Efficient Identification of Timed Automata

Theory and Practice

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.C.A.M. Luyben,
voorzitter van het College van Promoties,
in het openbaar te verdedigen op dinsdag 2 maart 2010 om 15:00 uur
door Sicco Ewout VERWER
informatica ingenieur
geboren te Papendrecht

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr. C. Witteveen

Samenstelling van de promotiecommissie:

Rector Magnificus	voorzitter
Prof.dr. C. Witteveen	Technische Universiteit Delft, promotor
Dr. M.M. de Weerdt	Technische Universiteit Delft, copromotor
Prof.dr. P. Dupont	Université catholique de Louvain
Dr. C. Costa Florêncio	Katholieke Universiteit Leuven
Prof.dr. P.W. Adriaans	Universiteit van Amsterdam
Prof.dr. F.W. Vaandrager	Radboud University Nijmegen
Prof.dr. R. Babuska	Technische Universiteit Delft

SIKS Dissertation Series No. 2010-16. The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

This research has been supported and funded by the Dutch Ministry of Economical Affairs under the SENTER program.

Cover design by Sjors van Roosmalen.

For Oscar

Acknowledgements

This thesis is the result of a four year long study into the interplay between learning theory, grammatical inference, and timed automata. I learned a lot in the past few years and I have many people to thank for it.

First, I would like to give great thanks to my supervisor Cees Witteveen for believing in me, supporting me, and giving me the opportunity to pursue an academic career. Dear Cees, I am first of all very grateful for the internship opportunity you offered me at Lockheed Martin in Texas. This internship lit a spark that resulted in me finishing my masters long before I had anticipated, and equally important, it triggered my interest in research and applications of theoretical computer science. I also thank you for you support and kindness, especially during the time that my father had gotten ill. The help you offered me during that difficult period means a lot to me. After obtaining my master's degree, you offered me a PhD position at TU Delft which I gratefully accepted. During this period, I am very grateful for the many suggestions and feedback that helped to develop this thesis into what it is today. In addition, I am especially grateful that you never stopped believing in me, even when our papers were not that well received by the research community. Last but not least, I thank you for your friendship during all these years. I am leaving Delft, but I hope that we will still meet on a regular basis. Dear Cees, although I have met and will meet many other great minds during my academic career, you have been and always will be my main role model.

Next, I give many thanks to Mathijs de Weerdt for his friendship, many suggestions and feedback, help with writing the software mentioned in this thesis, and being a fun office mate. Mathijs, I especially thank you for always being there for me to explain my often complicated and sometimes incorrect ideas. Testing these ideas has always been very helpful and a lot of fun. I also hope that our friendship won't suffer from the fact that I am leaving Delft.

I also give many thanks to colleagues from the research community that helped me to develop my ideas. I am especially grateful for the discussions with Christophe

Costa Florêncio on the topic of identification in the limit. Also the background chapter of his PhD. thesis on the same topic has been a lot of help in understanding this theory. I thank Pierre Dupont for our discussions on automaton identification and efficient implementations of the evidence-driven state-merging algorithm. I am also very grateful for the discussions with Pieter Adriaans and Maarten van Someren on grammatical inference, minimum description length, and potential applications of my techniques. Finally, I give many thanks to Eric Cator for explaining in detail the statistical techniques that are essential for the development of the algorithm for identification from unlabeled data.

I thank Frits Vaandrager and Julien Schmaltz for their interest in my techniques from a timed automata perspective and his suggestions to apply my technique to the interesting problem of model-based testing. I hope we can work together on such an application in the near future. I also thank Wil van der Aalst for his interest in my work, and hope that we will soon use grammatical inference techniques in order to perform process mining. I thank Toon Calders for noticing me and giving me a new job at TU Eindhoven. I hope we can get some additional funding and work together for at least a few more years and produce many interesting papers together.

Many thanks goes to the project group consisting of Joost, Tjay, Peter, Hugo, and Kees-Jan, without whom the application and the project funding would not have been possible. I hope we will reapply my techniques to the problem of identifying a real-time motor management system using more specialized preprocessing, and get this systems beyond its current state of proof-of-concept.

For making me enjoy every single day at the office, I thank my colleagues and friends: Cees, Mathijs, Hans, Henk, Tomas, Roman, Léon, Tamas, Adriaan, Renze, Jonne, Pieter, Chetan, Marijn, Paul, Jan-David, Jorik, Gertjan, Otto, and of course Yingqian. For their help with writing many of the scripts and communication interface used to obtain the results in this thesis I especially thank Jan-David, Jorik, and Otto. Also special thanks go to Jonne and Marijn for writing papers with me on topics outside the scope of this thesis.

Many thanks go to my many friends: Marten, Gert-Jan, Sonja, Eelse-Jan, Fien, Astrid, Sjors, Kim, Sean, Maarten, Arnoud, Eefje, Huibert-Jan, Corniel, Wan-Ruei, Sofia, Jia, Suu, Beibei, John, Wendy, Bas, Edward, for their friendship, talks we had, parties we shared, and games we played. Special thanks go to Sjors for designing the beautiful cover of this thesis.

I especially thank Jürgen Dix for being a very good supervisor for Yingqian, assisting her in finding a job at TU Delft, and coming all the way to the Netherlands to attend our wedding.

I thank my parents, brother, and sister, for always being there for me and helping me out in stressful times.

But most of all, I thank my lovely wife Yingqian, for supporting me, marrying me, and giving me a beautiful son Oscar. Yingqian, I simply do not know what I would do without you. Of course, I also thank her for proof-reading this entire thesis, and listening to my many strange thought processes.

Contents

List of Figures

CHAPTER 1

Introduction

1.1 A motivating example

Nowadays, there is more and more interest in technologies that increase the sustainability of existing processes. The main goal is to reduce the energy consumption and pollution of an existing process without increasing the cost of maintaining it too much. An example of such a process is freight transportation by trucks. The energy consumption of a truck is determined by two main factors: the engine and the driver. Both these factors currently result in a lot of *unnecessary* energy consumption:

- Engines are mass-produced for use in a certain geographical region. The settings of an engine are optimized for this region. Western-Europe is one such region. As a consequence, the engine settings of a truck are the same while driving in Switzerland (the Alps) and while driving in the Netherlands (highest mountain 322 meters). Driving on flat roads with an engine, which is optimized to be able to drive in the mountains, is of course not very efficient.

- Truck drivers currently receive little feedback regarding their style of driving. Hence, they have almost no incentive to adjust their driving style to be fuel-economic. As such, one can often see trucks pulling up too quickly and driving without cruise control.

We intend to reduce the energy consumption of trucks by *real-time monitoring* of the driving process by sensors on the truck. These sensors can measure almost everything we could possibly need: speed, engine thrust, pressure on gas and break pedals, temperature, fuel level, etc. By monitoring these values in real-time, we can select a different engine setting depending on the current road condition. In this way, the truck can get sufficient power in mountains and cities, and not too much power on flat highways and in traffic jams. In addition, we can give a signal to the driver whenever he or she is conducting driving behavior that is not fuel-economic.

In order to give this real-time feedback, we need (mathematical) *models* that can determine the current road condition and driving behavior. However, due to the lack of sufficient expert knowledge to construct these models by hand, we require techniques that can construct these models automatically from data collected by the different sensors. Such techniques are known as *system identification* techniques.

1.2 System identification

This thesis contains a study in a subfield of artificial intelligence, learning theory, machine learning, and statistics, known as *system (or language) identification.* System identification is concerned with constructing (mathematical) models from observations. A model is defined by the Cambridge Online Dictionaries[1] as:

> *Model*: a representation of something, either as a physical object which is usually smaller than the real object, or as a simple description of the object which might be used in calculation.

Essentially, a model is an intuitive description of a complex system. One of the main nice properties of models is that they can be visualized and inspected in order to provide *insight* into the different behaviors of a system. In addition, they can be used to perform different *calculations*, such as making predictions, analyzing properties, diagnosing errors, performing simulations, developing tests, computing control strategies, and many more. Models are therefore extremely useful tools for understanding, interpreting, and modifying different kinds of systems. In practice, however, it can be very difficult to construct a model by hand. Therefore, we are interested in automatically identifying models from data. In the Cambridge Online Dictionaries, identification is defined as:

> *Identify*: to recognize someone or something and say or prove who or what they are.

Identifying a model from data thus consists of using the data in order to recognize or prove that a system or process operates according to some model. The data consists of observations in the form of *event sequences* that are produced by the system or its surrounding environment. Intuitively, system identification tries to discover the logical structure underlying these event sequences. In practice, people discover this logic using a common sense rule-of-thumb: the simplest explanation for the observed events is assumed to be correct.[2] Automated system identification works in exactly the same way:

[1] http://dictionary.cambridge.org

[2] This rule is known as Occam's razor (see Section 2.2).

- Given a set of observed event sequences produced by a system or its surrounding environment, and a set of possible models for this system,
- find the simplest model in the set of possible models that explains all observed event sequences.

A model that is found (discovered) in this way can be seen as the common sense explanation for the observed events. System identification can be seen as a classic search procedure that tries to find this common sense explanation. Because this explanation is also a (mathematical) model, it can be used to give us insight into the inner workings of the system. In addition, an identified model can be used for the many different before-mentioned calculations.

In order to apply the system identification technique to our example of real-time monitoring of the driving process, we will first need discretize the sensor data into basic events such as: driving on the highway, speeding up, slowing down, breaking, signaling, using cruise control, etc. We can then identify models that use these events as input. In doing so we effectively identify models that describe the *languages* of engine settings and drivers. This approach of identifying a language model in order to describe different behaviors in data is known as syntactic pattern recognition (Fu 1974). One of the main reasons for using such a method is that we can use the identified language models not only to find out whether a driver is driving economically, but also to give insight into the exact road behavior of (non-)economic drivers, i.e., to learn what feedback to give to drivers in order to reduce their fuel usage.

1.3 Timed automata

A well-known model for characterizing systems is the *deterministic finite state automaton* (DFA, see e.g., (Sipser 1997)). An advantage of an automaton model is that it is easy to interpret. In contrast, many other well-known models that are commonly used to identify the behavior of a system are quite difficult to interpret, for example neural networks or support vector machines, see, e.g., (Bishop 2006). Such models can only be used to classify new data and to predict the future behavior of a system. The ability to analyze an identified model is one of the main reasons why we are interested in identifying automaton models from observations.

A DFA is a language model. Hence, its identification (or inference) problem has been well studied in the grammatical inference field, see (de la Higuera 2005). Knowing this, we want to take an established method to learn a DFA and apply it to our event sequences. However, when observing a system there often is more information than just the sequence of symbols: the *time* at which these symbols occur is also available. A straightforward way to make use of this timed information is to *sample* the timed data. For instance, a symbol that occurs 3 seconds after the previous event can be modeled as 3 special time tick symbols followed by the event symbol. This results in an automaton model that models time *implicitly*, i.e., using states. A disadvantage of such an approach is that it can result in an exponential blowup of both the input data and the resulting size of the model. In this thesis, we propose a different method that uses the time information directly in order to produce a timed model.

To be able to use the timed information directly, we require a DFA variant that includes the notion of time. A *timed automaton* (TA) (Alur and Dill 1994) is such a variant. TAs are commonly used to model and reason about real-time systems, see e.g., (Larsen, Petterson and Yi 1997). A TA contains objects that record the time that has elapsed since some specific event, known as *clocks*, and constraints on the time values of these clocks, known as *clock guards*. Using these clocks and clock guards, the language of a TA does not only depend on the types of events occurring, but also on their time values relative to previous event occurrences. In this way, TAs can for instance be used to model deadlines in real-time systems. The clock guards of a TA model these timing requirements *explicitly*, i.e., using numbers. Because numbers use a binary representation of time, and states use a unary representation of time, such an explicit representation can result in exponentially more compact models than an implicit representation. Therefore, also the time, space, and data required to identify TAs can be exponentially smaller than the time, space, and data required to identify DFAs. This efficiency argument is our main reason for modeling systems using TAs.

For our motivating example, the application of a TA identification technique will results in identification of the *timed languages* of engine settings and drivers. Thus, we not only identify a language over the basic events such as speeding up and slowing down, but also over their times of occurrence. These times of occurrence between vehicle speedups and slowdowns can be very significant for determining the road condition and driving behavior. For instance, a sequence of fast changes from slowing down to speeding up and vice versa indicates driving in a city, while a sequence of slow changes indicates driving on a freeway. An additional benefit of modeling these occurrence times explicitly is that it results in more succinct and thus more insightful models.

1.4 Related work

The problem of identifying a DFA from a data set is a well-studied problem in learning theory (de la Higuera 2005). There are, however, very few studies on the identification of a TA from data. Closely related work deals with the problem of learning event recording automata (ERAs), which is a restricted but still powerful class of TAs (Grinchtein, Jonsson and Petterson 2006). However, their identification algorithm requires an very large amount of data and is therefore difficult to apply in practice.

Other approaches to the problem of identifying a timed system mainly deal with the identification of probabilistic timed models, such as continuous time Markov chains (Sen, Viswanathan and Agha 2004). These probabilistic models are less powerful than TAs: they model probability distributions over time values instead of timing constraints such as deadlines. More specifically, in a TA, the meaning of an event can change over time. The occurrence times of events thus influence which events can occur in the future. In these probabilistic timed models, only the probability of an event can change over time. This does not influence the possible future events.

To the best of our knowledge, the learning methods for more complex probabilistic timed models only deal with the problem of optimizing the parameters of already known model structures. An example is a method for learning the parameters of (hidden) semi-Markov models (Guédon 2003). We aim to develop methods that identify the model structure in addition to the model parameters.

Closely related research comes from the temporal data-mining field (Roddick and Spiliopoulou 2002). In temporal data-mining, the objective is to discover previously unknown rules from time-series data. Moreover, these rules should be easy to understand and to validate. This is very much like the problem we are trying to solve. In temporal data-mining, however, the focus is on finding simple patterns or correlations in the data, but not on finding a more complex model for the actual system that generated this data.

1.5 Contributions of this thesis

The work in this thesis makes four major contributions to the fields of artificial intelligence, learning theory, machine learning, and statistics:

1. It contains a thorough theoretical study of the complexity of identifying TAs from data.
2. It provides an algorithm for identifying a simple TA from labeled data, i.e., from event sequences that are known to characterize some specific system behaviors.
3. It extends this algorithm to the setting of unlabeled data, i.e., from event sequences with unknown behaviors.
4. It shows how to apply this algorithm to the problem of identifying a real-time monitoring system.

We now discuss these four contributions in more detail.

Theory We study the problem of identifying a TA in the paradigm of identification in the limit. The focus of this study is on the efficiency of identifying TAs from data. The contributions of this study are summarized in the following list:

- We prove that a class of automata can be identified efficiently only if it is polynomially distinguishable (Lemma 3.9);
- We prove that deterministic TAs (DTAs) with two or more clocks are not polynomially distinguishable (Proposition 3.8 and Corollary 3.10), and hence that DTAs are not efficiently identifiable (Theorem 1);
- We prove that DTAs with a single clock (1-DTAs) are polynomially distinguishable (Theorem 4), and use this to prove that 1-DTAs are efficiently identifiable (Theorem 5);
- We provide an algorithm ID_1-DTA (Algorithm 3.1) for identifying 1-DTAs efficiently.

- We show that 1-DTAs and n-DTAs (DTAs with n clocks) are language equivalent (Theorem 6).

These contributions are of importance for anyone who is interested in identifying timed systems (and DTAs in particular). Most importantly, the efficiency results tell us that identifying a 1-DTA from timed data is more efficient than identifying an equivalent DFA. Furthermore, the results show that anyone who needs to identify a DTA with two or more clocks should either be satisfied with sometimes being inefficient, or he or she has to find some other method to deal with this problem, for instance by identifying a subclass of DTAs (such as 1-DTAs). In general, our results are statements regarding the expressive power of one-clock and multi-clock DTAs. These statements are important by themselves, i.e., not necessarily restricted to just the identification problem. We believe that there can be other problems (such as reachability analysis) that may benefit from our results.

Algorithms for labeled data Based on the theoretical results, we develop a novel identification algorithm RTI (Algorithm 4.5) for a simple type of 1-DTA, known as a deterministic real-time automaton (DRTA). These automata can be used to model systems for which the time between *consecutive* events is important for the system's behavior. The main reason for restricting ourselves to DRTAs and not to the full class of 1-DTAs is that we believe them to be expressive enough for many interesting applications, including modeling truck driver behavior. The input for RTI should be labeled data, i.e., the data should consist of positive event sequences that characterizes the (correct) behavior of the system, and negative event sequences that do not characterize this behavior (or that characterize faulty behavior). The current state-of-the-art for DFA identification from labeled data is the evidence driven state-merging (EDSM) algorithm, see, e.g., (de la Higuera 2005). RTI is a modification of the EDSM algorithm to the setting of DRTAs. To the best of our knowledge, ours is the first algorithm that can identify a TA model from timed data. In addition, it does so efficiently in time (Proposition 4.3), and it converges efficiently to the correct DRTA in the limit (Proposition 4.6).

We evaluate the performance of RTI experimentally on artificially generated data. The results of these experiments show that RTI performs sufficiently good when either the number of distinct events, or the number of states is small. In addition, the experimental results show that RTI outperforms the straightforward sampling approach in combination with EDSM if the data is indeed produced by a timed system. In other words, we show that if a system can be modeled efficiently using a DRTA, then it is less efficient and much more difficult to identify an equivalent DFA correctly from this data.

Algorithms for unlabeled data We adapt the RTI algorithm to the more frequently occurring setting of unlabeled data, i.e., only positive data. The result of the adaptation is the RTI+ algorithm (Algorithm 5.1). The RTI+ algorithm is a polynomial time algorithm that converges efficiently to the correct probabilistic DRTA (PDRTA) in the limit with probability 1 (Propositions 5.5 and 5.6). RTI+ uses statistical tests in order to identify PDRTAs. Although these tests are designed specifically for the purpose of identifying a PDRTA from unlabeled data, they can be modified in order to identify probabilistic DFAs. Hence, they also contribute to the current state-of-the-art in probabilistic DFA identification.

We tested the performance of the RTI+ algorithm with different statistical tests on artificially generated data. An interesting conclusion of these experiments is that in terms of data requirements it is often easier to identify a model than to set its parameters (event probabilities) correctly. Because of this, we argue that the traditional quality measures for probabilistic models are unsuited for testing the quality of the identified models, independent of its parameters. Therefore, we propose a different way to compute the quality of an identified PDRTA that separates the problem of tuning the parameters from the problem of identifying the model.

Our results show which of the introduced statistics achieves the best performance. The achieved performance using this statistic is shown to be sufficient in order to apply RTI+ in practice.

Applying the algorithms in practice We apply our algorithms to the problem of identifying a real-time monitoring system for driver behavior. Due to the lack of sufficient expert knowledge, the input for our algorithm consists of unlabeled event sequences. These sequences are recorded during 15 complete round trips from the Netherlands to Switzerland or England and back. From this data, we identify PDRTA models using RTI+. Then we collect a small amount of examples of an interesting example driving pattern: pulling-up too fast or normally from a traffic light. We use these labeled examples to label some states of the identified PDRTA models. In this way, we can identify a monitoring system even when it is difficult to obtain labeled data. The resulting PDRTA models are shown to be useful for monitoring whether the driver is pulling-up too fast. This application serves as a proof of concept of our techniques.

1.6 Potential applications

The techniques we develop in this thesis have many possible practical applications. The most straightforward application of our techniques occurs when one requires insight into a process that produces time-stamped events. Gaining insight into how such a system operates can be helpful in many settings, for instance the design, analysis and control of such systems, see, e.g., (Cassandras and Lafortune 2008). Insight into real-time systems can be gained using our RTI+ algorithm: given some observations of a process, this algorithm can be used to identify a PDRTA model for this process. The structure of the identified model can unravel previously unknown facts about (and other properties of) the process. Such an approach is common in the fields of syntactic pattern recognition (Fu 1974) and process mining (van der Aalst and Weijters 2005).

In other applications, one might not be very interested in the process structure itself, but just want a model that decides whether the process is displaying good or bad behavior. In such a case, one first needs to find a way to label some observed sequences as being either good or bad; afterwards one can use our RTI algorithm in order to find a DRTA model that can be used to make this decision.

This is similar to the standard machine learning approach of *classification*, see, e.g., (Bishop 2006). However, in addition to learning a classifier, our algorithms identify insightful models, i.e., models that can be interpreted by visualizing them. This visualization can lead to a better understanding of the underlying process. In contrast, most of the standard machine learning techniques do not result in insightful models. Instead, they result in for example high-dimensional hyperplanes, see, e.g., (Bishop 2006).

In the field of timed automata, there currently do not exist many identification techniques. Timed automata are commonly used as models for real-time systems such as network protocols, see, e.g., (Larsen et al. 1997). There exist many techniques that can be used to test whether such models satisfy certain desirable properties, such as dead-lock freeness. This is known as *model checking* (or verification) and most of the research regarding timed automata is focussed on this problem, see, e.g., (Alur and Madhusudan 2004). However, when the exact timed automaton model is unknown, identifying these models becomes an important issue. Besides this main motivation, there are many other examples of cases where identifying timed automata contributes to model checking (Leucker 2007).

1.7 Overview

This thesis is divided into chapters based on their contributions as follows:

Chapter 2. We begin this thesis with an explanatory survey of the necessary background on learning theory, discrete event systems, and language identification. In addition, we survey related work that deals with the identification of some kind of timed model.

