1 the length of the available data stream should influence the execution time in a linear way.

Figure 4.6 shows the execution time vs. the data stream length. The learned FSM is a very simple one with ten states with every state connected to only one other state (forming a circle); these ten transitions are labeled with one of five symbols. This very simple structure has been chosen in order to minimize machine complexity dependencies. In this case, the approximate execution time is $17\mu$sec per symbol in the data stream.

The number of tested strings in the improved algorithm depends on the machine and not only on the available data. For a simple machine as used above, which can only create one unique message stream (from a chosen start-state), the number of tested strings does not vary much with the available data length, since in all tested scenarios the data stream is highly redundant and an increase does not provide more information. The linear increase of execution time, however, originates from the longer time needed for a single membership query (finding an element in an unsorted list has linear order $\mathcal{O}(l)$ with $l$ as the length of the list; the available data stream).

The same experiment is repeated with more complicated FSM’s. Figure 4.7 shows the time dependency for a random discrete FSM with ten states twenty transitions and up to ten different messages.

Finally, to investigate how strong the performance depends on the structure of the automata, independent of the previously described parameters like number of states, transitions or distinguishable symbols. In order to do so hundred random state machines, with the same number of states transitions and distinguishable symbols as well as the same data stream length, are created and learned. The only difference between those FSM’s is the inner structure. Figure 4.8 shows the frequency distribution of the execution time for learning these state machines.
Figure 4.6: Performance evaluation depending on the available data stream length using a simple ten state circular FSM
4. State machine inference from a single positive data stream

(a) Performance analyses of a complex FSM from figure (b)

(b) Complex discrete FSM used for investigating stream-length performance dependency

Figure 4.7: Performance dependency on data stream length using a complex ten state, twenty transitions FSM
In order to assess this data a machine with the same parameters has been learned hundred times. The frequency distribution of this machine’s learning time is shown in Figure 4.9. All the tested machines had ten states, twenty transitions, ten different symbols and the data stream from which the machine was learned had one million symbols.

![Figure 4.8: Execution time frequency distribution](image)

Comparing the two execution time frequency distribution figures (4.9 and 4.8) the similar structure suggests only a minor dependency of the execution time on the inner structure of the deterministic FSM. However, the examples with significant longer execution times in 4.8 (about 400 sec) eventually result from a structural more complex machine and the distributions are similar because the majority of the random generated test-FSM have a similar "complexity" (for being learned with the suggested learning algorithm). The random FSM generator starts with creating an initial circular machine and then successive ads additional transitions. This procedure eventually leads to a narrow complexity distribution of the created FSM. On the other hand, the FSM which needed the longest learning time in figure 4.8 was learned several times with different learning times, mostly in the 50 second region. Therefore, the varying execution time for machines with the same number
of states, transitions and distinguishable symbols most likely originates from
the order in which the algorithm finds and adds counterexamples (the cre-
ation of counterexamples within the conjecture query uses some randomness
to produce test strings).

4.5.3 automata sizes

Here the algorithm’s performance dependency on the machine size shall be
investigated. Besides the already mentioned dependency on the inner struc-
ture and the available stream length, the dependency on the total size or
more precise the number of states shall be investigated.

In order to eliminate as many influences as possible for investigation of
the dependency on the number of states a pure circular FSMs with unique
symbols on every transition is used. In the following, it will be evaluated
how the execution time varies on the number of states of such circular state
machines. Figure 4.10 shows the dependency of execution time vs. the
number of states.

In order to investigate the dependency further, several regression tools
have been applied to the data. By fitting several functions to the data and calculating the $\text{rss}^5$, a quality measurement of the fitted function can be obtained. The minimum $\text{rss}$ and thus the best-fitted function was a polynomial with the degree four. The expected degree of three, see equation 4.9 although had a very small $\text{rss}$ value. The optimal parameters of the power function $ax^b$ had a $b \approx 3.078$ and a small $\text{rss}$ as well. Table 4.4 shows the results for the different function candidates. $a, b, c, ..$ are fitted variables, $x$ is the number or states and $t$ the execution time.