Chapter 3. Here, we prove our theoretical results regarding the efficiency of identifying TAs from data, identifying 1-DTAs from data, and on the power of clocks in DTAs. We also give an efficient algorithm for the identification of 1-DTAs from labeled data, and show how it can be adapted to identify n-DTAs.

Chapter 4. In this chapter, we describe the problem of identifying DRTAs from labeled data, prove that this problem is still hard (NP-complete), provide a novel algorithm (RTI) for solving this problem, prove its properties, provide several possible heuristics for RTI, and evaluate them on artificially generated data. In the experiments, we compare the performance of RTI with the straightforward sampling approach.

Chapter 5. In this chapter, we provide the RTI+ algorithm, which is an adaptation of RTI to the problem of identifying PDRTAs from unlabeled data. We provide several useful statistics that can be used by RTI+ and evaluate their performance on artificial data. In addition, we propose a way to compute the quality of an identified PDRTA that separates the problem of tuning the parameters from the problem of identifying the model.

Chapter 6. Here, we describe how to apply the RTI+ algorithm to the real-world problem of identifying truck driver behavior, and discuss the obtained results.

Chapter 7. We end this thesis with a concluding overview and some ideas for future work that builds on the proposed techniques and results.

CHAPTER 2

Background on inductive inference, discrete event systems, and language identification

2.1 Introduction

This chapter contains an explanatory survey of inductive inference, discrete event systems, and language identification. In this survey, we explain all of the concepts and notions that are required to understand the contents of this thesis. In addition, we give an overview of the techniques and the current state-of-the-art in each of these fields that are relevant for the problem of identifying timed automata. The survey can be read without any prior knowledge of any of these fields. However, we do assume the reader to be familiar with the basic notions of complexity theory, see, e.g., (Sipser 1997).

This chapter is split into three sections, one for each topic. The sections on these topics can be read independently, and skipped if necessary. In the main text of this thesis, we refer to the relevant background knowledge from this chapter whenever it is required. However, we encourage readers new to some of these three fields to read the respective sections. At the end of this chapter, we give an overview of related work that deals with the identification of timed models.

2.2 Inductive inference

Inductive inference is the theory of learning (also called inferring or identifying) a concept from examples.[1] The main idea is that of a *student* who is given examples of some concept by a *teacher*. The student tries to infer a *hypothesis* from the given

[1] Throughout the text, the verbs “to learn” and “to identify” are used interchangeably.

examples. The teacher draws the examples from a set called the *instance space*, sometimes called *input space*. We think of this space as being a set of instances, i.e., objects or observations, in the student's world. A *concept* is a subset of this instance space; this can be thought of as the set of all positive instances of some interesting rule. In the learning model we consider, the student has access to both positive and negative examples of such a concept. The hypothesis that the student tries to infer is also a subset of the instance space, and hence also a concept. The student has learned a target concept when the hypothesis and the target concept are the same subset of the instance space.

In learning theory, it is common practice to infer a hypothesis of minimal size. The reason for this is the idea that the smallest (or least complex) hypothesis, among all hypotheses that can explain a certain phenomenon, is most likely to be the correct one. This is a reformulation of the well-known principle of *Occam's razor*. This principle is intuitively very appealing, and it is commonly used as a heuristic in science. In learning theory several principles exist that incorporate Occam's razor in a different way. Examples are the *Minimum Description Length* (MDL) principle, see e.g. (Grünwald 2007), and the *Empirical Risk Minimization* (ERM) principle, see e.g. (Vapnik 1998). MDL is based on the idea that any regularity in the examples can be used to compress the examples, i.e., to describe it using fewer symbols. Within the MDL framework, the best solution to a learning problem is the hypothesis (or combination of hypotheses) that achieves the best compression of the examples. In ERM the idea is to obtain the smallest possible bound on the error made in future predictions, by minimizing the number of training errors (empirical risk).

For the construction of learning algorithms it is of course very important which measure to choose as learning criteria (or *inductive bias*), since this defines which hypothesis should be inferred. We will always try to infer a hypothesis of minimal size. In the following, we first explain the basics of inductive inference theory. Then, we discuss the three most common frameworks for learning problems. We end this section with an explanation of efficiency and complexity of learning problems. Most of the text in this section is based on the books (Kearns and Vazirani 1994) and (Jain, Osherson, Royer and Sharma 1999), and the survey chapter (Goldman 1999).

2.2.1 Learning basics

In each learning model we have a student s that tries to infer a hypothesis H. The student is given examples from an instance space X by a teacher t that tries to teach a concept C. In this section we give definitions of the notions of a concept, a student, and a teacher. We clarify the use of these notions by an example of an algorithm for the inference of boolean formulas.

Definition 2.1. *(concept) A* concept *C over an* instance space *X is a subset of X. We say that an example x is a* positive example *of C if $x \in C$, it is a* negative example *otherwise.*

Definition 2.2. *(hypothesis) A* hypothesis *H for a target concept C over an* instance space *X is a subset of X. We say that x evaluates to true in H if $x \in H$,*

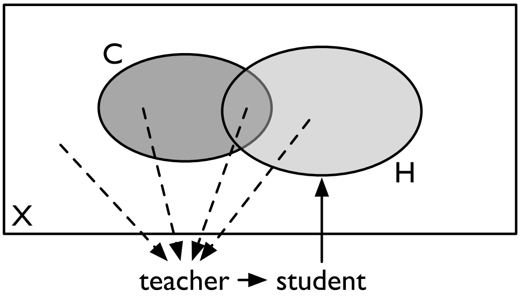


Figure 2.1: Learning. A teacher for a target concept $C \subseteq X$ gives an example from the instance space X to the student. This example is one of four types: true positive, true negative, false positive, or false negative. The student uses the example to adjust its hypothesis $H \subseteq X$.

otherwise it evaluates to false. A hypothesis H for a target concept C is said to be correct *if every example $x \in X$ evaluates to true in H if and only if it is a positive example of C.*

Example 2.1. Suppose we are given a set $V = \{v_1, \ldots, v_n\}$ of n boolean variables, and let $X = \{0,1\}^V$ be the set of all assignments to these variables. We can consider a concept C over X whose positive examples are exactly the satisfying assignments of some boolean formula f over V. Such a concept can be represented by a boolean formula f_c in disjunctive normal form (DNF) that has a small number of terms (the concept). A teacher can try to teach such a formula to a student. This student can then try to learn a minimal DNF formula f_h (the hypothesis).

Definition 2.3. *(teacher) A* teacher *t of a target concept C is an oracle that returns a* labeled example *(x, b), where b is a boolean value which is true if $x \in C$, and false otherwise.*

Definition 2.4. *(student) A* student *s is a learning algorithm, which learns a* hypothesis *H and has access to a teacher t of target concept C. The* goal *of a student is to find a correct hypothesis, i.e., a hypothesis H such that $x \in H$ if and only if $x \in C$.*

Figure 2.1 gives a graphical overview of learning. In Example 2.1, the student s (that tries to learn the minimal DNF formula) could start with a hypothesis formula such as $f_h = v_1$. This hypothesis is the smallest formula in DNF which satisfies all positive assignments known by s (currently none). It uses a function next to obtain the DNF formulas: given a formula in DNF, next returns the next smallest DNF formula. This function defines a *total order* on DNF formulas based on their size, and thus gives us the capability to enumerate these formulas from small to large. The trick is that the next function can be used to generate hypotheses until one agrees with all the examples seen so far. This hypothesis is the smallest boolean formula in DNF which satisfies the positive examples and does not satisfy the negative examples. This algorithm is a form of *identification by enumeration*. The algorithm is shown in Algorithm 2.1.

Algorithm 2.1 Identification of DNF formulas

```
Require: A teacher $t$ for $C$ over an instance space $X = \{0,1\}^n$, and a function next
  which returns the next smallest DNF formula
  Let $H$ be the smallest hypothesis: $f_h := v_1$ and $H := \{x \in X \mid f_h(x) = true\}$
  Start with an empty set of positive and negative examples $S = (S_+ := \emptyset, S_- := \emptyset)$
  while true do
    Get the next example: $(x, b) = t()$
    if $b = true$ then set $S_+ := S_+ \cup \{x\}$ else set $S_- := S_+ \cup \{x\}$
    while $H$ is inconsistent with $S$: $S_+ \not\subset H$ or $S_- \cap H \neq \emptyset$ do
      Get the next smallest hypothesis: $f_h :=$ next$(f_h)$ and $H := \{x \in X \mid f_h(x) = true\}$
    end while
    Output $H$ (the current smallest consistent hypothesis)
  end while
```

Note that Algorithm 2.1 is not a very efficient algorithm: before a consistent hypothesis is found, every possible smaller DNF formula is tried. An enumeration over all these DNF formulas will take an enormous amount of time. However, it is easy to prove that the student will eventually converge to the smallest *correct* DNF formula, i.e., f_h will be such that $C = \{x \mid f_h(x) = true\}$. This is called *identification in the limit* and will be formally defined in the next section. Although the student in Algorithm 2.1 is not very efficient, it can be used as a basis for a more efficient student. Ideally we would like such a student to be:

1. *Quick*: The number of calls to teacher t is small.
2. *Efficient*: The amount of computation performed is small.
3. *Correct*: The student outputs a correct hypothesis.
4. *Incremental*: The student only has to store a small amount of examples.
5. *Finite*: There is a way to tell how close the student is to converging.
6. *Robust*: Noise does not cause the student to fail.

Algorithm 2.1 only satisfies the correctness property. Another important issue for a student is the space of possible concepts it is trying to infer, and how a concept from this space is represented by a hypothesis. The space of possible concepts is viewed as a set of concepts over the input space, called the *concept class*. In our learning model the student will have access to positive and negative examples of an *unknown* concept, chosen from a *known* (fixed) concept class. Usually, a student tries to infer a hypothesis concept from the known concept class. However, this is not necessary, it is possible to use a different *representation* of a concept. For instance, in Example 2.1, the student could use conjunctive normal form (CNF) formulas instead of DNF formula as a representation for boolean formulas. A concept class and a representation are formally defined as follows:

Definition 2.5. *(concept class) A* concept class $\mathcal{C}$ *over an instance space* X *is a recursively enumerable set of concepts over* X.

Definition 2.6. *(representation) A* representation scheme *for a concept class $\mathcal{C}$ is a function $\mathcal{R} : \Sigma^* \to \mathcal{C}$, where Σ is a finite alphabet of symbols. We call any string $r \in \Sigma^*$ such that $\mathcal{R}(r) = C$, a* representation *of C (under $\mathcal{R}$).*

The notion of a representation is very important in learning theory because the chosen representation has an impact on the complexity of learning. For example, it is possible that a learning problem which is intractable using one representation becomes tractable using another representation as hypothesis. This has to do with the notion of tractability of learning problems and will be explained in Section 2.2.3.

2.2.2 Three learning frameworks

In inductive inference theory there exist three main learning frameworks. All three of these pose different requirements on the learning process and have access to different resources:

- *Identification in the limit*: eventually learn a correct hypothesis from a teacher that eventually supplies every example (Gold 1967).
- *Query learning*: learn a correct hypothesis from a teacher that answers specific questions (queries) (Angluin 1988).
- *PAC identification*: learn a probably approximately correct (PAC) hypothesis from a teacher that samples examples from an arbitrary probability distribution (Valiant 1984).

Of course there exist many variations of each of these frameworks. The choice of which learning framework to adopt depends on the application. In most applications it is easy to obtain a set of examples of some target concept. In these cases, both identification in the limit and PAC identification can be applied. PAC identification poses more strict requirements on the student than identification in the limit. Constructing an algorithm that satisfies these requirements is difficult, but the result is that the student does approximate the correct hypothesis with high probability. In contrast, a limit student is only guaranteed to learn the correct hypothesis in the limit.

Obtaining a teacher that answers specific questions as in query learning is more difficult than obtaining a set of examples. A possible setting is when domain experts need to build a system model but have difficulties with writing down their knowledge in logical rules. In this case, a query student could be used to infer these rules by asking questions to the experts.

Instead of just choosing one framework it is also possible to combine them. For example, it is possible to construct a learning algorithm that approximately correctly identifies some concept using a finite sample and membership queries. When both a teacher and a finite sample are available you might as well use them both. In general, one should use all the available resources for a learning problem. Learning is a difficult problem, and additional information (whether in the form of data or expert knowledge) only makes solving this problem easier.

In the remainder of this section we discuss all three of these frameworks in more detail. Our discussion begins with identification in the limit. We then explain query learning, followed by PAC identification.

Identification in the limit

The identification in the limit framework (Gold 1967) views learning as an infinite process in which the student is allowed to change its mind (hypothesis) arbitrarily often. A comprehensive overview of identification in the limit is given in (Jain et al. 1999).

Identification in the limit is an extension of the simpler notion of *finite identification*, in which the student cannot change its mind and just outputs one hypothesis:

Definition 2.7. *(finite identification) A student s* finitely identifies *a concept C from a teacher t when after a finite number of calls to t it outputs a correct hypothesis H and halts.*

This type of learning is also known as *one-shot learning* or the *FIN paradigm*. The finite identification model is extended by allowing the student to change its mind after it has output a hypothesis. This new model is more like the human kind of learning: first learn a part of a concept, then extend it by learning more details. When the student is allowed to change its mind infinitely often without any additional constraints there is no way to determine whether it has learned the concept or not. That is why the extended model also requires the property of *convergence*. This model is called *identification in the limit*:

Definition 2.8. *(identification in the limit) A student s* identifies *a concept C* in the limit *from a teacher t, when it outputs a sequence of hypotheses $H_1, H_2, \ldots$, and there exists a number $n \in \mathbb{N}$ such that from that point on the output hypothesis remains the same ($H_n = H_{n+1} = H_{n+2} = \ldots$) and H_n is a correct hypothesis.*

Algorithm 2.1 is an example of a student that identifies the concept of DNF formulas in the limit. Note that the only difference with finite learning is that we allow an infinite amount of mind changes. A *mind change* is a point in the hypothesis sequence where the output hypothesis is changed. Because a mind change requires the student to compute a new hypothesis, the amount of mind changes is a common measure for the computational complexity of a learning problem. In addition, the run-time required to compute this new hypothesis is of course relevant for this complexity. When a student s is guaranteed to identify any correct hypothesis H from a (representation of a) concept class $\mathcal{C}$ using a polynomial amount of mind changes, and polynomial run-time in the size of the H, then we say that s identifies $\mathcal{C}$ *efficiently in the limit*.

In identification in the limit, there is a big difference between teachers that return labeled, and those that return unlabeled examples. In the first case, we say that a student learns a hypothesis from *informant*. In the second case, the student learns the hypothesis from *text*. Many concept classes are learnable from informant but not from text. For instance, the regular languages are identifiable in the limit from informant, but not from text (Gold 1967). In practice, however, it is possible to simulate labels using the unlabeled examples. For instance, in practice it is commonly assumed that two examples have the same label if they display similar (statistical) properties (or features). If the probability that this simulation is incorrect goes to 0 if the number of examples goes to ∞, we say that a student identifies a concept class *in the limit with probability* 1.

It is also possible to include the notion of probabilities directly in the identification in the limit framework. These are included by allowing the student to be a randomized algorithm. The resulting identification model is called *probabilistic identification.*

Definition 2.9. *(probabilistic identification) A probabilistic student s* probabilistically identifies *a concept C from a teacher t with probability p, when the probability that s identifies C from t is at least p.*

This definition can be applied to the identification models above by replacing the word 'identifies' into 'finitely identifies' or 'identifies in the limit' in Definition 2.9. An interesting notion which is closely related to probabilistic identification is that of *team identification.* In team identification a set (team) of students try to solve the learning problem together. In such a team a (usually small) number of students is allowed to produce an inconsistent hypothesis.

Definition 2.10. *(team identification) A finite set $S = \{s_1, s_2, \dots, s_n\}$ of students* [r,n]team identifies *a concept C from a teacher t if at least r of the n students identify C from t.*

The main motivation for team learning is the result that there are collections of identifiable languages which are not closed under union. In other words, there are collections of languages that are identifiable, but the union of these languages is not identifiable. Team learning is also a nice way to distribute the learning problem among several students. For example in an agent-based system each student can be specialized in learning some subconcept. The learning process is correct when the set of students (agents) team identify the target concept.

There exist many other interesting modifications of the identification in the limit framework. A noteworthy example of this is that of *conservative inference*: the student is only allowed to make justified mind changes. A mind change is *justified* when the current hypothesis is contradicted by an example. Interestingly, there are examples of learning problems that are not learnable by a conservative student, but that are learnable by an non-conservative student (Wiehagen and Zeugmann 1995). For more information on finite learning, identification in the limit, team identification, and all kinds of modifications see (Jain et al. 1999).

Query learning

In the query learning framework (Angluin 1988), learning is seen from a different perspective: that of a student asking questions. This requires a teacher that is capable of answering these questions. Instead of just treating the teacher as an oracle, a student now calls it with a *query* as an argument. This type of learning is also known as *active learning.* Learning without queries is sometimes called *passive learning.*

These are four commonly used queries:

- *Membership queries*: The teacher takes as input an element $x \in X$ and returns *true* if $x \in C$, and *false* otherwise.
- *Equivalence queries*: The teacher takes as input a hypothesis $H \subset X$ and the output is either *yes*, if H is correct, or a pair (x, b) otherwise, where $x \in H \triangle C = (H \setminus C) \cup (C \setminus H)$ is an instance from the symmetric difference of H and C, and b is a boolean value indicating whether $x \in C$.
- *Subset queries*: The teacher takes as input a hypothesis $H \subset X$ and the output is either *yes*, if $H \subset C$, or x otherwise, where $x \in H \setminus C$ is an instance from the difference between H and C.
- *Superset queries*: The teacher takes as input a hypothesis $H \subset X$ and the output is either *yes*, if $H \supset C$, or x otherwise, where $x \in C \setminus H$ is an instance from the difference between C and H.

The instances returned by equivalence, subset, and superset queries are called *counterexamples*. A teacher capable of answering both membership and equivalence queries is also called a *maximally adequate teacher*. When designing a student which makes use of queries, an important question to ask is whether the used queries are available to the student. Membership queries are available in many settings but equivalence queries usually are problematic. For example, answering an equivalence query for the DNF formulas concept class is NP-hard (Bshouty, Cleve, Gavaldà, Kannan and Tamon 1996). Also the subset and superset queries are computationally hard. Just one of each of these can be used to simulate an NP-oracle (Bshouty et al. 1996). Still, these queries are widely used in query learning research because domain experts are sometimes able to provide them.

Formally query learning can be defined as follows:

Definition 2.11. *(query learning) A student s* identifies *a concept C from a* query teacher t_q*, when it uses only the answers from queries to* t_q *in order to output a correct hypothesis H.*

In query learning the most commonly used measure of complexity is the amount of queries required to identify a concept. In addition, the run-time required by the student is of course important for the complexity. When a student identifies any hypothesis H from a concept class $\mathcal{C}$ correctly using polynomial run-time and a polynomial amount of queries in the size of H, the student is said to identify $\mathcal{C}$ *efficiently from queries*. Algorithm 2.2 shows a classic algorithm by Angluin (Angluin 1988) which efficiently learns k-CNF formulas using only equivalence queries. A k-CNF formula is a formula in conjunctive normal form in which each clause consists of at most k literals.

Since there are at most $(2n+1)^k$ clauses over n variables with at most k literals, Algorithm 2.2 runs in time polynomial in n^k. The algorithm is an implementation of a general technique for equivalence query algorithms, known as *majority vote*. A majority vote is possible when the input to an equivalence query is not required to be a member of the hypothesis space. The idea is then to maintain a set S of all the indices of hypotheses compatible with all counterexamples seen so far. We gather from S a set of elements M_S such that for each $x \in M_S$, x is an element

Algorithm 2.2 Identification of k-CNF formulas using equivalence queries

```
Require: A query teacher $t_e$ of a target concept $C$ that answers equivalence queries
  (denoted $t_e(H)$)
Ensure: $H$ is a correct hypothesis
  Hypothesis $H := \emptyset$
  $f_h$ is the conjunction of all clauses over $n$ variables with at most $k$ literals
  while The hypothesis is incorrect: $t_e(H) \neq yes$ do
    Get the next counterexample: $x = t_e(H)$
    for All clauses: $c \in f_h$ do
      if Clause $c$ is inconsistent with $x$ then
        Remove $c$ from $f_h$
      end if
    end for
    Set $H := \{x \mid f_h(x) = true\}$.
  end while
  Return $H$
```

of at least half of the hypotheses indexed by S. Then we perform an equivalence query with M_S as input. Let x' be the received counterexample. If $x' \in M_S$, remove from S all indices of hypotheses that contain x'; otherwise, remove from S all indices of hypotheses that do not contain x'. In this way every answered query reduces the cardinality of S by at least one half, and therefore only $log((2n+1)^k)$ queries are needed to identify the target concept.

To see why Algorithm 2.2 implements the majority vote technique, we need to make the observation that the set of hypotheses consistent with the examples seen so far consists of every formula that is a conjunction of some subset of the non-eliminated clauses. When a counterexample eliminates some clause c, then for every non-eliminated clause it eliminates the hypothesis that could be made by conjoining with c. Thus each counterexample eliminates more than half of all possible hypotheses.

In the query learning framework several exact algorithms are known for interesting learning problems. In Section 2.4 we will describe a well-known query algorithm for the exact inference of deterministic finite automata. As we stated before, a problem with these algorithms is that of the computational hardness of most queries. However, it is also known that any concept which is identifiable using equivalence queries is also PAC identifiable (Angluin 1988). The basic idea is to simulate an equivalence query by using random samples to search for a counterexample to the current hypothesis. When there is a sufficient amount of examples this method can be used to obtain a hypothesis with a high probability of having a small error.

PAC learning

The probably approximately correct (PAC) learning framework (Valiant 1984) is based on the idea of a probabilistic teacher, which chooses examples based on an arbitrary probability distribution. Unlike the previous two frameworks, in which the goal of learning is to exactly identify the target concept, the goal of learning in the PAC framework is to classify any new examples with high accuracy. In the

PAC framework a probabilistic student is given the following inputs:

- A probabilistic teacher t that draws examples according to an unknown probability distribution $\mathcal{D}$.
- A value ϵ, which denotes the maximum allowed probability of error.
- A value δ, which denotes the minimum allowed probability of not being successful.

Using these inputs it is the task of a PAC student to identify, with probability at least $1 - \delta$, a hypothesis H that has a probability of at most ϵ of disagreeing with the target concept on a new example given by t. Formally:

Definition 2.12. *(PAC identification) A probabilistic student s* PAC *identifies a concept C from a probabilistic teacher t which draws examples according to an unknown distribution $\mathcal{D}$ if for all $0 < \epsilon < \frac{1}{2}$, $0 < \delta < \frac{1}{2}$, and every possible distribution $\mathcal{D}$, s outputs a hypothesis H such that with probability at least $(1 - \delta)$, the probability of error is less then ϵ: $P(t(c) \in H \triangle C) \leq \epsilon$.*

In PAC identification there is a high emphasis on the efficiency of learning. As one would expect, the efficiency of a PAC student depends on the complexity of the underlying target concept. For example, the complexity of learning a boolean formula depends on the number of literals in the target formula. But, as we will see in Section 2.2.3, it also depends on the used representation. In order to determine the time complexity of a PAC student, the following two values are used:

- A value k, which is an upper bound on the size of the minimal representation.
- A value n, which is the size (dimension) of the instance space.

A student is said to *PAC identify efficiently* when it runs in time polynomial in k, n, $\frac{1}{\epsilon}$, and $\frac{1}{\delta}$. Actually the algorithm does not necessarily need the value k. A technique called *doubling* (Goldman 1999) can be used to guess this value with sufficient probability. Algorithm 2.3 efficiently PAC identifies k-CNF formulas (Valiant 1984).

The claim that the above algorithm PAC identifies k-CNF formula is easily verified. Let p be the probability of obtaining an example from the teacher such that there is at least one inconsistent class in the current hypothesis. This probability decreases as the algorithm progresses. Now note that the algorithm can have two possible outcomes:

- At some point p becomes less than ϵ, in which case the identified hypothesis will be a good approximation of the target concept.
- The value of p never becomes less than ϵ, and hence the found hypothesis is not a good approximation. However, the probability of this happening is at most $1 - \delta$, since it corresponds with L Bernoulli trials, each of which has a probability of success greater than ϵ of obtaining fewer than $(2n)^{k+1}$ successes.

For a good overview of PAC identification the reader is referred to (Kearns and Vazirani 1994).

Algorithm 2.3 PAC identification of k-CNF

```
Require: A teacher $t$ of a target concept $C$. A value $L$, which is the smallest number of
  independent Bernoulli trials, each with a probability of success of at least $\epsilon$, and the
  probability of having fewer than $(2n)^{k+1}$ successes is less than $1 - \delta$
Ensure: $H$ is a PAC correct hypothesis
  Hypothesis $H := \emptyset$
  $f_h$ is the conjunction of all clauses over $n$ variables with at most $k$ literals
  for $i = 0; i < L; i = i + 1$ do
    Get the next example: $x = t(c)$
    for All clauses: $c \in f_h$ do
      if Clause $c$ is inconsistent with $x$ then
        Remove $c$ from $f_h$
      end if
    end for
    Set $H := \{x \in X \mid f_h(x) = \mathsf{true}\}$
  end for
  STATE Return $H$
```

2.2.3 Time complexity of learning problems

An interesting question in learning theory is how complex learning problems actually are, independent from the framework that is used to learn them. While at first sight it might not seem obvious, there is a close relation between 'normal' problems in computer science and learning problems. The key insight is that the problem of learning a specific hypothesis can be seen as a search problem. The search space consists of all possible hypotheses, and the goal of the search is to find a hypothesis for which the error is zero.

This relation can be used to prove hardness results for learning problems by reduction from known hard problems. Such a reduction maps every instance a of a hard problem A to a set S of labeled examples, which is then used to learn a concept class $\mathcal{C}$. The cardinality of S needs to be bounded by a polynomial in the length of a. The key property that is desired of this mapping is that $a \in A$ if and only if S is *consistent* with some concept $C \in \mathcal{C}$. Consistency is formally defined as follows:

Definition 2.13. *(consistency) A concept $C \in \mathcal{C}$ is* consistent *with a finite set of labeled examples $S = \{(x_1, b_1), (x_2, b_2), \ldots, (x_n, b_n)\}$ if for all $1 \leq i \leq n$, $b_i = true$ if $x_i \in C$, and $b_i = false$ otherwise.*

This notion of consistency allows us to reduce the problem of finding a solution for a known problem to the problem of learning a specific concept. We clarify this by giving a reduction from the NP-complete problem of graph 3-coloring to the problem of learning 3-term DNF formulas (Kearns and Vazirani 1994). The graph 3-coloring and 3-term DNF formulas learning problems are defined as follows:

Definition 2.14. *(graph 3-coloring) Given an undirected graph $G = (V, E)$, where V is a finite set of vertices and E a finite set of edges. Is there an assignment of a color to each vertex $v \in V$, such that at most 3 colors are used, and for each edge $\{v_1, v_2\} \in E$, vertex v_1 and vertex v_2 are assigned different colors?*

Definition 2.15. *(learning 3-term DNF formulas) Given a set of literals L, and a set labeled examples $S = \{(x_1, b_1), (x_2, b_2), \ldots, (x_n, b_n)\}$, where for all $0 \leq i \leq n$, x_i is an assignment to L and b_i is a value which indicates whether the assignment made by x_i should be true or false. Is there a DNF formula f over L, consisting of at most 3 terms (clauses), such that f is consistent with S?*

The basic idea of the reduction is that each term corresponds to a color, and each literal corresponds to a node. The constraints of the graph 3-coloring problem are imposed by the edges of the graph. These are represented by the negative examples in the learning problem. Given a graph 3-coloring instance, a learning 3-term DNF formulas instance is created in the following way:

- $L = V$
- For each $v \in V$, S contains a labeled example $(v(i), \mathit{true})$, where $v(i)$ is an assignment such that v is *false* and all other literals are *true*.
- For each $\{v_i, v_j\} \in E$, S contains a labeled example $(e(i,j), \mathit{false})$, where $e(i,j)$ is an assignment such that v_i and v_j are *false* and all other literals are *true*.

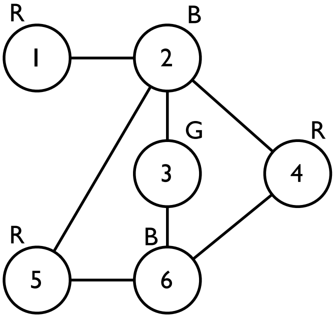
A term in the constructed learning 3-term DNF instance corresponds to a color. Each of the positively labeled examples represents a vertex. The idea of the reduction is that a pair of vertices can be given the same color if their corresponding examples are satisfied by the same term in a solution to the learning problem. This is illustrated in Figure 2.2. By using this correspondence it is not that hard to prove that the graph is 3-colorable if and only if the constructed sample is consistent with some 3-term formula.

The conclusion of this reduction is that the problem of learning 3-term DNF formulas is NP-hard. It is thus impossible to construct an efficient learning algorithm for learning 3-term DNF formulas, unless $\mathsf{P} = \mathsf{NP}$. Also, the problem is not efficiently PAC learnable, i.e., approximable, unless $\mathsf{RP} = \mathsf{NP}$.

The type of reduction we just showed is a general technique which can be used to prove (in)tractability results for learning problems. Note that in the previous section we discussed an algorithm that efficiently PAC learns k-CNF formulas. Therefore k-CNF formulas are efficiently PAC learnable. Also, every 3-term DNF formula can be represented by an equivalent 3-CNF formula using the fact that for a boolean algebra, $\vee$ distributes over $\wedge$:

$$T_1 \vee T_2 \vee T_3 \equiv \bigwedge_{u \in T_1, v \in T_2, w \in T_3} (u \vee v \vee w) \tag{2.1}$$

Using this equivalence it can be shown that 3-term DNF formulas are efficiently learnable *using* a 3-CNF formula as a hypothesis. This demonstrates an important principle in learning theory: even when the target concept class is fixed, the choice of hypothesis representation matters greatly for the efficiency of learning algorithms. The specific cause of intractability is worth noting: the problem of learning a prediction rule for 3-term DNF formula is tractable, but expressing the rule in a particular form is hard.



S_+	S_-
(011111, *true*)	(001111, *false*)
(101111, *true*)	(100111, *false*)
(110111, *true*)	(101011, *false*)
(111011, *true*)	(101101, *false*)
(111101, *true*)	(110110, *false*)
(111110, *true*)	(111010, *false*)
	(111100, *false*)

$T_R = x_2 \wedge x_3 \wedge x_6$

$T_B = x_1 \wedge x_3 \wedge x_4 \wedge x_5$

$T_G = x_1 \wedge x_2 \wedge x_4 \wedge x_5 \wedge x_6$

Figure 2.2: The construction used in the reduction from graph 3-coloring to learning 3-term DNF formulas. The examples consist of assignments to the literals $x_1, \ldots, x_6$ and an indication whether that assignment should be $true$ or $false$. Each literal x_i corresponds to vertex i. A learned 3-term DNF formula is shown below the examples. If a literal x_i does not occur in a term R in the learned 3-term DNF formulas, then vertex i should be given the color R in the coloring. Due to the construction of the examples, this coloring is guaranteed to be correct. Hence a correct 3-term DNF formula can be found if and only if the graph is 3-colorable.

Because of this difference in tractability there is a distinction between learning using the specified representation and learning using a different representation. When a student s uses a representation from a different hypothesis class $\mathcal{H}$, we say that s (efficiently) *identifies a concept using* $\mathcal{H}$. The only restriction an efficient student has on its hypothesis class is that it should be *efficiently evaluable*, i.e., the question whether $t(c) \in h$ can be evaluated efficiently (in polynomial time). The reason for this is that it seems useless to efficiently learn an inefficient hypothesis. Fortunately, there are many interesting hypotheses that can be evaluated in polynomial time.

2.2.4 Data complexity of learning problems

For learning problems, there exists an important measure of complexity in addition to time complexity: the number of examples required for convergence, also known as the *data complexity* or *sample complexity*. Intuitively, the data complexity measures the minimal amount of examples that is required to learn a sensible hypothesis. A hypothesis is *sensible* if it can be proven to be correct for at least some examples other than the ones given by the teacher. In other words, the student should have learnt at least something. We now formalize this notion.

The data complexity of a concept can be calculated by examining the amount of different labelings that its concept class is capable of achieving. A *labeling* S_l of a set of examples S is a set of all elements from S, paired with a boolean value, i.e., $S_l = \cup_{x \in S}(x, b)$, such that $b \in \{true, false\}$. A concept class $\mathcal{C}$ is capable of achieving a labeling S_l if a concept $C \in \mathcal{C}$ exists such that $x \in C$ if and only if

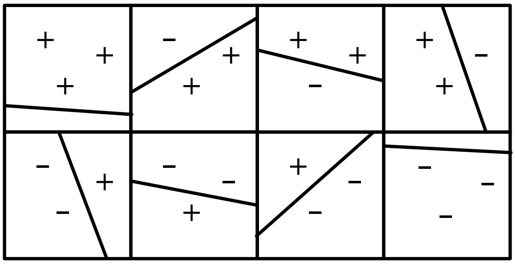


Figure 2.3: A set of 3 points in a plane is shattered by the concept class of a linear separator. There is no set of 4 points which is shattered by a linear separator, hence the VC-dimension of a linear separator is 3.

$(x, \mathsf{true}) \in S_l$.

When a concept class $\mathcal{C}$ achieves all possible labelings of a set of examples S, S is said to be *shattered* by $\mathcal{C}$, see Figure 2.3. Intuitively, when a set of examples of cardinality n is shattered by a concept class, a student that learns a concept from that class will need at least n examples to be able to learn anything. If a hypothesis H is learned from only $n - 1$ examples S', then for every possible other example $x \notin S'$, there exists a hypothesis H' such that H is not consistent with $S' \cup \{x\}$, and H' is consistent with $S' \cup \{x\}$. The cardinality of the largest set shattered by a concept class is a combinatorial concept known as the *Vapnik-Chervonenkis dimension* (Blumer, Ehrenfeucht, Haussler and Warmuth 1989). Formally:

Definition 2.16. *(Vapnik-Chervonenkis dimension) The* Vapnik-Chervonenkis (VC) dimension *of a concept class $\mathcal{C}$, denoted $VCD(\mathcal{C})$, is the cardinality of the largest set S shattered by $\mathcal{C}$. If arbitrarily large finite sets can be shattered by $\mathcal{C}$, then $VCD(\mathcal{C}) = \infty$.*

The VC dimension is especially important in PAC identification because it provides bounds on the amount of samples required to learn a certain concept. It has, for instance, been shown that any hypothesis $h \in \mathcal{H}$ consistent with a sample of size max $\left(\frac{4}{\epsilon}log(\frac{1}{\delta}), \frac{8}{\epsilon}VCD(\mathcal{H}) \cdot log(\frac{13}{\epsilon})\right)$ has error at most ϵ with probability at least $1 - \delta$. So a simple PAC identification algorithm consists of a polynomial time algorithm that outputs a hypothesis consistent with a sample of that size. However, this bound on the size of the sample is an overestimate. In practice a much smaller sample size will be sufficient to approximately learn a concept.

2.2.5 Learning from good examples

The data complexity of a concept discussed in the previous section is the minimum amount of examples that is required to learn a sensible hypothesis. However, the problem of finding a hypothesis that is consistent with these examples is often NP-hard. Such complexity results are based on the fact that the data can encode a hard decision problem. More specifically, it relies on the presence of a fixed input for the learning problem. While in normal decision problems this is very natural, in a learning problem the amount of input data is somewhat arbitrary: more data can be sampled if necessary. Therefore, it makes sense to study the behavior of a learning process when it is given more and more data. In practice, when the data size increases, the data will at some point no longer encode the hard problem because the data adds more and more constraints. At this point, the learning problem can be solved by a more efficient (polynomial time) algorithm.

Unfortunately, the amount of data that is required before such changes occur is difficult to formalize. The current method used in learning theory to deal with this shift in complexity is to determine the *minimum* amount of examples that is required for this shift to occur. The learning framework that studies this is known as *learning from good examples*, or *learning in helpful environments*. It is called good or helpful because the data that ensures convergence to the correct hypothesis is selected by a teacher that wants the student to learn a correct hypothesis, i.e., a helpful teacher. Therefore, in contrast to normal time complexity analysis, this complexity analysis is not a worst-case analysis, but a best-case analysis. In other words, the fact that a concept class is efficiently learnable from good examples does not guarantee that it will in fact be identified efficiently, only that it is possible. It can be the case that a malicious teacher only selects examples that continue to encode a hard problem.

We now discuss this notion of complexity for each of the three learning frameworks.

Identification in the limit In identification in the limit, the common complexity notion presented earlier is the amount of mind changes that is required a student to converge to a correct hypothesis. We now consider a helpful teacher that selects good examples for a student. The resulting complexity notion is the minimum number of examples that is required for the student to converge to a correct hypothesis. The notion is not the same as mind change complexity because a single mind change can require multiple examples, and moreover, because the examples that force a mind change do not have to be good examples.

In identification in the limit, if a polynomial amount of examples is sufficient for a student to converge to any correct hypothesis H from a concept class $\mathcal{C}$, then $\mathcal{C}$ is said to be *identifiable in the limit from polynomial data*. In addition, if the converging student requires run-time polynomial in the size of these examples, $\mathcal{C}$ is said to be *identifiable from polynomial time and data*, or simply *efficiently identifiable in the limit*. Efficient identifiability in the limit can be showed by proving the existence of polynomial *characteristic sets* (de la Higuera 1997).

Definition 2.17. *(characteristic set) A* characteristic set *S_{cs} of a target concept C for a learning algorithm A is an input sample $\{S_+ \in C, S_- \in C^c\}$ such that:*

- *given S_{cs} as input, algorithm A converges to a correct hypothesis H;*
- *given any input sample that contains S_{cs} as input, algorithm A still converges to H.*

Definition 2.18. *(efficient identification in the limit) A concept class $\mathcal{C}$ is* efficiently identifiable in the limit *if there exist two polynomials p and q and an algorithm A such that:*

- *given an input sample of size n, A runs in time bounded by $p(n)$;*
- *for every concept $C \in \mathcal{C}$, there exists a characteristic set S_{cs} of C for A of size bounded by $q(\|r_c\|)$, where r_c is the representation of C.*

We also say that the algorithm A identifies $\mathcal{C}$ efficiently in the limit. There exist many difficult learning problems that can be efficiently identified in the limit. For example, learning the smallest deterministic finite state automaton representation for a regular language is NP-complete (Gold 1978), but a deterministic finite state automaton is efficiently identifiable in the limit (Oncina and Garcia 1992). Again, the representation that is used for the concept is important for the complexity of the learning problem. For example, non-deterministic finite state automata can be used to represent regular languages, but they are not efficiently identifiable in the limit (de la Higuera 1997).

In the learning theory and grammatical inference literature there exist other definitions of efficient identification in the limit. It has for example been suggested to allow the examples in a characteristic set to be of length unbounded by the size of r_c (Yokomori 1995). The algorithm is then allowed to run in polynomial time in the sum of the lengths of these strings and the amount of examples in S. We use Definition 2.17 and 2.18 because these are the most restrictive form of efficient identification in the limit, and moreover, because allowing the lengths of strings to be exponential in the size of r_c results in an exponential (inefficient) identification procedure.

Query learning The usual notion of complexity in query learning is the amount of queries required to learn a correct hypothesis. This can be seen as a data complexity notion. However, a big difference is that the complexity in query learning is a worst case analysis. In other words, it assumes a malicious teacher instead of a helpful one.

Of course, the idea of a helpful teacher also exists in query learning. Because this views learning from the teachers perspective, this is referred to as *teachability* analysis. This type of analysis tries to answer the question of how many queries are necessary in the best case in order to teach any correct hypothesis. When this amount is guaranteed to be polynomial in the size of the representation of the target concept, then the class is said to be *efficiently teachable*. In order to avoid collusion, the student should also converge if more queries are asked, and other examples are returned by the teacher.

Interestingly, both efficient teachability and efficient query learnable imply efficient identifiability in the limit: the examples asked by the student and returned by the teacher can be used as a characteristic set for the query student (Goldman and Mathias 1996).

PAC identification The notion of good examples can also be used in PAC identification. Like identification in the limit, this complexity also measures the minimum amount of examples needed to learn a concept. However, unlike identification in the limit, PAC identification has to deal with a distribution over these examples. Therefore, we cannot select a specific set of examples to be used as a characteristic set. We can, however, select the type of distribution function. PAC identification becomes easier if this distribution function is one of the so-called *simple distributions* (Li and Vitányi 1991). These are all distributions that can be represented using the universal distribution. The concepts that are efficiently PAC learnable from simple distributions share close relations with the concepts that are efficiently identifiable in the limit, efficiently query learnable, and efficiently teachable (Parekh and Honavar 2000). For example, deterministic finite state automata are *simple-PAC learnable* (Parekh and Honavar 2001).

2.3 Discrete event systems

We are interested in identifying models for languages over event sequences. In this section, we give an overview of different kinds of such models. An advantage of these models is that they are insightful, as can be seen in the following example:

Example 2.2. When people try to explain the executions of a real-world system, they often describe sequences of events that change the state of the system. An example of such a description is an explanation of the execution of a combustion engine, taken from the website of HowStuffWorks (*http://www.howstuffworks.com*):

- The piston starts at the top, the intake valve opens, and the piston moves down to let the engine take in a cylinder-full of air and gasoline.
- Then the piston moves back up to compress this fuel/air mixture.
- When the piston reaches the top of its stroke, the spark plug emits a spark to ignite the gasoline. The gasoline charge in the cylinder explodes, driving the piston down.
- Once the piston hits the bottom of its stroke, the exhaust valve opens and the exhaust leaves the cylinder to go out the tail pipe.

In this description, the complete execution of the engine is a description of states, which are linked by the occurrences of events. Examples of these states are: the piston starts at the top, the piston moves down, the piston moves back up, the piston reaches the top, etc. Examples of event occurrences are: the intake valve opens, the gasoline explodes, the exhaust valve opens, etc. The combination of these discrete events and states constitutes an insightful model of the combustion engine.