4.5.4 alphabet length variation

The influence of the number of distinguishable messages is investigated in the known manner by creating random deterministic state machines with a different number of distinguishable messages or symbols. Again all FSM had 10 states and 20 transitions. In figure 4.11 the resulting learning time vs. the number of symbols is shown. Unfortunately, the execution time scattering is

---

5Residual sum of squares (the sum of the squared difference between the measured values and the fitted function)
4. State machine inference from a single positive data stream

<table>
<thead>
<tr>
<th>function prototype</th>
<th>rss</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t = ax^2 + bx^3 + c$</td>
<td>2.420</td>
</tr>
<tr>
<td>$t = ax^3 + bx^2 + cx + d$</td>
<td>0.840</td>
</tr>
<tr>
<td>$t = ax^4 + bx^3 + cx^2 + dx + e$</td>
<td>0.760</td>
</tr>
<tr>
<td>$t = ae^{bx}$</td>
<td>2.006</td>
</tr>
<tr>
<td>$t = ax^b$</td>
<td>1.114</td>
</tr>
</tbody>
</table>

Table 4.4: Different functions are fitted to the execution time vs. number of state dependency from the experiments. The low rss value from the polynomial and power functions, suggests a non exponential behavior.

too big to draw valid conclusions, (see figure 4.9). However, from the made experiments it seems that the influence of the number of different symbols plays a minor role for the execution time. The used data streams had one million symbols.

4.6 choice probability and timing

In addition to the topology of the state machine, one might be interested in some additional features of the modeled system. In general states, transitions or both might have known or unknown properties or dependencies. However, here only features of transitions shall be investigated. This is not a limitation since a state-property might be seen as a common feature of either all ingoing, all outgoing or all transitions from or to the state, and therefore, a state-properties can be transformed into a state with no properties and additional properties for appropriate transitions.

A very simple statistical property is the probability of taking a certain transition, if there are more outgoing transitions in a state. For example, there are two outgoing arcs from one state but one arc may be taken 90% of the time. Another simple property is a time delay a transition might have.

Those simple features might be "learned" by learning the automata the usual way first and then tracing the complete path represented by the data stream through the learned automata. By doing so, the probability of taking a certain arc can simply be calculated by counting how often every arc has been used. From this information, the transition probability for every arc can be estimated by dividing the transition’s occurrence by the total number
4. State machine inference from a single positive data stream

Figure 4.11: The theoretical linear dependency of the $L^*$ algorithm could not be found, because of the big variations of execution time of all taken outgoing transitions of the transition’s origin state. By a similar procedure, statistics over timing can be estimated. For example, if a time-stamp for every message of the data stream is known. The time-stamps from the messages can be mapped to transitions in the learned FSM. Using statistic tools, the distribution of the time needed for every transition can be calculated.
Chapter 5

Conclusions

5.1 summary

The use of FSM models in error detection systems for distributed systems seems to be a promising approach, especially when the needed FSM can be created automatically. The conducted experiments showed that this automatic creating of FSM models is possible for reasonable sized automata.

This approach does not need any expert with deep understanding of the system and can, in principle, be used for supervising any data stream. Using statistic tools a FSM model is generated, which most likely created the given data stream.

In the automobile, where bus systems like CAN bus or FlexRay are in use, the supervised data stream originates from wiretapping the communication between different electronic control units. These bus systems usually transmit the data packet orientated and every packet can be associated with a symbol and separated into sub streams. The separation would reduce the size of the learned model but is not essential for the FSM based fault detection approach and was therefore not explicitly evaluated here.

Overall, automatically creating state machines for fault detection purposes seems to be a promising approach worth being investigated further.

5.2 results

The implemented adapted $L^*$ algorithm showed that automata learning from a single positive data stream is, in principle, possible. The tests with different
5. Conclusions

complexity and size of state machines showed that the presented algorithm needs a certain length of the data stream in order to generate a correct machine. The algorithm performs in polynomial time in the data stream length and number of states (of the inferred FSM).

In particular, an algorithm creating a deterministic finite state machine from a single positive message stream, has been proposed. The resulting state machine can be represented by a strongly connected directed graph and thus is especially suited for modeling continuous working systems. The packet-orientated communication between the single components from a distributed system can be mapped to a message stream, which can be learned with the suggested algorithm and verified with the inferred state machine.