A model that consists of a set of discrete states that are associated with a finite set of discrete events is known as a *discrete event system* (DES) (Cassandras and Lafortune 2008). A DES has the following properties:

- It is in a single state at each given moment in time.
- An event can occur instantaneously, which causes a transition from one state to the next (possibly the same) state.
- It is completely *event-driven*, which means that its state evolution depends entirely on the occurrence of discrete events.

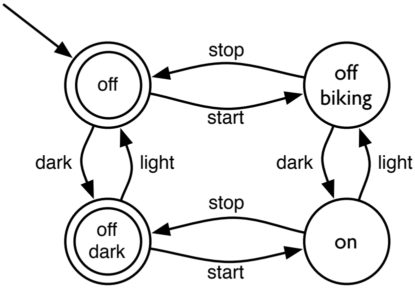


Figure 2.4: A deterministic finite state automaton that models a bike-light controller.

The study of DESs is mainly concerned with the sequence (ordering) of events that could happen in a given system, these sequences are called *strings*. The set of all possible strings is known as the *language* of a DES. There are several ways to represent a DES language. The most common DES model is the *finite state automaton* (from now on simply called automaton). An automaton is a directed graph consisting of a set of states (nodes) and transitions (directed arcs). The transitions are labeled with *events* (denoted by symbols). When an event occurs it activates the transition labeled with that event; this changes the current state of the DFA to the one the transition points to. We clarify these notions using a small example.

Example 2.3. The automaton in Figure 2.4 models a bike light controller. The bike light should be turned on when it is both dark, and someone is riding the bike. The states of the automaton are given names for convenience. Execution of the automaton starts in the state *off*, this is indicated by an arc that points to this state from nowhere. It accepts all strings that cause the bike to be back in its starting position, this is indicated by the double circle. The alphabet of the automaton consists of the following events: *start biking*, *stop biking*, *turns dark*, and *turns light*. When an event occurs that is the label of an outgoing arc, the current state of the automaton changes to the state the arc points to. This is an example of an accepted string: *start stop dark start stop light*.

The type of languages that are represented as automata are known as the *regular languages*. Other types of languages require other (more expressive) models. For example the *context-free languages* can be represented as *push-down automata*. These languages are also an interesting topic, but they are notoriously difficult to learn. Because we are interested in language learning, we therefore only consider models for regular languages.

In this section we will describe several automaton models that are commonly used for modeling a DES. These models can be grouped into three distinct classes of machines, depending on the type of information that is associated with an event. In addition to information on the sequence of events, we sometimes have timing and statistical information. Timing information is available in the form of points in time at which the events in a sequence occur. Adding this additional timing information results in strings of event-time value pairs, sets of these strings form a *timed language*. This kind of language can be modeled by a *timed automaton*. In this model time delay between two successive events is sometimes called the event *lifetime*.

The statistical information can be used in two ways. The first way is to model the lifetime of an event using a probability distribution function. The second is to just add probability functions to state transitions of an automaton model. Adding both these probability functions to the events results in a *probabilistic timed language*, which is modeled by a *probabilistic timed automaton*. People often use the word stochastic instead of probabilistic. In the study of DESs this is known as the *three levels of abstraction*:

- *Non-timed* (languages): sequences of events.
- *Timed* (timed languages): sequences of event-time value pairs.
- *Probabilistic* (probabilistic timed languages): sequences of event-time value pairs with probability functions for the event lifetimes and the state transitions.

The choice of the appropriate level of abstraction depends on the application. In most cases a non-timed model will be sufficient. It is our opinion, however, that when one has relevant timing or statistical information available, one should use it in the model since this better reflects reality.

In each of the following sections we discuss one specific type of automaton. We discuss the non-timed models in Section 2.3.1. Then we explain the timed models in Section 2.3.2, and we end with probabilistic models in Section 2.3.3.

2.3.1 Non-timed Automata

The non-timed automaton model of a DES is based on the theory of formal languages and automata (see any textbook on formal languages, e.g., (Sudkamp 2006)). In this theory an automaton is an abstract machine that has only a finite, constant amount of memory. The internal states of the machine carry no further structure. An automaton consists of *states* and *transitions*; it can be described as a directed graph such as the one in Figure 2.4. An automaton operates on a finite set of symbols, known as an *alphabet*. The alphabet of an automaton is denoted by Σ; the elements of an alphabet are known as *symbols*, in a DES they are sometimes called events. Each transition is associated with a symbol called the transition *label*.

Automata computation is a process that executes *state transitions*, while doing this it maintains a state known as the *current state* of the automaton. A state transition *activates* of *fires* one of the outgoing transitions from the current state, this changes the current state to the state that the activated transition points to. An automaton computation can be described by a (not necessarily finite) sequence of states and transitions:

$$state_1 \xrightarrow{transition_1} state_2 \xrightarrow{transition_2} \ldots$$

In this sequence the first state is a fixed state known as the *start state*, and each transition is an activated transition that changes the current state (just before it in the sequence) to the next state (right after it in the sequence). The labels of the transitions in a computation sequence form a *string* (a sequence of symbols). Depending on the type of automaton, an automaton either generates a string, accepts a string, or both. There are three types of automata:

- *Generators*: a computation generates an output string.
- *Acceptors*: a computation accepts an input string.
- *Transducers*: a computation generates an output string from an input string.

A DES is usually seen as a generator of event strings. The set of strings that an automaton $\mathcal{A}$ accepts or generates is called the *language* $L(\mathcal{A})$ of the automaton. This set is determined by the set of valid computations of the automaton. A finite computation is *valid*, sometimes called *marked*, when it ends in one of the *final states*. Final states are a predetermined subset of the states of an automaton. The language of an automaton consists of the strings formed by the labels of transitions in valid computations of an automaton, each of these is called a *valid string*. The set of all possible strings is denoted by Σ^*.

There is an important distinction between a *deterministic finite automaton* (DFA) and a *non-deterministic finite automaton* (NFA). The difference is that in the deterministic case there exists at most one transition that is capable of activation with the same label. This implies that when an input or output string is given, the computation that accepts or generates this string is easy to determine. In a non-deterministic automaton, there can be more than one such transition. Thus given an input or output string there is a choice of possible computations. This difference does not influence which languages can be generated or accepted by these automata, it only influence the representation of these languages.

Another distinction has to do with the length of a string, which is usually assumed to be finite. It is however also possible to create automata that accept or generate infinite strings. A set of these strings is known as an *ω-language*. The most common type of automata that operate on these strings are the Büchi automata, which are explained in the end of this section.

Deterministic automata

The DFA is the basic, and most commonly used, automaton. All the other automata models we describe in this section use the same structure as a DFA. A DFA is formally defined as follows:

Definition 2.19. *(DFA) A* deterministic finite state automaton $\mathcal{A}$ *is a 5-tuple* $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ *where* Q *is a finite set of states,* Σ *is a finite set of symbols,* δ *is a partial mapping of* $\Sigma \times Q$ *into* Q*,* $q_0 \in Q$ *is the start state, and* $F \subseteq Q$ *is a set of final states.*

The mapping δ defines the transitions of the automaton, and is therefore known as the *transition function*. The automaton uses this function to generate or accept a string. How this is done by the automaton is defined in the definition of *computation* in a DFA.

Definition 2.20. *(DFA computation) A (finite)* computation of a DFA $(Q, \Sigma, \delta, q_0, F)$ *over a string* $a_1 a_2 \dots a_n$ *is a sequence*

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \dots \xrightarrow{a_n} q_n$$

of states and transitions, such that $\delta(a_i, q_{i-1}) = q_i$*, for all* $1 \leq i \leq n$*,* $q_i \in Q$*, and* $a_i \in \Sigma$*. A finite computation of a DFA is* valid *when* $q_n \in F$*.*

The transition function δ can be extended from symbols to strings. This *extended transition function* is a mapping of $\Sigma^* \times Q$ into Q and is denoted by δ^*. The state returned by δ^* is defined by multiple applications of δ: $\delta^*(a_1 \dots a_n, q) = \delta(a_n, \delta^*(a_1 \dots a_{n-1}, q))$, where $a_1 \dots a_n \in \Sigma^*$ and $q \in Q$. When the input to δ^* is just one symbol instead of a string, the result is identical to that of δ.

Sometimes a DFA contains a special function Γ, called the *active event function.* This function maps each state to the set of events that are capable of activation in that state. These events are called the *feasible* events. An event that is not feasible is called *infeasible*. Feasible events are used by the transition function of the DFA: when the transition function is applied to an infeasible event the result is undefined. This makes the transition function a *partial* function on its domain, and Γ is used to determine where it is defined. Because of this there is a distinction between two different languages of a DFA that contains a Γ function:

- The *language generated* by a DFA $\mathcal{A}$ is the set of strings for which a computation is feasible (using δ).
- The *language recognized* by a DFA $\mathcal{A}$ is the set of valid strings, i.e., those strings that end in a final state. This is a subset of the language generated by $\mathcal{A}$.

Each of the strings in the language generated by a DFA is called a *generated string.* If δ is not a partial mapping, all possible strings (Σ^*) are generated strings. The automaton in Figure 2.4 is an example of a DFA.

Non-deterministic automata

The difference between a deterministic and a non-deterministic model is that in a non-deterministic model the exact state of the system is not necessarily determined. This means that instead of talking about the state of a system we talk about a *set of possible states.* There are two situations in which a DES will need a non-deterministic model. The first is when its state can change into two (or more) possible states when an event occurs. The second is when the state of a DES can change without the occurrence of an event. However, because a DES is event-driven, we say that an event actually does occur, but that we cannot observe this occurrence. These events are called *unobservable events*, which are modeled by transitions that have the empty string as label, called ϵ-transitions. In order to model these two situations we need to make the following changes to a DFA:

- In each transition the next state is changed into a set of possible next states: Q becomes 2^Q in the range of δ.

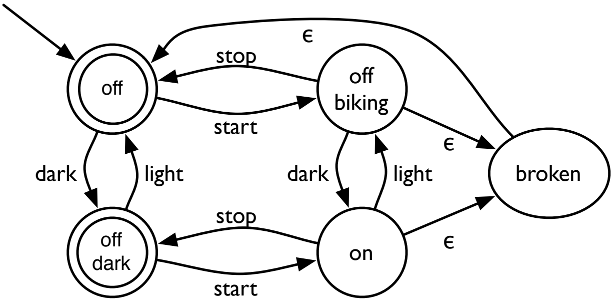


Figure 2.5: A non-deterministic finite state automaton that models a bike light controller which can get broken and fixed.

- Unobservable events are added to the alphabet: Σ becomes $\Sigma \cup \{\epsilon\}$.

We extend the DFA example to a non-deterministic finite state automata (NFA) by including unobservable events.

Example 2.4. The automaton in Figure 2.5 is a bike light controller which can get broken and fixed, due to some unobservable event ϵ. The difference with the automaton from Figure 2.4 is that this automaton recognizes strings such as: *start dark start stop*. Obviously, when such a string occurs the controller must have been broken. In the automaton in Figure 2.4 it is impossible that two *start* events occur without a *stop* event between them.

Formally an NFA is:

Definition 2.21. *(NFA) A* non-deterministic finite state automaton *$\mathcal{A}$ is a 5-tuple $\mathcal{A} = (Q, \Sigma \cup \{\epsilon\}, \delta, Q_0, F)$ where Q is a finite set of states, Σ is a finite set of symbols, ϵ is the empty symbol, δ is a partial mapping of $\Sigma \cup \{\epsilon\} \times Q$ into 2^Q, $Q_0 \subseteq Q$ is a set of possible start states, $F \subseteq Q$ is a set of final states.*

Note that the starting location is also modeled by a set of possible states. A finite computation of an NFA is basically the same as the computation of a DFA. The main difference is that in an NFA there is a set of possible computations given an input string, instead of just one. Therefore the notion of a valid string needs to be modified. We say that a string s is *valid* if at least one of the possible computations given an input string s is a valid computation. Formally the computation of a non-deterministic event automaton is:

Definition 2.22. *(NFA computation) A (finite)* computation of an NFA $(Q, \Sigma \cup \{\epsilon\}, \delta, Q_0, F)$ *over a string $a_1a_2 \ldots a_n$ is a finite sequence*

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \cdots \xrightarrow{a_n} q_n$$

of states and transitions, such that $q_0 \in Q_0$, and $q_i \in \delta^*(\epsilon^* a_i, q_{i-1})$ for all $1 \leq i \leq n$, $q_i, q_i' \in Q$, and $a_i \in \Sigma$. A finite computation of an NFA is valid *when $q_n \in F$.*

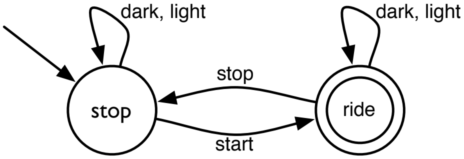


Figure 2.6: A Büchi automaton that models infinite sequences that do not neglect the bike.

Like the language of a DFA, the language $L(\mathcal{A})$ of an NFA $\mathcal{A}$ is the set of valid strings. Like the DFA, an NFA can also be equipped with a Γ function. The computation of an NFA looks fundamentally different from the computation of a DFA. However, given an NFA $\mathcal{A}$, a simple procedure exists that constructs a DFA $\mathcal{A}'$ that accepts the same language as $\mathcal{A}$ (see e.g. (Sudkamp 2006)). This DFA can be of size exponential in the size of the NFA.

Büchi automata

In the two automata models that we discussed above, a computation is guaranteed to terminate. This means that the real-world system that it models will operate until the point in time where it is finished. But sometimes there is no such point in the system that we want to model. Such a system operates *continuously*, producing *infinite* strings. There exist for example reactive systems, such as an automatic pilot, that need to interact continuously with their environment. An automaton that is commonly used to generate or accept infinite strings is the *Büchi automaton* (Thomas 1991). The language of infinite strings that such an automaton generates or accepts is known as an *ω-language*. A Büchi automaton is identical in structure to a normal finite automaton (deterministic or non-deterministic). The difference between the two lies in the computation, which is infinite for Büchi automata.

An infinite computation, sometimes called a *run*, will never terminate. Because of this we require a different notion of a valid computation. We say that an infinite computation C is valid if and only if there exists a final state that appears infinitely often in C. An infinite string s is valid if there exists a valid infinite computation of s. These are the strings accepted by a *Büchi automaton*, as shown in the next example.

Example 2.5. Figure 2.6 shows a Büchi automaton. The automaton recognizes all infinite strings that do not neglect the bike: Every time when a *stop* event occurs, a *start* event must eventually occur, otherwise the automaton rejects the string.

Here we give the formal definition of *infinite computation* of a Büchi automaton.

Definition 2.23. *(Büchi computation) An (infinite)* computation of a Büchi automaton $(Q, \Sigma, \delta, Q_0, F)$ *over an infinite string* $a_1 a_2 \ldots$ *is an infinite sequence*

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \cdots$$

of states and transitions, such that $q_0 \in Q_0$, and $q_i \in \delta(e_i, q_{i-1})$ for all $i \geq 1$, $q_i \in Q$, and $e_i \in \Sigma$. An infinite computation c is called valid *if and only if there exists a $q_n \in F$ such that q_n occurs infinitely often in c.*

The condition of infinite occurrence of a final state is known as the *Büchi acceptance condition*. Of course there are other acceptance conditions possible (see (Thomas 1991)); the Büchi condition is the most common one.

2.3.2 Timed Automata

The automata models described in the previous section are powerful models for describing real-world event systems. A problem, however, is that they fail to model an important part of many real-world event systems, namely the timed relations between events. Timed relations are modeled by adding a time-stamp to each event occurrence. Sequences of these timestamped events are called *timed strings*.

Definition 2.24. *(timed string) A* timed string *τ over an alphabet Σ is a (possibly finite) sequence of pairs $(a_1, t_1)(a_2, t_2)\ldots$, where $a_i \in \Sigma$ is an event and $t_i \in \mathbb{R}_+$ is a time value. An* untimed string *corresponding to a timed string $(a_1, t_1)(a_2, t_2)\ldots$ is the string $a_1 a_2 \ldots$, obtained by removing the time values.*

The time values of a timed string represent the lifetime of the events. That is, time value t_i denotes the time delay between the occurrences of event a_{i-1} and a_i. Sometimes, the exact times of occurrence are used as time values. These two representations are equivalent. A *timed language* is a set of timed strings over an alphabet, and an *untimed language* consists of all untimed strings of a timed language. In a DES, a timed string is known as a *timed event sequence*. An automaton that accepts or generates a timed language is called a *timed automaton* (TA) (Alur and Dill 1994). A TA is an automaton that is constrained by timing requirements so that it can operate on timed strings. The timing requirements can be added to automata in different ways. We will discuss two approaches for this: clock guards and clock structures.

The additional timing information needed by the timing requirements is usually maintained in an automaton in the form of *clocks*. A *clock* $x \in \mathbb{R}_+$ is a real-valued variable. How this value changes depends on the type of clock, which can be either increasing or decreasing. We will explain both types by their usage in a DES model. The first model we discuss is a timed acceptor that uses clock guards and increasing clocks. The second model is a timed generator that uses clock structures and decreasing clocks. Other combinations are also possible, for example an automaton that uses clock guards and both increasing and decreasing clocks for events is called an *Event Clock Automaton*. Also, timed Büchi automata exist. These accept or generate infinite timed strings.

Guarded timed automata

The first timed DES model we discuss is known as the *event recording automaton* (ERA) (Alur, Fix and Henzinger 1999). An ERA is an acceptor in which each event is associated with an increasing clock. Such a clock records the time since the last time an event occurred and is therefore called an *event recording clock*.

Definition 2.25. *(event recording clock) An* event recording clock *is an object with the following properties:*

- *it increases over time, synchronously with all other clocks;*
- *it is reset to* 0 *every time the associated event occurs; after a clock has been reset it will immediately start increasing again;*
- *it can be* valuated*, which means that there is a function that maps the clock to its value.*

When a transition in an ERA is activated, it resets the value of the clock associated with the event that activated it. A *clock valuation* v maps each clock x to a value $v(x) \in \mathbb{R}_+$. A clock records the time elapsed since its last reset. Timing restrictions are included in an ERA in the form of guards.

Definition 2.26. *(clock guard) A* clock guard *(or constraint) is an arithmetic constraint on clocks defined inductively as*

$$\begin{array}{rcl} g & := & x \leq n \\ & | & n \leq x \\ & | & x \leq y \\ & | & g_1 \vee g_2 \\ & | & g_1 \wedge g_2 \end{array}$$

where x and y are clocks, $n \in \mathbb{Q}$, and g_1 and g_2 are clock guards. A clock guard is satisfied *by a clock valuation when the guard evaluates to* true *given the clock values.*

Usually, clock guards are defined without the so-called *inter-clock constraint* $x \leq y$. We include it here for completeness. Also, sometimes a distinction is made between *closed clock guards* ($x \leq n$) and *open clock guards* ($x > n$). The inclusion of guards in an automaton makes the transition mapping difficult to define. It is more convenient to define the timed automaton using transition tuples. Here we give the definition of an event recording automaton.

Definition 2.27. *(ERA) An* event recording automaton *is a 6-tuple $\mathcal{A} = (Q, X, \Sigma, \Delta, q_0, F)$, where Q is a finite set of states, X is a finite set of clocks of size $|\Sigma|$ (one for each event), Σ is a finite set of symbols, Δ is a finite set of transitions, q_0 is the start state, and $F \subseteq Q$ is a set of final states. A transition $\delta \in \Delta$ is a tuple $\langle q, q', a, g \rangle$, where $q, q' \in Q$ are the source and target states respectively, $a \in \Sigma$ is the occurring event, and g is a clock guard.*

A transition $\langle q, q', a, g \rangle$ is interpreted as follows: whenever the machine is in state q, the next symbol is a, and the clock guard g is satisfied, then the machine can move to state q', resetting the clock associated with a. An ERA is called deterministic if no two transitions in Δ exist such that the source states are identical and the clock guards can both be satisfied by a single clock valuation. If two such transitions exist the automaton is non-deterministic, i.e., at some point there are at least two possible next states. Also ϵ-transitions can be added to an ERA to make it non-deterministic. The computation of an ERA is defined as follows:

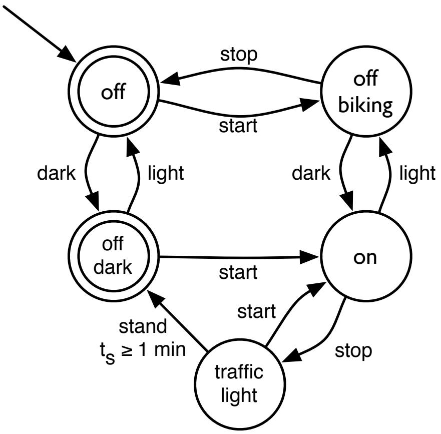


Figure 2.7: An event recording automaton that models a 'smart' bike light controller.

Definition 2.28. *(ERA computation) A* computation of an ERA $\mathcal{A}$ *over a timed string* $\tau = (a_1, t_1) \ldots (a_n, t_n)$ *is a finite sequence of states and transitions*

$$q_0 \xrightarrow{(a_1,t_1)} q_1 \xrightarrow{(a_2,t_2)} q_2 \ldots q_{n-1} \xrightarrow{(a_n,t_n)} q_n$$

such that there exists a transition $\langle q_{i-1}, q_i, a_i, g\rangle \in \Delta$*, such that* g *is satisfied by the valuation* v_i *for all* $0 \leq i \leq n$*,* $q_i \in Q$*, and* $a_i \in \Sigma$*. The valuation* v_i *is defined inductively as:* $v_i(x) = 0$ *if* a_i *is associated with* x*,* $v_i(x) = v_{i-1}(x) + t_i$ *otherwise, and* $v_0(x) = 0$*, for all* $x \in X$*. A finite computation of an ERA is called* valid *when* $q_n \in F$*.*

Example 2.6. The automaton in Figure 2.7 models a 'smart' bike light controller. The idea is that when someone stops with his or her bike it can have several causes, one being a traffic light or stop sign. In this case the light should not be turned off, since the other road users would otherwise not be able to see the bike. It is known that such a stop cannot last longer than one minute. This is why, when a *stop* event occurs, one additional event is read continuously, the *stand* event. This lasts until a *start* event occurs again, or until the bike has been stopped for at least a minute.

The timed language $L(\mathcal{A})$ recognized by an ERA $\mathcal{A}$ is the set of timed strings for which there exists a valid computation. The ERA is a simplified version of the timed automaton model by Alur and Dill (Alur and Dill 1994). In the original TA there is no restriction on the amount of clocks, and each transition is capable of resetting any multitude of clocks. These TAs are defined as follows.

Definition 2.29. *(TA) A* timed automaton *is a 6-tuple* $\mathcal{A} = \langle Q, X, \Sigma, \Delta, q_0, F\rangle$*, where* Q *is a finite set of states,* X *is a finite set of clocks,* Σ *is a finite set of symbols,* Δ *is a finite set of transitions,* q_0 *is the start state, and* $F \subseteq Q$ *is a set of final states.*

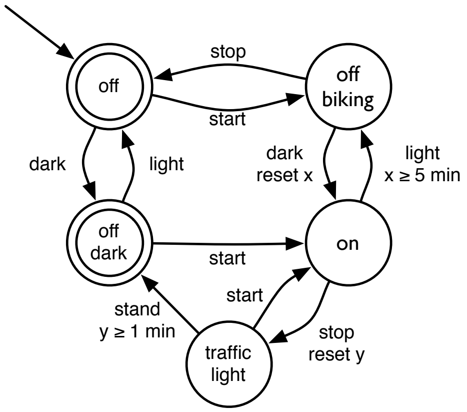


Figure 2.8: A timed automaton version of the 'smart' bike light controller.

A transition $\delta \in \Delta$ is a tuple $\langle q, q', a, g, R\rangle$, where $q, q' \in Q$ are the source and target states, $a \in \Sigma$ is a symbol called the transition label, g is a clock guard, and $R \subseteq X$ is the set of clock resets.

Example 2.7. Figure 2.8 shows a TA version of the 'smart' bike light controller from Figure 2.7. In addition, it contains a clock guard on the occurrence of a light event while driving the bike. The idea is that when driving on a lantern lit road, the light will not turn off when the bike is under a lantern. Note that only the *dark* event that occurs while driving resets the clock x that is used in the clock guard of the transition with the *light* event.

Every transition δ in a TA is associated with a set of clocks R. When a transition δ occurs (or fires), the values of all the clocks in R are set to 0, i.e. $\forall x \in R, v(x) := 0$. The values of all other clocks remain the same. We say that *δ resets x* if $x \in R$. In this way, clocks are used to record the time since the occurrence of a specific event. Clock guards are then used in the normal way to change the behavior of the TA depending on the value of clocks:

Definition 2.30. *(TA computation) A* computation of a TA $\mathcal{A}$ *over a timed string $\tau = (a_1, t_1) \dots (a_n, t_n)$ is a finite sequence of states and transitions*

$$q_0 \xrightarrow{(a_1,t_1)} q_1 \xrightarrow{(a_2,t_2)} q_2 \cdots q_{n-1} \xrightarrow{(a_n,t_n)} q_n$$

such that there exists a transition $\langle q_{i-1}, q_i, a_i, g, R_i\rangle \in \Delta$, such that g is satisfied by the valuation v_i for all $0 \leq i \leq n$, $q_i \in Q$, and $a_i \in \Sigma$. The valuation v_i is defined inductively as: $v_i(x) = 0$ if $x \in R_i$, $v_i(x) = v_{i-1}(x) + t_i$ otherwise, and $v_0(x) = 0$, for all $x \in X$. A finite computation of a TA is called valid *when $q_n \in F$.*

The timed language $L(\mathcal{A})$ recognized by a TA $\mathcal{A}$ is the set of timed strings for which a valid computation exists. Although TAs are very powerful models, any TA $\mathcal{A}$ can be transformed into an equivalent non-timed automata, recognizing an non-timed language that is equivalent to $L(\mathcal{A})$. This is done using the so-called *region construction method.* We briefly explain this method and its consequences.

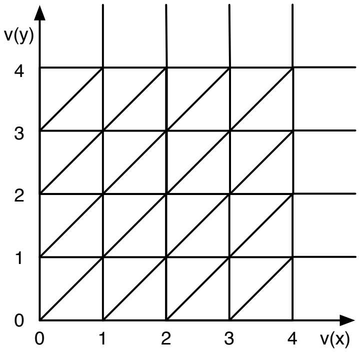


Figure 2.9: The regions that are defined by the construction method. The clock regions are: every line that does not intersect with other lines (56 regions), every line intersection (25 regions), and every space bordered by these lines and intersections (41 regions). In total there are 122 clock regions for two clocks x and y that have an upper bound of 4.

Region Construction The region construction method (Alur and Dill 1994) divides the space defined by the real-time clocks into small equivalence classes, known as regions. For the construction we first assume that every constant used in a clock constraint is an integer (if there are rational constants, all constants need to be multiplied by the greatest common denominator). The construction then goes as follows. For any $t \in \mathbb{Q}$, let $\langle t \rangle$ denote the fractional part, and let $\lfloor t \rfloor$ denote the integer part. For each clock $x \in X$, let n_x be the largest integer such that x is compared with n_x in some clock guard. We say that two clock valuations v and w are *region equivalent* if and only if all of the following conditions hold:

- For all clocks $x \in X$, either $\lfloor v(x) \rfloor$ and $\lfloor w(x) \rfloor$ are the same, or both $v(x)$ and $w(x)$ exceed n_x.
- For all clocks $x, y \in X$ with $v(x) \leq n_x$ and $v(y) \leq n_y$, $\langle v(x) \rangle \leq \langle v(y) \rangle$ if and only if $\langle w(x) \rangle \leq \langle w(x) \rangle$.
- For all clocks $x \in X$ with $v(x) \leq n_x$, $\langle v(x) \rangle = 0$ if and only if $\langle w(x) \rangle = 0$.

A *clock region* is an equivalence class of region equivalent clock valuations. Figure 2.9 shows an intuitive picture of the clock regions for two clocks. The idea of a clock region is that it contains all clock valuations for which the clock guards must have the same behavior. More specifically, let the runs of two timed strings τ and τ' be such that they end in the same state q, and such that their last valuations v_n and v'_n are within the same region. Now it holds that the TA accepts $\tau\tau''$ if and only if the TA accepts $\tau'\tau''$. Thus, for the acceptance of timed strings it only matters which region is reached, not the exact valuation within a region. The number of regions of a TA is finite but exponential, the number is at most $k! \times 4^k \cdot \prod_{x \in X}(n_x + 1)$. The regions are used to construct a new automaton, known as the *region automaton*. This is a DFA that contains a unique state for every combination of a state and a region of the TA. The transitions between these states (regions) are equivalent to the normal transitions of the TA plus transitions for time increases between regions. Because all valuations within one region have the same behavior, this DFA is behaviorally equivalent to the original TA.

The problem with region automata is that the explosion of states is exponential in the number of clocks and constants in clock guards. A more efficient (but still exponential) representation combines regions symbolically (based on identical behavior) into so-called *zones*. A zone is the maximal set of clock valuations satisfying a clock constraint. These sets can be efficiently represented and stored in *difference bound matrices* (DBMs) (Alur 1999). A lot of timed automata based model checking methods (tools) are based on DBMs.

Structured timed automata

A different timed DES model that we call the *structured timed generator* (STG) is mentioned in (Cassandras and Lafortune 2008). This model is a generator in which each event is associated with a decreasing clock. Such a decreasing clock associated with an event e is an indicator for the time until the next occurrence of e. Because of this we call these clocks *event predicting clocks*. In an STG there is a difference between an event that is active and an event that actually occurs. An event is *active* when it is feasible in the current state of the automaton. An event actually *occurs* when its clock runs down to zero. Because of this difference an STG makes use of a Γ-function. The properties of an event predicting clock are given in the following definition:

Definition 2.31. *(event predicting clock) An* event predicting clock *is an object with the following properties:*

- *it decreases over time, synchronously with all other clocks;*
- *it can be reset to any positive real valued number $n \in \mathbb{R}_+$, after a clock has been reset it will immediately start decreasing;*
- *it can never go below the value* 0*, when it reaches* 0 *the clock will stop;*
- *it can be valuated.*

The occurrence of an event in an STG forces a state transition. In addition to this state transition some events are activated and deactivated, determined by the new state of the STG. When an event becomes *activated* in an STG, its clock is reset to the event lifetime. An event e will usually remain active for the duration of its lifetime, but sometimes it is *deactivated* by another event occurring which makes e infeasible. When an event occurs it is reactivated if it is feasible in the new automaton state. The way in which an STG determines the next occurring event is summarized by the following rules:

- To determine the next event, compare clock values of all events feasible at the current state and select the smallest one. Let this be event e.

```
clock structure:

start:  3.0 ;  3.5 ;  0.5 ;  4.5 ; ...
stop:  0.25 ;  0.1 ;  0.1 ; 0.25 ; ...
dark:  12.0 ; 12.0 ; 12.0 ; 12.0 ; ...
light: 12.0 ; 12.0 ; 12.0 ; 12.0 ; ...
```

Figure 2.10: A clock structure for the lifetimes of events for a day at work for a bike rider.

- Event e is *activated* when:
  - e has just occurred and it remains feasible in the new state, or
  - a different event has just occurred while e was not feasible, causing a transition to a new state where e is feasible.
- Event e is *deactivated* when a different event occurs causing a transition to a new state where e is not feasible.

What the new lifetime value of an event should be when it is activated is determined by a fixed sequence of time values. These time values are maintained in a set called a *clock structure*. Such a clock structure is given as input for the computation of an STG.

Definition 2.32. *(clock structure) A* clock structure *associated with a set of symbols Σ is an ordered finite set $C = (c_e : e \in \Sigma)$ of (possibly infinite) time value (lifetime) sequences $c_e = t_{e,1}, t_{e,2}, \ldots$.*

Example 2.8. Figure 2.10 shows a clock structure for the lifetimes of events for a day at work for a bike rider. The day starts at 5:30 in the morning. The bike is used to: ride to work, ride to lunch, ride back from lunch, and ride back from work. In the ride back home it will get dark and the bike light should be turned on. When the structure of the automaton in Figure 2.4 is followed, the constructed timed string is: $(\mathit{start}, 3.0)(\mathit{stop}, 3.25)(\mathit{start}, 6.75)(\mathit{stop}, 6.85)(\mathit{start}, 7.35)$ $(\mathit{stop}, 7.45)(\mathit{start}, 11.95)(\mathit{dark}, 12.00)(\mathit{stop}, 12.20)\ldots$.

In a *computation of an STG* the decreasing clock of each feasible event e is set to $t_{e,1}$. In order to determine which value in this sequence should next be given to a clock, a value called the *clock score* is maintained for each event.

Definition 2.33. *(clock score) The* clock score *of an event e at time t is the number of times that e has been activated in the interval $[t_0, t]$, where t_0 is the starting time of the computation.*

This value keeps track of how often the event has been activated and is a pointer to a time value of the lifetime sequence. For example when an event e has already been activated twice and is again activated by an event occurrence, the clock of e is reset to the the third value of the lifetime sequence of e. By using clock scores, an STG can be used to generate a timed string from a clock structure.

2.3.3 Probabilistic Automata

In a real-world applications of DESs one often has to deal with the problem of uncertainty. In these cases, the exact state of a DES is often unknown. Usually, however, the probability of each possible state is known. These probabilities can for example be derived from sensor data. The most common way to add these probabilities to an automaton model, and the one which we will describe in this section, is to add probability values to the state transitions. This results in an automaton that largely resembles an NFA (see Section 2.3.1), known as a *probabilistic automaton* (PA). Formally a probabilistic automaton is defined as follows:

Definition 2.34. *(PA) A* probabilistic automaton *$\mathcal{A}$ is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is a finite set of events, δ is a mapping from $Q \times Q \times \Sigma$ into $\mathbb{R}$ (the* transition probability*), $q_0 \in Q$ is the start state, $F \subseteq Q$ is a set of final states.*

Note that the model in this definition does not contain ϵ-transitions. These can be added but they only complicate the computation of probabilities. The transition function δ can be used to define a notion of computation that assigns probabilities to strings. Sometimes a PA contains a *probability mass function over start states*, instead of a fixed start state. This probability function contains for each state the probability of starting a computation in that state. Also, states sometimes have a *final probability*. This is the probability that a computation ends in that state, instead of taking a transition to another state.

It is possible to define a notion of computation of a PA that accepts or generates pairs of strings and probabilities. However, we believe it to be more intuitive to think of a computation of a PA as a process that assigns probabilities to strings. Therefore our notion of computation of a PA takes a string as its input and produces a probability value as output.

Definition 2.35. *(PA computation) A* computation of a PA *$\mathcal{A}$ over a string $s = e_1 \ldots e_n$ is a pair consisting of a finite sequence of states and transitions*

$$q_0 \xrightarrow{e_0} q_1 \xrightarrow{e_1} q_2 \cdots q_{n-1} \xrightarrow{e_{n-1}} q_n$$

and a probability value $p = \prod_{0 \leq i \leq n} \delta(q_i, q_{i+1}, e_i)$, where $q_i \in Q$ and $e_i \in \Sigma$ for all $0 \leq i \leq n$. A finite computation of a PA is called valid *when $q_n \in F$ and $p > 0$.*

This definition of a computation requires a special notion of a valid string-value pair (which we call a *probabilistic string*) in a PA. A probabilistic string (s, p) is valid in a PA $\mathcal{A}$ when the sum of all probabilities of valid computations of $\mathcal{A}$ over s equals p. A PA $\mathcal{A}$ is called deterministic (DPA) if for any string s there exists at most one computation of $\mathcal{A}$ over s. The language of a PA $\mathcal{A}$ is the set of all valid probabilistic strings in $\mathcal{A}$.

Example 2.9. Figure 2.11 shows a PA that models the bike light controller that can be broken from Figure 2.5. In the PA model, it has a probability of being broken and fixed. The other transitions also have probabilities, for each state the sum of the probabilities of all outgoing transitions equals 1. The final states have final probabilities of 1 minus the sum of all outgoing probabilities. For example the probability that, once broken, the bike/controller stays broken is $1 - (0.7 + 0.1) = 0.2$.

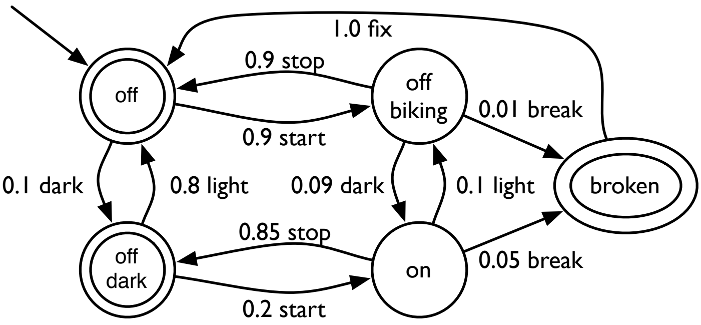


Figure 2.11: A probabilistic automaton that models a bike light controller that has a probability of being broken and fixed.

During a computation, a PA computes a probability value p. When designing a probabilistic automaton one of the main questions is what kind of probability is needed, i.e. which values need to sum to one. In practice there are three common methods for doing this:

1. In each state, given an event e, the sum of all probabilities of outgoing transitions labeled with e sum to one: $\sum_{q_n \in Q} \delta(q_p, q_n, e) = 1$, for all $q_p \in Q$ and $e \in \Sigma$. This type of PA models the probability of choosing a specific computation path given an input string in an NFA. This models the probability p_q that the PA currently is in state q, the sum of these probabilities over all states sum to one. In other words, it models a *probability distribution over states.*

2. In each state the probabilities of all outgoing transitions over all events e sum to one: $\sum_{q_n \in Q} \sum_{e \in E} \delta(q_p, q_n, e) = 1$, for all $q_p \in Q$. The probability *assigned to* a string τ is the product of the probabilities of the transitions it fires during its computation. The probabilities of different possible computations are summed up in the case of non-determinism. These probabilities model *distributions over strings of the same length* (sometimes denoted Σ^n).

3. In each state the probabilities of all outgoing transitions and the final probability sum to one: $p_q + \sum_{q_n \in Q} \sum_{e \in E} \delta(q_p, q_n, e) = 1$, for all $q_p \in Q$, p_q is the final probability of q. In this case, the sum of all probabilities of all strings sum to one. Hence this PA models a *probability distribution over strings*, i.e., over Σ^*.

The models that use the first method are useful for dealing with uncertainty. Even if we know the exact string s that has been produced by a DES $\mathcal{A}$, we sometimes we are not sure of the exact state that $\mathcal{A}$ is in. Using the first model we can determine a probability distribution over the states of $\mathcal{A}$ given s. Models that use the second method, are mainly used for modeling continuous processes. At every point in time, we can use the model to determine the probability of the next event given an observed history s. The third method results in a model for a *probabilistic language*. The probabilities of all strings in this language sum to one. Given a string s, and such a model $\mathcal{A}$, we can determine the probability that $\mathcal{A}$ generated s.

The PA model can be extended with a *cutpoint* δ on the valid probabilistic strings. This cutpoint defines a lower bound on the allowed probabilities of marked paths: A finite computation of a PA is called *valid with cutpoint* δ when $q_n \in F$ and $p \geq \delta$. A PA with a cutpoint can for example be used to generate or accept strings that are not highly unlikely to occur in practice.

Another way to represent uncertainty in the input data is to just add real numbers to transitions, without the restriction of it being a probability. This results in *weighted automata*. In weighted automata, the certainty values do not have to sum to one. Sometimes, this results in a more natural model of uncertainty.

In the remainder of this section we discuss several commonly used probabilistic models. All of these models are probabilistic generators. Usually these models are not used to model uncertainty, but to approximate the exact model. We will first discuss the (non-timed or discrete-timed) Markov chain. After that we discuss the continuous-timed Markov chain. We conclude with a probabilistic version of the timed generator.

Markov chain and hidden Markov models

A *Markov chain* (Ross 1997) is a stochastic process $\{X_n \mid n = 0, 1, 2, \ldots\}$ that takes on a finite or countable number of possible values. If $X_n = i$, then the process is said to be in state i at time n. Whenever a Markov chain is in state i at time n, there is a fixed probability P_{ij} that it will be in state j next. This is an important property known as the *Markov property*: given the present state, the future is independent of the past (the process is *memoryless*).

When such a memoryless process can only take on a finite number of possible values, it can be modeled by a special kind of PA, called a *Markov chain*. A Markov chain only models the probability of being in a certain state after a certain amount of time/steps. Thus, only the states are labeled, the transitions between states are not. For example, the PA of Figure 2.11 can be transformed into a Markov chain by removing the input values (*start*, *stop*, *dark*, *light*), and keeping the transition probabilities.

In language learning applications, a commonly used Markov chain is the *N-gram model*. This model is an approximation of the actual underlying automaton (or grammar) of the language. An N-gram model consists of states that all denote a finite sequence of past symbols of length N. For example a *trigram* model consists of all possible sequences of length three encoded in states, with transition probabilities between these states. These probabilities encode the probability of what the next symbol will be. For example from state *abc* to state *bcd* there can be a non-zero probability $P(d|abc)$, which is the probability that the next symbol will be a d given that the last three symbols were *abc*.

A more general type of Markov chain that is widely used in practice is the *Hidden Markov Model* (HMM) (Rabiner 1989). An HMM is a statistical model where the system being modeled is assumed to be a Markov chain with unknown parameters. This means that when an HMM is in a particular state an observation can be generated. This observation is generated according to a probability distribution over possible events (one for every state of the HMM), which is independent from the state transition probabilities. Only this observation, not the actual state, is visible to an external observer. The states of an HMM are said to be hidden to the outside (hence the name). Although the transitions of an HMM are unlabeled, HMMs are very similar to PAs. In fact the probability distributions they create are identical to the ones created by PAs (Dupont, Denis and Esposito 2005).

Continuous-time Markov chains

In the real world there are many examples of probabilistic systems that behave in real time. Such systems can be modeled in a way similar to stochastic clock structures. This model consists of a Markov chain with probability distributions over the amount of time it spends in each state. Such a Markov chain is called a *continuous-time Markov chain* (CTMC) (Ross 1997). Like the discrete-time (normal) Markov chain the CTMC has the Markov property. This property requires, in addition to the future states being independent of the past, that the future time spent in states is independent from past. In other words, the probability distributions over the amount of time spent in states have to be memoryless. Since the only memoryless probability distribution is the exponential distribution, the random variable that denotes the amount of time spent in a state has to be exponentially distributed. When we drop this requirement we obtain a *semi-Markov process*. In this process, the time spend in a state is allowed to have any probability distribution. In essence a CTMC is a stochastic process which has the properties that each time it enters state i:

- The amount of time T_i it spends in state i before making a transition into another state j is exponentially distributed.
- When the process leaves state i, it next enters state j with some fixed probability P_{ij} (independent from T_i). The transition probabilities must satisfy two properties: they sum to one ($\sum_j P_{ij} = 1$ for all i) and transitions to the same state are impossible ($P_{ii} = 0$ for all i).