The feasibility of learning state machines from the available data stream is given, but since the algorithm failed to create a correct FSM for too short data streams the practical applicability in the future automobile fault detection systems depend on the practical available data and the needed system size. In section 5.3.1 further tests with real world examples are suggested to investigate the practical applicability.

5.3 future works

Many modifications of the original $L^*$ algorithm are known. The implementation of those modifications may improve the performance of the presented algorithm.

A drawback on the current algorithm, depending on the application, is that not for all data streams a correct machine is created. This is the case if too less data is available to infer a correct machine. It could be worthwhile to investigate if further modifications enable the learning of an automata from every possible data stream length. Introducing an additional property of the observation table, which forces the machine to include a path from every state to any other state, could solve this problem.

Besides these improvements, some thoughts on tests with real-world systems and better longtime dependency learning are mentioned in the following sections.

$^1$every state is reachable via a path from any other state
5. Conclusions

5.3.1 special test systems

The length of the needed data to infer a correct FSM depends strongly on the complexity of the system, which originally produces the data stream. The execution time of the proposed learning algorithm grows polynomial with the available data stream length. The economic consideration if this approach can be used in practice as a fault detection system in the automobile or other complex systems has to be investigated. The main points for the practice are:

- How can the learned model be verified? Is the model sufficiently detailed and not over-fitted?
- Is the needed amount of data available for learning an accurate model?
- Does the learning algorithm perform sufficiently fast for an economic application?

In order to answer these questions a study with real-world systems would be useful. This does not necessary require to take measurements from a real system. As a first step, a known model of such a system could be used as a test-system.

For example [2] studies the original Anilin algorithm [1] on random generated examples and on real-world system-models like buffers, vending machines, schedulers and communication protocols. The authors report that they failed to learn a huge ATM protocol (1715 states) due to memory issues, but successfully learned many other examples.

5.3.2 learning special long time dependencies

In the presented algorithm, the maximum length of test examples is limited by the probability of finding the test example in the available data stream. Moreover, this probability depends on the amount of test strings of a certain length, which the underlying state machine may create. In general it can be assumed that the amount of those creatable strings grows exponential with the length those test strings have. This property depends on the topology of the machine but any non-trivial closed\(^2\) machines with junctions exhibit this

\(^2\)every state may be reached again from any other state
exponential behavior. Therefore, the maximum length of test examples is logarithmic with the data stream length. Moreover, it may happen in machines with a lot of states that the needed data stream length is not practicable long in order to test examples, which include distant states. Distant states are states that need many hops (transitions) to reach each other. This feature does not necessarily limit the practical use of the automata but some long time dependencies are possible not discovered and thus an over-generalized FSM is created.

This fact motivates the idea of detecting dependencies between transitions without relying on concrete test examples. Following this idea, one has two main tasks. Finding the dependency and integrating the found knowledge into the FSM.

A possibility is focusing only on certain types of dependencies. For example, single decisions long-time dependencies like the one in the embedded Reber grammar figure 4.2. Meaning, at a single state the automata branches in equal and independent parts and then re-unites into a single state again. Using the Reber grammar example: Starting in the state after the $B$ the automata branches into two equal parts and re-unites after the next $E$ transition.

This kind of long-time dependency could be recognized by marking which transitions may be reached after a certain transition until reaching the transitions final state again. All transitions, not marked after testing the whole data stream have to be transitions of reuniting branches, split up at the origin state. This or other approaches may be implemented and tested (e.g. using multiple embedded Reber grammars from [8]).
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


Appendices
Appendix A

$L^*$ example run

In the following a run of the $L^*$ algorithm with a very simple example (figure A) is demonstrated in order to illustrate the working of the algorithm and to show the components of the observation table. The created data stream from this machine could be the following on:

...ABABABABAB...

![Finite State Machine](image)

Figure A.1: Example finite state machine

The observation table consists of two sections, the state section and the transitions section. The upper part of the table is the state section where every unique row represents a state from the FSM. The transitions’ section defines the transitions of the FSM. The basic steps to build up the correct observation table are:

- A field in the observation table is true if the row label concatenated by the column label is accepted by the machine, otherwise the file is false. If a field is true or false is decided by a membership query with the concatenated string.
A. $L^*$ example run

- If the observation table is not closed (the transitions/bottom part of the table contains a row different from all rows in the state/top part of the table) the row from the transitions’ part differing from all state rows is added as a state row and corresponding transition rows are added.