CTMCs are widely used to model real-time systems and to model their performance and reliability. For example a CTMC of a software system can be used to determine the average failure time, or to find performance bottlenecks.

Stochastic clock structures

In a structured timed generator (STG, see Section 2.3.2) the event lifetimes are predetermined in a clock structure, which contains fixed sequences of real numbers. However, this is unrealistic for systems that operate in uncertain environments. In order to develop realistic DES techniques in the presence of uncertainty, a more refined model of the clock structure is required. In this refined model the sequences of real numbers are specified as stochastic sequences. This means that for each event i we no longer have a sequence of real numbers at our disposal, but a distribution function G_i that describes the *random clock sequence* c_i (Cassandras and Lafortune 2008). Formally:

Definition 2.36. *(stochastic clock structure) A* stochastic clock structure *associated with an event set Σ is a set of distribution functions $G = \{G_i : i \in \Sigma\}$. Each G_i, where $i \in \Sigma$, characterizes a random clock value (lifetime) sequence $c_i = t_{i,1}, t_{i,2}, \ldots$.*

The random clock value sequences are often assumed to be idd (independent and identically distributed). There are, however, several ways in which a clock structure can be extended to include situations where elements of a sequence c_i are correlated, or two clock value sequences are dependent on each other.

In addition to a stochastic clock structure a *probabilistic structured timed automaton* (PSTA) is equipped with two additional probabilistic features: a probability mass function of the start state and a transition probability for each transition. A PSTA basically is the probabilistic version of a STA.

The output of a PSTA is a stochastic process known as a *generalized semi-Markov process* (GSMP). A GSMP is a semi-Markov process where the distribution over the time spent in a state is not given beforehand. This distribution depends on the distribution functions in the probabilistic clock structure, and the clock and score updating mechanisms (which are identical to those in an STA). PSTAs are often used for simulation purposes, since they generate the possible output of a probabilistic DES.

2.4 Language identification

In this section we describe several well-known learning algorithms for the untimed and probabilistic automata described in Section 2.3. The problem of learning a model from data is known as *system identification*. Our problem is therefore known as *automaton identification*, which is also known as *grammatical inference*. The data that is used for the learning process comes from observations of an operational system. This setting occurs when we do not know how a system works internally, yet we can use sensors to monitor it.

We are mainly interested in learning the *structure* of automata. In practice, people often try to learn the parameters of a given structure. Examples are the learning of conditional probabilities in a Bayesian network or weight values in a neural network. The structure of the model needs to be generated from expert knowledge before the learning algorithm starts. The benefit of this approach is that the learning problem is relatively easy. The drawback of this approach is that we do not gain knowledge about how a system operates. If we want to answer questions regarding the operation of an unknown system, identifying the structure is necessary.

The problem of learning the structure of an unknown automaton is to find a (non-unique) smallest automaton that can generate (is consistent with) some given input data. This DFA has to be as small as possible because of an important principle in learning theory (or philosophy of science), known as Occam's razor. This states that the simplest possible explanation is the best possible. A smaller automaton is simpler, and therefore a better explanation for the observed examples.

Unfortunately, finding such a smallest DFA can be very difficult. It is the optimization variant of the problem of finding a DFA of fixed size, which has been shown to be NP-hard when the data contains both positive and negative data (Gold 1978). Even more troublesome is the result that this optimization problem cannot be approximated within any polynomial (Pitt and Warmuth 1989). When the data contains only positive examples, it is impossible to provably learn the correct DFA (Gold 1967). However, it is possible to use a probabilistic automaton (PA) in order to learn the distribution of the data. Unfortunately, this problem is again difficult: the problem of approximating a PA distribution cannot be solved efficiently unless $\mathsf{NP} = \mathsf{RP}$ (Abe and Warmuth 1990).

In spite of these hardness results, quite a few DFA learning algorithms exist. In this section, we discuss two important algorithms for this problem: state-merging and query learning of DFA. The state-merging method is the most common method for learning the structure of automata from both positive and negative data. Essentially, state-merging is a heuristic method that tries to find a good local optimum. In the limit, however, the algorithm is guaranteed to efficiently converge to the correct DFA, i.e., to the global optimum. We describe this method in Section 2.4.1.

Query learning of DFA is an elegant and efficient algorithm that shows the power of learning from queries. It is difficult to apply this algorithm to a real-world problem, since it requires equivalence queries. However, we know that we can easily construct a PAC learning algorithm that uses just membership queries by creating approximations of the equivalence queries. The algorithm is described in Section 2.4.2.

State-merging can also be applied to probabilistic automata. In this case there is no need for negative examples. This is useful in many applications since positive data is easy to come by: just use sensors to monitor the system. Pure negative data, is a lot harder to obtain. It is possible to use data obtained by monitoring other systems as negative data. But usually, there exists no system that generates exactly the inverted language. The state-merging method for probabilistic automata is explained in Section 2.4.3.

When the system is a continuous process that produces data non-stop, the positive data consists of a single (very long) generated string. The state-merging algorithm for identifying DFAs requires a set of strings as input. We show how and why a simple pre-processing of the single string is sufficient to apply state-merging in this setting in Section 2.4.4.

We end this section with some pointers to algorithms for the identification of timed automata in Section 2.5. All of these timed automata identification algorithms are based on the algorithms for the identification of untimed automata.

2.4.1 Stare-merging

An automaton identification process tries to find an automaton $\mathcal{A}$ such that its language $L(\mathcal{A})$ is equal to the target language L_t. The identification process should get some form of data as input from that it can identify L_t. We assume it is given a pair of finite sets of positive sample strings $S_+ \subseteq L_t$ and negative sample strings $S_- \subseteq L_t^{\mathrm{C}}$, called the *input sample*. The goal is to find the *smallest* DFA that is *consistent* with $S = \{S_+, S_-\}$, i.e., accepting all positive and rejecting all negative

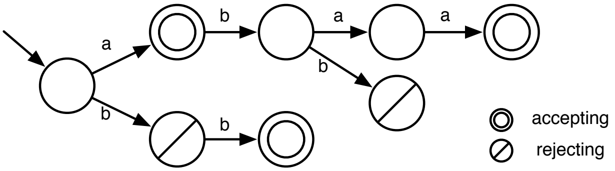


Figure 2.12: An augmented prefix tree acceptor for $S = (S_+ = \{a, abaa, bb\}, S_- = \{abb, b\})$.

strings.

Algorithm 2.4 Construct the APTA: apta

```
Require: an input sample $S = \{S_+, S_-\}$
Ensure: $\mathcal{A}$ is the APTA for $S$
  $\mathcal{A}$ is a DFA containing only a start state $q$
  for each sample strings $\sigma = a_1, \ldots, a_n$ from $S$ do
    state $q' = q$, integer $i = 1$
    while $i \leq n$ do
      if $\mathcal{A}$ contains no transition $\delta$ with $q'$ as source and $a_i$ as symbol then
        Add a new state $q''$ to $\mathcal{A}$.
        Add such a transition $\delta$ to $\mathcal{A}$, set its target to be $q''$.
      end if
      $q'$ = the target of $\delta$, $i = i + 1$
    end while
    if $\sigma \in S_+$ then
      Set $q'$ to be an accepting state.
    else
      Set $q'$ to be a rejecting state.
    end if
  end for
  Return $\mathcal{A}$
```

The idea of a state-merging algorithm is to first construct a tree-shaped DFA $\mathcal{A}$ from this input, and then to merge the states of $\mathcal{A}$. This DFA $\mathcal{A}$ is called an *augmented prefix tree acceptor* (APTA), see Figure 2.12. Algorithm 2.4 shows its construction routine apta. An APTA $\mathcal{A}$ is a DFA that is *consistent* with the input sample S. Moreover, it is such that there exists only one path from the start state to any other state. This implies that the computations of two strings s and s' reach the same state q if and only if s and s' share the same prefix until they reach q, hence the name prefix tree. An APTA is called augmented because it contains (is *augmented* with) states that are neither accepting nor rejecting. No execution of any sample string from S ends in such a state. Therefore, it is unknown whether these should be accepting or rejecting. This is determined by merging the states of this APTA and trying to find a DFA that is as small as possible.

A merge (see Figure 2.13 and Algorithm 2.5) of two states q and q' combines the states into one: it creates a new state q'' that has the incoming and outgoing transitions of both q and q'. Such a merge is only allowed if the states are *consis-*

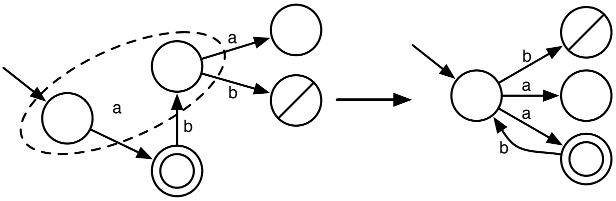


Figure 2.13: A merge of two states from the APTA of Figure 2.12. On the left the original part of the automaton is shown, the states that are to be merged are surrounded by a dashed ellipse. On the right the result of the merge is shown. This resulting automaton still has to be determinized.

tent, i.e., it is not the case that q is accepting while q' is rejecting or vice versa. When a merge introduces a non-deterministic choice, i.e., q'' is the source of two transitions with the same symbol, the target states of these transitions are merged as well. This is called the determinization process (the while loop in Algorithm 2.5), and is continued until there are no non-deterministic choices left. The result of a merge is a new DFA that is smaller than before, and still consistent with the input sample S. A state-merging algorithm continually applies the state-merging process until no more consistent merges are possible.

Algorithm 2.5 Merging two states: merge

```
Require: an augmented DFA $\mathcal{A}$ and two states $q, q'$ from $\mathcal{A}$
Ensure: if $q$ and $q'$ are consistent, then $q$ and $q'$ are merged, $\mathcal{A}$ is updated accordingly,
  and true is returned, false is returned otherwise
  if $q$ is accepting and $q'$ is rejecting or vice versa then
    Return false
  end if
  Add a new state $q''$ to $\mathcal{A}$ that is neither accepting nor rejecting.
  if $q$ or $q'$ is an accepting state then
    Set $q''$ to be an accepting state.
  end if
  if $q$ or $q'$ is a rejecting state then
    Set $q''$ to be a rejecting state.
  end if
  for each occurrence of $q$ or $q'$ as source or target of transitions $\mathcal{A}$ do
    Replace the occurrence of $q$ or $q'$ by $q''$.
  end for
  while $\mathcal{A}$ contains a non-deterministic choice of transitions with target states $q_n$ and
  $q'_n$ do
    boolean $b = \mathsf{merge}(\mathcal{A}, q_n, q'_n)$
    if $b$ equals false then
      Undo the merge of $q$ with $q''$ and return false.
    end if
  end while
  Return true
```

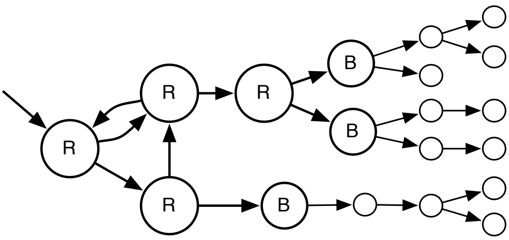


Figure 2.14: The red-blue framework. The red states (labeled R) are the identified parts of the automaton. The blue states (labeled B) are the current candidates for merging. The uncolored states are pieces of the APTA.

Algorithm 2.6 State-merging in the red-blue framework

```
Require: an input sample $S$
Ensure: $\mathcal{A}$ is a small DFA that is consistent with $S$
  $\mathcal{A} = \mathsf{apta}(S)$
  Color the start state $q$ of $A$ red and all the children of $q$ blue
  while $\mathcal{A}$ contains blue nodes do
    if $\mathcal{A}$ contains a red state $r$ and a blue state $b$ such that $\mathsf{merge}(\mathcal{A}, r, b)$ equals true
    then
      Perform the merge
    else
      Change the color of a blue state in $\mathcal{A}$ to red
    end if
    Change the color all uncolored children of red states in $\mathcal{A}$ to blue
  end while
  Return $\mathcal{A}$
```

The red-blue framework follows the state-merging algorithm just described, but in addition maintains a core of red states with a fringe of blue states (see Figure 2.14 and Algorithm 2.6). A red-blue algorithm performs merges only between blue and red states. The new states resulting from these merges are colored red. During a merge, the determinization process can only merge uncolored states with any other state q. The new states resulting from these merges take the color of q. If no red-blue merge is possible the algorithm changes the color of a blue state into red; we call this changing of color a color operation. The algorithm is guaranteed not to change any of the transitions between red states. The red core of the DFA can be viewed as a part of the DFA that is assumed to be correctly identified.

Note that when one looks at just the red nodes in the red-blue framework, the state-merging algorithm is equivalent to an algorithm that starts with a single state automaton, and that adds transitions (possibly to new states) until a solution is found.

A red-blue state-merging algorithm is capable of producing any DFA that is consistent with the input sample and smaller than the original APTA. The main goal of a DFA identification algorithm is to find one of the smallest such DFAs. Currently, the most successful method to find such a DFA is evidence driven state-merging (EDSM) (Lang, Pearlmutter and Price 1998). In EDSM each possible merge is given a score based on the amount of *evidence* in the merges that are performed by the merge and determinization processes. A possible merge gets an evidence score equal to the number of accepting states merged with accepting states plus the number of rejecting states merged with rejecting states. At each iteration of the EDSM algorithm, the merge with the highest evidence value is performed.

The evidence measure that is used by EDSM is based on the idea that bad merges can often be avoided by performing merges that have passed the most tests, and are hence most likely to be correct. Using this evidence measure EDSM participated in and won (in a tie with the SAGE algorithm) the Abbadingo DFA learning competition in 1997 (Lang et al. 1998). The data set in this competition consisted of sparse data-sets. In the competition EDSM was capable of approximately (with 99% accuracy) learning a DFA with 500 states and an alphabet of size 2 from a training set consisting of 60.000 strings.

Theoretically, there exist polynomial characteristic sets (see Section 2.2.5) that force the state-merging algorithm (with some minor changes) to return the correct DFA. Originally, this was shown using a different algorithm called RPNI (Oncina and Garcia 1992). However, it is quite straightforward to see that RPNI is a special kind of state-merging algorithm. Hence, state-merging (with some minor changes) converges efficiently in the limit to the correct DFA. The problem is still hard however (as mentioned in the introduction) and it is therefore often beneficial to use some form of search procedure around the basic state-merging algorithm. Some results of the state-merging algorithm with different search procedures can be found in (Bugalho and Oliveira 2005).

2.4.2 Query learning of DFA

Most DFA learning algorithms (such as state-merging) are passive learning algorithms. These algorithms have no information besides the finite input sample. In some real-world applications, however, one can have more information than just an input sample. For example, information can be available in the form of the knowledge of a domain expert. This kind of information can be modeled in the form of a teacher in the query learning framework (see Section 2.2.2). In this framework a polynomial time DFA learning algorithm exists that exactly learns the target DFA. This is not very surprising since DFAs are learnable in polynomial time given a characteristic set, and there is a close relation between the answers from a teacher and characteristic sets (Parekh and Honavar 2000). In order to show the power of query learning we discuss a DFA learning algorithm known as *Angluin's algorithm* (Angluin 1987). The description below is from (Kearns and Vazirani 1994).

Angluin's algorithm uses two types of queries: membership queries and equivalence queries. The idea of the algorithm is to continuously discover new states of the target DFA. A new state is a state that exhibits behavior that is demonstrably different from the states discovered so far. The algorithm consists of one main loop. In this loop the algorithm performs the following actions:

- Construct a hypothesis DFA whose states are the currently discovered states.

- Ask the teacher an equivalence query.
- Use the counterexample from this query to discover a new state by using membership queries.

When all the states of the target DFA have been discovered, the equivalence query will have yes as its answer, and the algorithm stops. In order to discover information about the states of a DFA $\mathcal{A}$, the algorithm maintains a set C of *access strings* and a set D of *distinguishing strings*:

- The computation of each *access string* $c \in C$ in $\mathcal{A}$ leads to a different state of $\mathcal{A}$; we denote this state by $\mathcal{A}[c]$.
- For each pair of strings $c, c' \in C$ such that $c \neq c'$, there is a *distinguishing string* $d \in D$ such that the computation of one of the strings cd and $c'd$ reaches a final state of $\mathcal{A}$ and the other does not.

The goal of the algorithm is to discover all the states of $\mathcal{A}$, by finding $size(\mathcal{A})$ access strings C, together with a set of distinguishing strings D (one for every pair of access strings). The sets C and D are maintained in a structure known as a *binary classification tree*. In this tree each internal node is labeled by a string from D, and each leaf is labeled by a string from C. This tree is constructed in the following way:

- The root contains an arbitrary distinguishing string $d \in D$.
- All strings $c \in C$ such that cd is rejected by $\mathcal{A}$ are placed in the left subtree of the root.
- All strings $c \in C$ such that cd is accepted by $\mathcal{A}$ are placed in the right subtree of the root.
- Recurse at each subtree (pick another distinguishing string $d \in D$, etc.) until each string in C has its own leaf.

In the algorithm the binary classification tree is constructed in such a way that the distinguishing string of the root is always the empty string ϵ. This ensures that all the access strings to accepting states lie in the right subtree and the access strings to rejecting states in the left subtree. In addition, ϵ is always one of the access strings. This allows us to access the start state of the DFA. Figure 2.15 shows an example of a DFA and a corresponding binary classification tree.

The classification tree is used in two simple subroutines of the algorithm. The first one, called *sifting down the classification tree* (denoted sift()), simulates a computation of the target DFA. This simulation partitions the states of the target DFA into several *equivalence classes*. These equivalence classes are the leaves of the classification tree. The second subroutine *constructs the hypothesis DFA* (denoted create_hypo()). We give a short description of both subroutines:

- *Sifting down the classification tree:* Given a string s, starting at the root of the classification tree, get the label $d \in D$ of the internal node, and make a membership query on the string sd and go left on reject and right on accept. Continue until a leaf node is reached, return the label $c \in C$ as result.

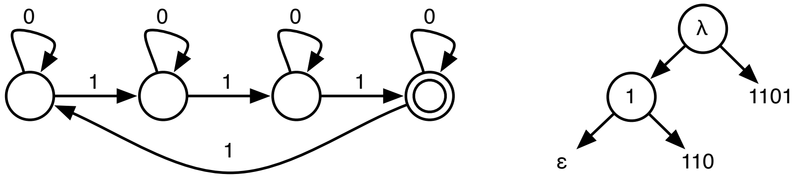


Figure 2.15: An automaton that accepts string for which the number of 1's in the input equals 3 modulo 4. To the right side of the automaton is a classification tree for this automaton. The state reachable by the string 1101 is a final state, so it is placed on the right-hand side. The automaton does not accept the string 110, but does accept the string 1101. Therefore an access string 110 is placed left of the distinguishing string ϵ, and right of the distinguishing string 1.

- *Constructing the hypothesis DFA:* For each $c \in C$ create a state with label c. For each state with label s, and each symbol from the alphabet $a \in \Sigma$, create a transition labeled with a from the state labeled with c, to the state labeled with the result of sifting sb down the classification tree.

When the algorithm asks an equivalence query of the hypothesis DFA, the answer will either be a counterexample, or an answer that the hypothesis DFA is correct. Given the counterexample s and the two subroutines sift() and create_hypo() it is easy to discover a new state of the DFA. The idea is to simulate the behavior of both the target DFA and the hypothesis DFA on the string s. This simulation is possible because of the way the classification tree is constructed. The fact that ϵ is one of the access strings guarantees that the start states of the hypothesis DFA and the target DFA will always coincide. The fact that ϵ is the distinguishing string of the root node guarantees that ϵ is used as distinguishing string during the *sift* procedure for every access string. Because of this, no equivalence class contains both a final and a non-final state. This implies that the simulation of both automata on s starts in the same equivalence class, and since s is a counterexample they will end up in different equivalence classes.

Knowing this we can use the counterexample to update the classification tree, i.e., to discover a new state of the DFA. Let $s[i]$ denote the prefix of s of length i. Let p be the first index such that the equivalence classes of both automata on $s[p]$ are different. We know that the last symbol of $s[p]$ caused transitions from the same equivalence class c_1 to different equivalence classes c_2 and c_3. Since the transition is caused by the same symbol, the equivalence class c_1 has to be a class consisting of at least two states. Every time this occurs the algorithm splits the leaf node of c_1, creating one new leaf node c_4 with $s[i-1]$ as its access string. If d is the distinguishing string between c_2 and c_3, then the correct distinguishing string between c_1 and c_4 is $s[i]d$. We denote this operation of updating the classification tree by update_tree().

Now that we know how to use a counterexample, all the algorithm needs is a loop that continuously calls for counterexamples until the DFA is correct. Algorithm 2.7 shows this main loop of Angluin's algorithm. Figure 2.16 shows an execution trace of this algorithm. The algorithm is correct since, as long as the

Algorithm 2.7 Angluin's algorithm for query learning a DFA

```
Require: A teacher for membership mem() and equivalence eq() queries.
Ensure: $\mathcal{A}$ is the DFA that the teacher wants us to learn.
  Call $mem(\epsilon)$ to determine whether the start state is accepting or rejecting.
  Construct a hypothesis DFA $\mathcal{A}$ consisting of this single state.
  Call $s := \mathsf{eq}(\mathcal{A})$.
  Initialize the classification tree $\mathcal{T}$ to have a root labeled $\epsilon$ and two leaf nodes with
  access strings $\epsilon$ and $s$.
  while true do
    $\mathcal{A} = \mathsf{construct\_hypo}(\mathcal{T})$.
    Call $s := \mathsf{eq}(\mathcal{A})$.
    if $s \neq \top$ ($\mathcal{A}$ is correct) then
      return
    end if
    $\mathsf{update\_tree}(\mathcal{T}, \mathcal{A}, s)$
  end while
```

number of leaves of the classification tree is smaller than the size of the target DFA, the hypothesis automaton has to be different. Therefore an equivalence query will return a counterexample that is used to update the classification tree. Eventually the classification tree will have a number of leaves equal to the size of the target DFA; the DFA corresponding to this classification tree is necessarily the target DFA. The algorithm uses a polynomial amount of queries and runs in polynomial time (see (Kearns and Vazirani 1994) for a complete analysis).

2.4.3 Stare-merging probabilistic automata

The problem of learning probabilistic automata (PAs) is somewhat different from the problem of learning DFAs. The difference lies in that there is no real need for negative examples. In the problem of learning DFAs, negative examples are imperative, since otherwise there is no real reason for excluding the possibility that the language we are trying to learn does not consists of all possible words (Σ^*). In fact, it has been known for a long time that regular languages are not identifiable in the limit from only positive examples (Gold 1967).

In PAs, however, the problem consists of learning a *distribution over strings*. Such a distribution can be learned from just positive examples. For example, suppose we are given a positive sample of size n, and we observe that some word occurs t times. Then we must try to find a PA that distributes over strings in such a way that the probability of that word occurring equals $\frac{t}{n}$. Naturally this has to hold for every word in the input sample. Because a larger PA is capable of producing more possible distributions, and because we believe that the simplest explanation is the best, the problem is to find the smallest PA that could have produced (with sufficient confidence) the distribution found in the sample. This distribution is known as the *sample distribution*, and is defined as:

$$\psi(s) = \frac{\mathsf{count}(s)}{\sum_t \mathsf{count}(t)}$$

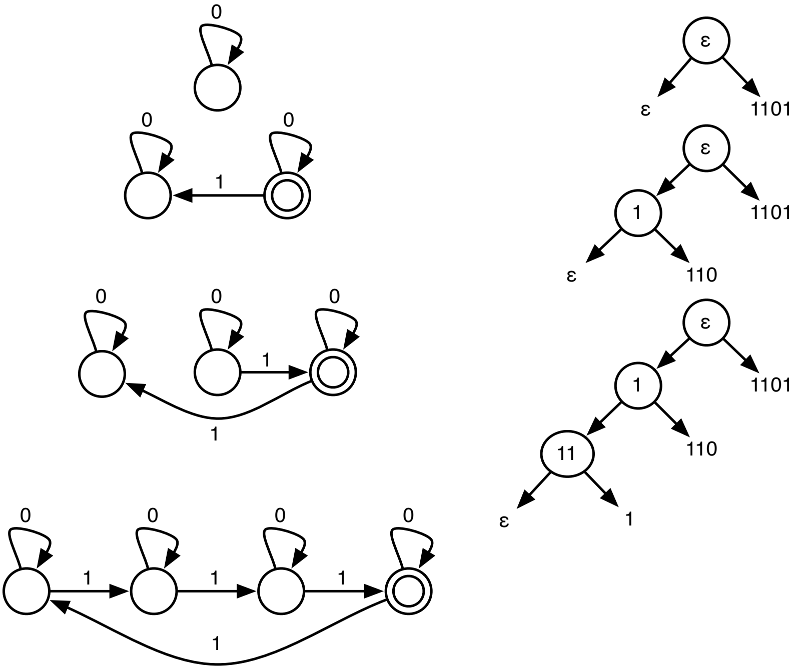


Figure 2.16: An execution of Angluin's algorithm on the automaton from Figure 2.15. The left hand side shows the hypotheses used in equivalence queries. On the right hand side the evolution of the binary classification tree is shown. Every equivalence query is answered with the same counterexample $(1101, \mathit{true})$, until the hypothesis is correct.

Here s and t are strings, $\psi(s)$ is the probability that string s is generated, and count is a function which returns the number of times a word occurs in the input sample. Like the DFA induction algorithms, most of the induction algorithms for PAs are based on the state-merging technique. The PA state-merging algorithms start with a ***probabilistic prefix tree acceptor*** (PPTA), that represents the sample distribution. A PPTA is the probabilistic equivalent of a PTA. The method that is used to construct a PPTA is similar to the method for construction a PTA. The only difference is that now each arc needs some probability value. Given a state q and a symbol a, the probability of firing the transition from q with symbol a is computed by just taking the amount of examples that fire this transition (denoted $\mathsf{count}(q, a)$) divided by the total amount of examples that leave q (denoted $\mathsf{count}(q)$). The resulting PPTA produces exactly the sample distribution. Also, the merge operation of two states in a PA is similar to the method for merging two states in a DFA. The only difference being that the probability values need to be updated.

The main difference between DFA and PA state-merging algorithms is the check for compatibility. In DFA state-merging they are compatible if there is no inconsistency. In PA state-merging they are compatible if some statistical

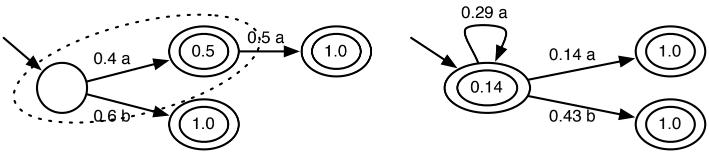


Figure 2.17: A probabilistic merge. The final states contain numbers denoting the final probabilities. The left automaton is a PPTA constructed from the input sample: $S_+ = \{a, aa, b, b, b\}$. The two states surrounded by a dashed line are merged. The right automaton is the PA resulting from the merge of these states.

criterion is satisfied. For example the ALERGIA algorithm uses a compatibility measure derived from the Hoeffding bound (Carrasco and Oncina 1994). Using this criterium two states q and q' are μ-compatible if the following two conditions hold for all $a \in \Sigma$:

1. $\left|\frac{\mathsf{count}(q,a)}{\mathsf{count}(q)} - \frac{\mathsf{count}(q',a)}{\mathsf{count}(q')}\right| < \sqrt{\frac{1}{2}ln(\frac{2}{\mu})}\left(\frac{1}{\sqrt{\mathsf{count}(q)}} + \frac{1}{\sqrt{\mathsf{count}(q')}}\right)$

2. $\delta(q, a)$ and $\delta(q', a)$ are μ-compatible

The first condition defines the compatibility using some precision parameter μ. The second condition requires that the compatibility is satisfied in every pair of children of q and q'. Another difference is in the stopping condition of the merging algorithms. A DFA state-merging algorithm stops when all possible merges are inconsistent. A PA merging algorithm can have a statistical stopping criterion. The ALERGIA algorithm stops when all possible merges are not μ-compatible. It can be shown that the ALERGIA algorithm identifies PAs in the limit with probability one.

There are also approaches to learning PAs other than state-merging, such as *successive state splitting* (Takami and Sagayama 1992). The idea of this method is to start with a small PA (possible a single state), and then to split states based on the training data. The algorithm uses split operations to maximize the expected maximum likelihood. When the likelihood gain falls below a certain threshold the algorithm stops. Due to the way in which the split operations are defined, this method can only be used to learn PAs without any cycles (unless these cycles were already defined in the original PA). There are also methods that use parameter estimation to learn the structure of a PA. The MAP learning approach (Brand 1999), for example, starts with a large initial PA, and then uses parameter estimation to drive irrelevant parameters to zero. Simple statistical tests can then be used to prune transitions and states from the PA, while increasing the posterior probability. Techniques based on parameter estimation are often used to learn N-gram modes.

In addition, there exists many other techniques similar to ALERGIA. For example, there is one that learn PAs under a slightly modified PAC criterion (Clark and Thollard 2004). A modification is required because it has been shown that PAs cannot be approximated efficiently under the original criterion (and some cryptographic assumptions) (Kearns, Mansour, Ron, Rubinfeld, Shapire and Sellie 1994).

This algorithm for learning PAs uses a setup similar to ALERGIA, but it uses different tests in order to determine whether two states are the same. In addition, it comes with a lower bound on the amount of data that is required to satisfy the modified PAC criterion. Hence, this algorithm has stronger learning properties than ALERGIA (ALERGIA converges only in the limit). A problem of this algorithm is that it requires a huge amount of examples and many user-specified parameters before it satisfies the PAC criterion. Recently, an attempt has been made to modify the algorithm in order to resolve these issues (Castro and Gavaldà 2008).

2.4.4 Computational mechanics

We just described how to identify a DFA from only positive data. In many real-world applications, such as modeling reactive systems, it is often the case that the system never stops producing events. Such systems (or processes) are called *continuous*. They produce an infinite sequence of events. We can only observe a finite part of this infinite sequence. In other words, we need to learn from a single positive string. In this section, we show how to adapt the algorithms from the previous section to this setting.

The method and theory that can be used to adapt state-merging to the problem of identifying continuous processes comes from the field of *mechanics*. Mechanics is the branch of physics concerned with the behavior of physical bodies and their environment when subjected to forces or displacements. In the physical world, many processes exist that exhibit patterns in their behavior. In (Shalizi and Crutchfield 2001), a thorough study is performed on how information theory can be used to predict the future behavior of such processes. What they call *computational mechanics* is a way of modelling a physical process by using a computational method. The thing that makes their approach different from traditional statistical mechanics is that they do not only want to recognize or predict these patterns, but they also want to find the causal structure of the process that actually generated these patterns. Learning this structure should lead to genuine understanding of the underlying physical process.

Computational mechanics models a physical process as a bi-infinite sequence of random variables $\overleftrightarrow{X} = \ldots X_{-2}X_{-1}X_0X_1X_2\ldots$. These variables can take any value from a countable set, which in our case is the set of events Σ. Thus, we are interested in modeling a system that is capable of generating a bi-infinite sequence of events:

$$\overleftrightarrow{\sigma} = \ldots e_{-2}e_{-1}e_0e_1e_2\ldots$$

Naturally, one can never observe a complete bi-infinite sequence $\overleftrightarrow{\sigma}$. When observing a system producing $\overleftrightarrow{\sigma}$, one will only see a *snapshot* (finite consecutive subsequence) of $\overleftrightarrow{\sigma}$ of length l up to or starting at some point in time t:

$$\overleftarrow{\sigma}^{l}_{t} = e_{t-l}e_{t-l+1}\ldots e_{t-2}e_{t-1} \text{ or } \overrightarrow{\sigma}^{l}_{t} = e_te_{t+1}\ldots e_{t+l-2}e_{t+l-1}$$

Given such a snapshot we want to determine the (behavior of the) system that produced it. In computational mechanics, these systems (processes) are assumed to be stationary, i.e., the probability of a certain snapshot is time-translation invariant: $P(\overleftarrow{X}_t^n = e_1 \dots e_n) = P(\overleftarrow{X}_u^n = e_1 \dots e_n)$ and $P(\overrightarrow{X}_t^m = e_1 \dots e_m) = P(\overrightarrow{X}_u^m = e_1 \dots e_m)$ for all $t, u \in \mathbb{Z}, m, n \in \mathbb{N}$, and $e_i \in \Sigma$. Hence, we can safely drop the subscript t from the snapshots. The goal of computational mechanics is to predict the future $F = \overrightarrow{X}^m$ of a process based only on the observed finite history $h = \overleftarrow{e}^n = e_1 \dots e_n$ and nothing else.

In prediction, there is always some uncertainty regarding the future. Information theory captures this uncertainty in the notion of entropy. The entropy $H[X]$ of a random variable X taking values in a countable set E is:

$$H[X] = -\sum_{e \in E} P(X = e) \log_2 P(X = e)$$

In this equation, $P(X = e)$ is the probability that X takes the value e. Intuitively, the entropy of X is the expected number of bits that are needed to represent the value of X in a binary string. For a good introduction to information theory, see (Cover and Thomas 2006).

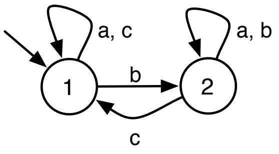
Example 2.10. Suppose X is distributed such that $P(X = e_1) = \frac{1}{3}$, $P(X = e_2) = \frac{1}{3}$, and $P(X = e_3) = \frac{1}{3}$. Then the entropy of X is: $H(X) = -\frac{1}{3} \log \frac{1}{3} - \frac{1}{3} \log \frac{1}{3} - \frac{1}{3} \log \frac{1}{3} \approx 1.58$ bits. When these probabilities are less evenly distributed, often occurring values can be encoded using few bits and rarely occurring values using many bits. Hence, in this case less bits will be required on average. For example, suppose X is distributed such that $P(X = e_1) = \frac{3}{4}$, $P(X = e_2) = \frac{1}{8}$, and $P(X = e_3) = \frac{1}{8}$. Then the entropy of X is: $H(X) = -\frac{3}{4} \log \frac{3}{4} - \frac{1}{8} \log \frac{1}{8} - \frac{1}{8} \log \frac{1}{8} \approx 1.06$ bits.

A prediction method m is a mapping from finite histories to probability distributions of futures. With slight abuse of notation, we use $m(h) = P(F \mid m, h)$ to denote these distributions. Such a method uses patterns that were observed in the past in order to reduce the uncertainty about the future $H[F \mid m, h]$. However, because the past contains all information, mapping the past to patterns can only result in a loss of information. Therefore, m can never do better than an optimal method for prediction that uses the entire infinite history $\overleftarrow{e} = \dots e_{-2} e_{-1}$ of the process in order to predict the future, i.e., for any m and h:

$$H[F \mid m, h] \geq H[F \mid \overleftarrow{e}]$$

In other words, conditioning on the entire past reduces the uncertainty about the future as much as possible. A prediction method tries to remember patterns (i.e., relevant bits of information) from finite histories in order to achieve this bound on the uncertainty. When a prediction method achieves this bound it is called *prescient*.

Given a process $\overleftrightarrow{X}$ and a prediction method m, a partitioning of the histories of $\overleftrightarrow{X}$ can be made based on whether these histories result in identical predictions. The set of effective states Q of a process under m is a partitioning of all possible histories such that each state $q \in Q$ is a set that contains a history h if and only if $m(h) = m(h')$ for every other history $h' \in q$. The goal of a learning algorithm should be to find a predictor m such that:



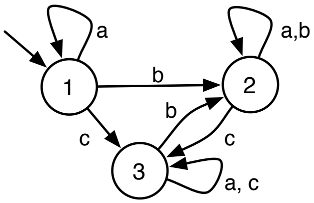


Figure 2.18: The right DFA are the causal states or ϵ-machine of the left DFA.

- The effective states Q under m are prescient, i.e., $H[F \mid Q] = H[F \mid \overleftarrow{e}]$. Prescient states are sufficient statistics for the process's future, and therefore they implement the optimal prediction strategy.
- The effective states Q under m should be as simple as possible, i.e., for any other set of prescient states Q' it holds that $H[Q] \leq H[Q']$. This implements Occam's razor: all other things being equal (there are several optimal predictors), the simplest solution is to be preferred.

The causal states S of a process are defined as prescient states of minimal complexity. Moreover, causal states are the unique combination of these two properties.

Definition 2.37. *(causal states) The causal states S of a process are the effective states of an optimal prediction method ϵ that uses infinite histories in order to predict the future, i.e., they are sets $s \in S$ of infinite histories such that $\overleftarrow{e} \in s$ if and only if for all $\overleftarrow{e'} \in s : \epsilon(\overleftarrow{e}) = \epsilon(\overleftarrow{e'})$,where $\epsilon(\overleftarrow{e}) = P(F \mid \overleftarrow{e})$.*

Given a process's causal states it is straightforward to define the transitions between these states:

Definition 2.38. *The causal transitions T of a process $\overleftrightarrow{X}$ are labeled transition probabilities between the causal states C of $\overleftrightarrow{X}$. Each transition $t \in T$ is a quadruple $\langle c, c', e, p\rangle$ consisting of a source state $c \in C$, a target state $c' \in C$, an event $e \in E$, and a transition probability $p \in [0, 1] \subset \mathbb{R}$. The transition probability $p = P(\overleftarrow{e}a \in c', a = e \mid a \in E, \overleftarrow{e} \in c)$ is the probability that the next state is c' and the next event is e given that the current state is c.*

The combination of causal states and transitions is called an ϵ-machine. From the construction of the causal states and the causal transitions it is fairly straightforward to prove that the causal states form a Markov process and the ϵ-machine is deterministic. In other words, the structure of an ϵ-machine is that of a probabilistic deterministic finite state automaton (PDA). Thus, the optimal predictor of minimal complexity for a process $\overleftrightarrow{X}$ is a deterministic automaton.

Intuitively, we try to identify a model that determines the current state of the system, given some finite past observation. Of course, it can be the case that this past observation is not sufficient information to determine the current state.

Example 2.11. Suppose we want to identify the left system (process) depicted in Figure 2.18. For this process, an occurrence of an a event is not sufficient to determine whether the current state of the system is q_0 or q_1. However, an occurrence of ab is sufficient. Hence, the actual model we want to identify is the right DFA in Figure 2.18. This is the ϵ-machine (containing the causal states) of the left DFA.

As a consequence, we can use the state-merging algorithm in order to find this optimal predictor.

2.5 Timed automaton identification

The problem of identifying a DFA from a data set is a well-studied problem in learning theory (de la Higuera 2005). There are, however, very few studies on the identification of a TA from data. Closely related work deals with the problem of learning event recording automata (ERAs), which is a restricted but still powerful class of TAs (Grinchtein et al. 2006). That work proposes an algorithm for learning these TAs from a timed teacher using membership and equivalence queries. It is possible to adapt a query learning algorithm to the setting of learning from a data set by simulating the queries using the data set (Goldman and Mathias 1996). Therefore, it should also be possible to adapt the ERA query learning algorithm to the problem of learning an ERA from a data set. However, since the ERA learning algorithm requires an exponential amount of queries, this simulation would require a very large amount of data.

Other approaches to the problem of learning a timed system mainly deal with the inference of probabilistic timed models. Timed PAs can be learned from positive data with the state-merging method, similar to ALERGIA. It has, for example, been used to learn continuous time Markov chains (CTMCs) (Sen et al. 2004). Since a CTMC uses two probability values, the definition of the compatibility condition is split into two parts. For the transition probabilities the same condition is used as in the ALERGIA algorithm. For the amount of time spent in a state a condition based on the Chebyshev inequality is used. Apart from this difference the algorithm is identical to the ALERGIA merging algorithm.

Note, however, that in Markov chains there are no final states. This means that the words produced by a CTMC are of infinite length. In the CTMC learning algorithm, this is solved by generating a sample from a *finitary CTMC*. This is a CTMC that has a non-zero stopping probability in any state. This sample can be generated by taking the CTMC, picking a state, and each time it reaches this state making the generated string stop with some probability. The learning algorithm is then given this stopping probability in addition to the learning sample. It can be shown that, when a learning algorithm correctly identifies a finitary CTMC from this sample, then the CTMC underlying the identified finitary CTMC is identical to the target CTMC. A technical detail of the algorithm is that the CTMC models that are used in the algorithm have labels on their edges, in addition to labels on their states. Only the labels on the edges are used in the learning process.

One may observe that TAs are somewhat similar to continuous time Markov models: the difference being (except the transition probabilities) that these Markov models use a distribution over the execution times of events instead of time constraints. It may be the case that a probabilistic variant of an TA can be used to define exactly the same timed probabilistic languages as some continuous-time hidden semi-Markov model. This is the case for regular probabilistic DFAs and hidden Markov models (Dupont et al. 2005).

For (hidden) semi-Markov models several inference methods exists (see e.g.. (Guédon 2003)). However, these methods deal with the problem of optimizing the parameters of known models as to maximize the likelihood. Since in the context of the current setting, we do not know the structure of the model to be identified, we are interested in methods that identify the structure in addition to the parameters.

In timed automata research, the model identification problem has not received a lot of attention. However, there are several related problems that can benefit from a good model identification method. For instance, the problem of testing timed automata (Springintveld, Vaandrager and D'Argenio 2001) deals with a similar setting. Given access to a black-box system, the problem is to determine whether the system acts conform its specification. This problem is usually solved by finding a (preferably small) test set of labeled examples such that the system acts conform its specification if and only if it gives the correct labels to each of the examples. Given a model identification method and a set of labeled historic data, this problem can be solved in a different manner, without requiring the specification. The idea is to first identify a model from the historic data and then subsequently test this model either by hand (visually) or using a model verification tool, such as UPPAAL (Larsen et al. 1997).

Another related problem is the problem of controller synthesis for timed automata (Pnueli, Asarin, Maler and Sifakis 1998). Here the idea is to find a controller, that restricts the transitions a given TA is allowed to take, such that the behavior of the combined system satisfies certain properties. With the help of a model identification method, it becomes possible to synthesize controllers for unknown or black-box systems. This can be of importance in for example agent-based systems, where the internal structure of agents should remain unknown to other agents, but the behavior of the complete agent system should satisfy some properties, such as deadlock-freeness.