- If the observation table is not consistent (two state rows are equal but their corresponding transition rows differ) an additional column is added to the observation table such that the two inconsistent states have different results in the new column. The label of the new column is one letter from the alphabet concatenated by the label of an already existing column label.

- If a counterexample is found the counterexample and its prefixes are added into the state section of the observation table. Corresponding transitions are added in the transitions’ part.

- A FSM is built from an observation table by creating a state for every unique state row. Every state has a transition for every possible letter from the alphabet. The transitions row, labeled equal to the origins states label concatenated the transition’s letter, determines the destination state. The destination state is the state with the same state row as the mentioned transitions row.

States whose empty column ($\lambda$) if false are not final states and not created. Where $\lambda$ donates the empty symbol. In order to keep the observation table clearly arranged, transition rows that are equal to state rows are not shown.

<table>
<thead>
<tr>
<th></th>
<th>$\lambda$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>true</td>
</tr>
<tr>
<td>A</td>
<td>true</td>
</tr>
<tr>
<td>B</td>
<td>true</td>
</tr>
</tbody>
</table>

Table A.1: Initial observation table and corresponding FSM

Table A.1 shows to initial observation table of the minimal canonical automata. After a found counterexample "AA" is added, the observation table is shown in table A.2.
A. $L^*$ example run

<table>
<thead>
<tr>
<th></th>
<th>( \lambda )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda )</td>
<td>true</td>
</tr>
<tr>
<td>A</td>
<td>true</td>
</tr>
<tr>
<td>AA</td>
<td>false</td>
</tr>
<tr>
<td>AAA</td>
<td>false</td>
</tr>
<tr>
<td>AB</td>
<td>true</td>
</tr>
<tr>
<td>AAB</td>
<td>false</td>
</tr>
<tr>
<td>B</td>
<td>true</td>
</tr>
</tbody>
</table>

Table A.2: Inconsistent observation table after adding the counterexample AA

Table A.2 is inconsistent since the first row \( \lambda \) and the second row \( A \) are equal (true) but the successors are not. \( \lambda A \) and \( \lambda B \) are true but \( AA \) is false. For the table to be consistent \( \lambda A \) has to be equal to \( AA \) and \( \lambda B \) has to be equal to \( AB \). In order to make the table consistent a column labeled with \( A \) is added to the table and all free fields are queried.

<table>
<thead>
<tr>
<th></th>
<th>( \lambda )</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda )</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>A</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>AA</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>AAA</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>AB</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>AAB</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>B</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Table A.3: Closed and consistent observation table and corresponding FSM after adding column \( A \)

The FSM represented by the observation table A.3. The conjecture query with that FSM returns the counterexample \( BB \), which is not accepted by the correct machine.

Again, the table is inconsistent since the first and fourth row are equal but their corresponding transitions are not.
A. $L^*$ example run

<table>
<thead>
<tr>
<th></th>
<th>$\lambda$</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>A</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>AA</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>B</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>BB</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>AAA</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>AB</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>AAB</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>BA</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>BBA</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>BBB</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Table A.4: Inconsistent observation table after adding counterexample BB

$\text{row('}\lambda'\text{')} = \text{row('}B'\text{')} \text{ but } \text{row('}\lambda B'\text{')}! = \text{row('}BB'\text{')}$

Therefore, a column labeled 'B' is added to the observation table.

Table A.5 shows the final observation table and FSM. The not solid state corresponds to the $\lambda$ state row. It is the initial state from which every possible symbol stream is accepted. In a last step, this state is removed since it is redundant with the assumption that the FSM can start in any of the remaining states.
### A. $L^*$ example run

<table>
<thead>
<tr>
<th></th>
<th>$\lambda$</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>A</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>AA</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>B</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>BB</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>AAA</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>AB</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>AAB</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>BA</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>BBA</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>BBB</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Table A.5: Consistent and closed observation table after adding additional column 'B'
Appendix B

Maximal canonical automata example

The maximal canonical automata from which learning algorithms, for example, the sk-method start to create a FSM is shown in figure B.1. The sample strings, which were used to infer this automata, are:

ABAA
BA
ABA
BAB

Figure B.1: Example of a maximal canonical automata