Closely related research comes from the temporal data-mining field (Roddick and Spiliopoulou 2002). In temporal data-mining, the objective is to discover previously unknown rules from time-series data. Moreover, these rules should be easy to understand and to validate. This is very much like the identification problem we are interested in. Identifying a TA model to discover unknown rules looks similar to one that is used to mine a hierarchy of temporal patterns from (multivariate) time-series data (Mörchen and Ultsch 2004). It is different, however, in that this approach (and most other approaches in temporal data mining) focuses on finding simple patterns or correlations in the data, and not on finding a more complex model for the actual system that generated this data.

To sum up, there is little research on the identification of TAs. Some work has been done on identifying other timed models, but these identification methods usually focus on other problems than identifying the structure of the model. In the cases that the identification methods do try to identify the model structure, either the models are much less expressive, or the identification method is inefficient. Thus, there is currently a lack of an efficient algorithm for the identification of an expressive class of TAs. In addition, there currently exist no theoretical studies regarding the efficiency of identifying TAs. In this thesis, we perform such a study in order to discover what classes of TAs can be identified efficiently. We then use the results of this study to construct an algorithm that identifies these TAs efficiently both from labeled data and from unlabeled data.

CHAPTER 3

The complexity of identifying timed automata

This chapter is based on work published in Grammatical Inference: Algorithms and Applications (Verwer, de Weerdt and Witteveen 2008b), Language and Automata Theory and Applications (Verwer, de Weerdt and Witteveen 2009).

3.1 Introduction

The main goal of this thesis is to develop efficient algorithms for the identification (learning) of timed automata (TAs) from data. Before designing these algorithms, we first have to analyze the complexity (difficulty) of identifying TAs. Such an analysis is important because it tells us whether there actually exists an efficient algorithm that solves this identification problem. If the problem turns out to be too difficult to solve efficiently, we have to find a suitable subclass of TAs that can be identified efficiently. This is the topic of this chapter.

In contrast to deterministic finite state automata (DFAs) and hidden Markov models (HMMs) (see Sections 2.3.1 and 2.3.3), until now the identification problem for TAs has not received a lot of attention from the research community. We are only aware of studies on the related problem of the identification of event-recording automata (ERAs) (Alur et al. 1999). It has for example been shown that ERAs are identifiable in the query learning framework (Grinchtein et al. 2006). However, the proposed query learning algorithm requires an exponential amount of queries, and hence is data inefficient. We would like our identification process to be efficient. This is difficult because the identification problem for DFAs is NP-complete (Gold 1978). This property easily generalizes to the problem of identifying a TA (by setting all time values to 0). Thus, unless $\mathsf{P} = \mathsf{NP}$, a TA cannot be identified efficiently. Even more troublesome is the fact that the DFA identification problem cannot even be approximated within any polynomial (Pitt and Warmuth 1989). Hence (since this also generalizes), the TA identification problem is also inapproximable.

These two facts make the prospects of finding an efficient identification process for TAs look very bleak. However, both of these results rely on there being a fixed input for the identification problem (encoding a hard problem). While in normal decision problems this is very natural, in an identification problem the amount of input data is somewhat arbitrary: more data can be sampled if necessary. Therefore, it makes sense to study the behavior of an identification process when it is given more and more data (no longer encoding the hard problem). The framework that studies this behavior is *identification in the limit* (see Section 2.2.2).

The class of all DFAs has been shown to be efficiently identifiable in the limit using a state merging method (Oncina and Garcia 1992). Also, it has been shown that the class of *non-deterministic finite automata* (NFAs) are not efficiently identifiable in the limit (de la Higuera 1997). This again generalizes to the problem of identifying a non-deterministic TA (by setting all time values to 0). Therefore, we only consider the identification problem for *deterministic timed automata* (DTAs). Our goal is to determine exactly when and how DTAs are efficiently identifiable in the limit. In this chapter, we prove many interesting results in order to achieve this goal. Our main results are:

1. Polynomial distinguishability (Definition 3.7) is a necessary condition for efficient identification in the limit (Lemma 3.9).

2. DTAs with two or more clocks are not polynomially distinguishable, and thus not efficiently identifiable (Theorem 1 and Corollary 3.10).

3. DTAs with one clock (1-DTAs) are polynomially distinguishable (Theorem 4).

4. 1-DTAs are efficiently identifiable using our ID_1-DTA algorithm (Algorithm 3.1 and Theorem 5).

5. 1-DTAs and n-DTAs (DTAs with n clocks) are language equivalent (Theorem 6).

We prove the first three results in Section 3.2. These results lay the theoretical foundation necessary for identifying DTAs efficiently. In addition, the fact that 1-DTAs are polynomially distinguishable has interesting consequences for the modeling power of DTAs and 1-DTAs. These consequences are interesting outside the scope of the DTA identification problem. For instance, it proves membership in NP and coNP for the reachability and equivalence problems for 1-DTAs (Corollaries 3.23 and 3.24), respectively. In addition, it has consequences for the power of clocks in general. We give an overview of these consequences in Section 3.5.

Our algorithm for identifying 1-DTAs efficiently is described in Section 3.3.1. In Section 3.3.2, we use the results of Section 3.2 in order to prove our fourth main result. The fact that 1-DTAs can be identified efficiently is surprising because the standard method of transforming a DTA into a DFA (by region construction, see Section 2.3.2) results in a DFA that is exponentially larger than the original 1-DTA. This blow-up is due to the fact that time is represented in binary in a 1-DTA, and in unary (using states) in a DFA. In other words, 1-DTAs are exponentially more compact than DFAs, but still efficiently identifiable.

In addition, our results show that for the purpose of identifying a timed system, 1-DTAs are more preferred models than n-DTA because they are more efficient to identify. Our fifth main result (Section 3.4.2) strengthens this preference: it shows that any n-DTA can be modeled using a 1-DTA. Thus, even when other factors (for instance expert knowledge) give preference to an n-DTA model, one could consider learning a 1-DTA model instead. Furthermore, we sketch how our algorithm can be adapted in order to identify n-DTAs instead of 1-DTAs, albeit inefficiently.

We end this chapter with a summary and discussion regarding the obtained results (Section 3.6). Before providing our results and proofs, we first recap timed automata (see Section 2.3.2) and efficient identification in the limit (see 2.2.5) for the setting of DTA identification. In addition, we introduce some additional notation that makes our proofs easier to read.

3.1.1 Deterministic timed automata

A *timed automaton* (see Section 2.3.2) is an automaton that accepts (or generates) strings of symbols paired with time values, known as timed strings. In the setting of TA identification, we always measure time using finite precision, e.g. milliseconds. Therefore, we define a finite *timed string* τ over a finite set of symbols Σ as a sequence $(a_1, t_1)(a_2, t_2) \dots (a_n, t_n)$ of symbol-time value pairs $(a_i, t_i) \in \Sigma \times \mathbb{N}$ (instead of $\mathbb{R}$). We use τ_i to denote the length i prefix of τ, i.e., $\tau_i = (a_1, t_1) \dots (a_i, t_i)$. A time value t_i in τ represents the time until the occurrence of symbol a_i as measured from the occurrence of the previous symbol a_{i-1}. We define the length of a timed string τ, denoted $|\tau|$, as the number symbol occurrences in τ, i.e., $|\tau| = n$.[1]

In TAs, timing conditions are added using a finite set X of *clocks* and one *clock guard* on every transition. These clocks may have different valuations, but all move at the same speed. A *valuation* v is a mapping from X to $\mathbb{N}$, returning the value of a clock $x \in X$. We can add or subtract constants or other valuations to or from a valuation: if $v = v' + t$ then $\forall x \in X : v(x) = v'(x) + t$, and if $v = v' + v''$ then $\forall x \in X : v(x) = v'(x) + v''(x)$. Every transition δ in a TA is associated with a set of clocks R, called the *clock resets*. When a transition δ occurs (or fires), the values of all the clocks in R are set to 0, i.e., $\forall x \in R : v(x) := 0$. The values of all other clocks remain the same. We say that δ *resets* x if $x \in R$. In this way, clocks are used to record the time since the occurrence of some specific event. Clock guards are then used to change the behavior of the TA depending on the value of clocks. A clock guard g is a boolean constraint defined by the grammar

[1] Thus, the length of a timed string is defined as the length in *bits*, not the length in *time*. This makes sense because our main goal is to obtain an efficient identification algorithm for TAs. Such an algorithm should be efficient in the size of an efficient encoding of the input, i.e., using binary notation for time values.

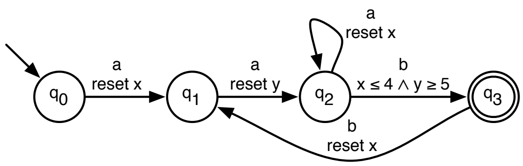


Figure 3.1: A timed automaton. The labels, clock guards, and clock resets are specified for every transition. When no guard is specified it means that the guard is always satisfied.

$g := x \leq c \mid x \geq c \mid g \wedge g$, where $x \in X$ is a clock and $c \in \mathbb{N}$ is a constant.[2] A valuation v is said to *satisfy* a clock guard g, denoted $v \in g$, if for each clock $x \in X$, each occurrence of x in g is replaced by $v(x)$, and the resulting constraint is satisfied. A timed automaton is defined as follows:

Definition 3.1. *(TA) A* timed automaton *(TA) is a 6-tuple $\mathcal{A} = (Q, X, \Sigma, \Delta, q_0, F)$, where Q is a finite set of states, X is a finite set of clocks, Σ is a finite set of symbols, Δ is a finite set of transitions, q_0 is the start state, and $F \subseteq Q$ is a set of final states.*

A transition $\delta \in \Delta$ is a tuple $\langle q, q', a, g, R\rangle$, where $q, q' \in Q$ are the source and target states, $a \in \Sigma$ is a symbol called the transition label, g is a clock guard, and $R \subseteq X$ is the set of clock resets.

Example 3.1. In the TA of Figure 3.1, the clock guard of the transition to state q_4 cannot be satisfied directly after entering state q_3 from state q_2: the value of x is greater or equal to the value of y and the guard requires it to be less then the value of y. The fact that TAs can model such behavior is the main reason why identifying TAs can be very difficult.

Definition 3.2. *(TA computation) A finite* computation *of a TA $\mathcal{A} = (Q, X, \Sigma, \Delta, q_0, F)$ over a (finite) timed string $\tau = (a_1, t_1) \ldots (a_n, t_n)$ is a finite sequence*

$$\begin{array}{ccccccccc} (q_0, v_0) & \xrightarrow{t_1} & (q_0, v_0 + t_1) & \xrightarrow{a_1} & (q_1, v_1) & \xrightarrow{t_2} & \ldots \\ \ldots & \xrightarrow{a_{n-1}} & (q_{n-1}, v_{n-1}) & \xrightarrow{t_n} & (q_{n-1}, v_{n-1} + t_n) & \xrightarrow{a_n} & (q_n, v_n) \end{array}$$

such that for all $1 \leq i \leq n : q_i \in Q$, a transition $\delta = \langle q_{i-1}, q_i, a_i, g, R\rangle \in \Delta$ exists such that $v_{i-1} + t_i \in g$, and for all $x \in X : v_0(x) = 0$, and $v_i(x) := 0$ if $x \in R$, $v_i(x) := v_{i-1}(x) + t_i$ otherwise.

We call a pair (q, v) of a state and a valuation a *timed state*. In a computation the subsequence $(q_i, v_i + t) \xrightarrow{a_{i+1}} (q_{i+1}, v_{i+1})$ represents a state transition analogous to a transition in a conventional non-timed automaton. In addition to these, a TA performs time transitions represented by $(q_i, v_i) \xrightarrow{t_{i+1}} (q_i, v_i + t_{i+1})$. A *time transition* of t time units increases the value of all clocks of the TA by t.

[2] Since we use the natural numbers to represent time, open ($x < c$) and closed ($x \leq c$) timed automata are equivalent.

One can view such a transition as moving from one timed state (q, v) to another timed state $(q, v + t)$ while remaining in the same untimed state q. We say that a timed string τ *reaches* a timed state (q, v) in a TA $\mathcal{A}$ if there exist two time values $t \leq t'$ such that $(q, v') \xrightarrow{t'} (q, v' + t')$ occurs somewhere in the computation of $\mathcal{A}$ over τ and $v = v' + t$. If a timed string reaches a timed state (q, v) in $\mathcal{A}$ for some valuation v, it also reaches the untimed state q in $\mathcal{A}$. A timed string *ends* in the last (timed) state it reaches, i.e., (q_n, v_n) (or q_n). A timed string τ is *accepted* by a TA $\mathcal{A}$ if τ ends in a final state $q_f \in F$. The set of all strings τ that are accepted by $\mathcal{A}$ is called the *language* $L(\mathcal{A})$ of $\mathcal{A}$.

In this chapter we only consider deterministic timed automata. A TA $\mathcal{A}$ is called *deterministic* (DTA) if for each possible timed string τ there exists at most one computation of $\mathcal{A}$ over τ. We only consider DTAs because the class of non-timed non-deterministic automata are already not efficiently identifiable in the limit (de la Higuera 1997). In addition, without loss of generality, we assume these DTAs to be *complete*, i.e., for every state q, every symbol a, and every valuation v, there exists a transition $\delta = \langle q, q', a, g, R \rangle$ such that g is satisfied by v. Any non-complete DTA can be transformed into a complete DTA by adding a garbage state.

3.1.2 Efficient identification in the limit

An identification process tries to find (learn) a model that explains a set of observations (data). The ultimate goal of such a process is to find a model equivalent to the actual language that was used to produce the observations, called the *target language*. In our case, we try to find a DTA model $\mathcal{A}$ that is equivalent to a timed target language L_t, i.e., $L(\mathcal{A}) = L_t$. If this is the case, we say that L_t is identified correctly. We try to find this model using *labeled data* (also called *supervised learning*): an *input sample* S is a pair of finite sets of positive examples $S_+ \subseteq L_t$ and negative examples $S_- \subseteq L_t^C = \{\tau \mid \tau \not\in L_t\}$. With slight abuse of notation, we modify the non-strict set inclusion operators for input samples such that they operate on the positive and negative examples separately, for example if $S = (S_+, S_-)$ and $S' = (S'_+, S'_-)$ then $S \subseteq S'$ means $S_+ \subseteq S'_+$ and $S_- \subseteq S'_-$.

In the *identification in the limit* framework (see Section 2.2.2), an identification process is given more and more data. The identification is considered to be successful if the identification process at some point (in the limit) converges to the target language L_t. If there exists such a process (algorithm) A, C is said to be *identifiable in the limit*. If a polynomial amount of examples in the size of the smallest model for L_t is sufficient for convergence of A, C is said to be *identifiable in the limit from polynomial data*. If A requires time polynomial in the size of the input sample S, C is said to be *identifiable in the limit in polynomial time*. If both these statements hold, then C is said to be *identifiable in the limit from polynomial time and data*, or simply *efficiently identifiable in the limit*. Efficient identifiability in the limit can be shown by proving the existence of polynomial *characteristic sets* (see Section 2.2.5). In the case of DTAs:

Definition 3.3. *(characteristic sets for DTAs) A* characteristic set S_{cs} *of a target DTA language L_t for an identification algorithm A is an input sample $\{S_+ \in L_t, S_- \in L_t^c\}$ such that:*

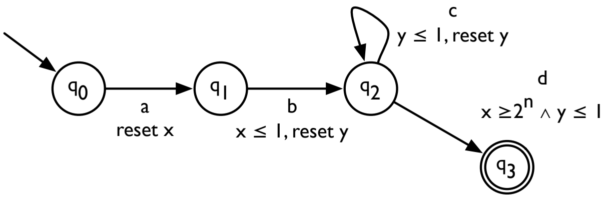


Figure 3.2: In order to reach state q_3, we require a string of exponential length (2^n). However, due to the binary encoding of clock guards, the DTA is of size polynomial in n.

1. *given S_{cs} as input, algorithm A identifies L_t correctly, i.e., A returns a DTA $\mathcal{A}$ such that $L(\mathcal{A}) = L_t$;*

2. *given any input sample $S' \supseteq S_{cs}$ as input, algorithm A still identifies L_t correctly.*

Definition 3.4. *(efficient identification in the limit of DTAs) A class DTAs C is* efficiently identifiable in the limit *(or simply* efficiently identifiable*) if there exist two polynomials p and q and an algorithm A such that:*

1. *given an input sample of size n, A runs in time bounded by $p(n)$;*

2. *for every target language $L_t = L(\mathcal{A})$, $\mathcal{A} \in C$, there exists a characteristic set S_{cs} of L_t for A of size bounded by $q(|\mathcal{A}|)$.*

We also say that algorithm A identifies C efficiently in the limit. We now show that, in general, the class of DTAs cannot be identified efficiently in the limit. The reason for this is that there exists no polynomial q that bounds the size of a characteristic set for every DTA language.

3.2 Polynomial distinguishability of DTAs

In this section, we prove many results regarding the length of timed strings in DTA languages. In particular, we focus on the properties of polynomial reachability (Definition 3.5) and polynomial distinguishability (Definition 3.7). These two properties are of key importance to identification problems because they can be used to determine whether there exist polynomial bounds on the lengths of the strings that are necessary to converge to a correct model.

3.2.1 Not all DTAs are efficiently identifiable in the limit

The class of DTAs is not efficiently identifiable in the limit from labeled data. The reason is that in order to reach some parts of a DTA, one may need a timed string of exponential length. We give an example of this in Figure 3.2. Formally, this example can be used to show that in general DTAs are not *polynomially reachable*:

Definition 3.5. *(polynomial reachability) We call a class of automata C* polynomially reachable *if there exists a polynomial function p, such that for any reachable state q from any automaton $\mathcal{A} \in C$, there exists a string τ, with $|\tau| \leq p(|\mathcal{A}|)$, such that τ reaches q in $\mathcal{A}$.*

Proposition 3.6. *The class of DTAs is not polynomially reachable.*

Proof. Let $C^* = \{\mathcal{A}_n \mid n \geq 1\}$ denote the (infinite) class of DTAs defined by Fig. 3.2. In any DTA $\mathcal{A}_n \in C^*$, state q_3 can be reached only if both $x \geq 2^n$ and $y \leq 1$ are satisfied. Moreover, $x \leq 1$ is satisfied when y is reset for the first time, and later y can be reset only if $y \leq 1$ is satisfied. Therefore, in order to satisfy both $y \leq 1$ and $x \geq 2^n$, y has to be reset 2^n times. Hence, the shortest string τ that reaches state q_3 is of length 2^n. However, since the clock guards are encoded in binary, the size of $\mathcal{A}_n$ is only polynomial in n. Thus, there exists no polynomial function p such that $\tau \leq p(|\mathcal{A}_n|)$. Since every $\mathcal{A}_n \in C^*$ is a DTA, DTAs are not polynomially reachable. □

The non-polynomial reachability of DTAs implies non-polynomial distinguishability of DTAs:

Definition 3.7. *(polynomial distinguishability) We call a class of automata C* polynomially distinguishable *if there exists a polynomial function p, such that for any two automata $\mathcal{A}, \mathcal{A}' \in C$ such that $L(\mathcal{A}) \neq L(\mathcal{A}')$, there exists a string $\tau \in L(\mathcal{A}) \bigtriangleup L(\mathcal{A}')$, such that $|\tau| \leq p(|\mathcal{A}| + |\mathcal{A}'|)$.*

Proposition 3.8. *The class of DTAs is not polynomially distinguishable.*

Proof. DTAs are not polynomially reachable, hence there exists no polynomial function p such that for every state q of any DTA $\mathcal{A}$, the length of a shortest timed string τ that reaches q in $\mathcal{A}$ is bounded by $p(|\mathcal{A}|)$. Thus, there is a DTA $\mathcal{A}$ with a state q for which the length of τ cannot be polynomially bounded by $p(|\mathcal{A}|)$. Given this DTA $\mathcal{A} = \langle Q, X, \Sigma, \Delta, q_0, F\rangle$, construct two DTAs $\mathcal{A}_1 = \langle Q, X, \Sigma, \Delta, q_0, \{q\}\rangle$ and $\mathcal{A}_2 = \langle Q, X, \Sigma, \Delta, q_0, \emptyset\rangle$. By construction, τ is the shortest string in $L(\mathcal{A}_1)$, and $\mathcal{A}_2$ accepts the empty language. Therefore, τ is the shortest string such that $\tau \in L(\mathcal{A}_1) \bigtriangleup L(\mathcal{A}_2)$. Since $|\mathcal{A}_1| + |\mathcal{A}_2| \leq 2 \times |\mathcal{A}|$, there exists no polynomial function p such that the length of τ is bounded by $p(|\mathcal{A}_1| + |\mathcal{A}_2|)$. Hence the class of DTAs is not polynomially distinguishable. □

It is fairly straightforward to show that polynomial distinguishability is a necessary requirement for efficient identifiability:

Lemma 3.9. *If a class of automata C is efficiently identifiable, then C is polynomially distinguishable.*

Proof. Suppose a class of automata C is efficiently identifiable, but not polynomially distinguishable. Thus, there exists no polynomial function p such that for any two automata $\mathcal{A}, \mathcal{A}' \in C$ (with $L(\mathcal{A}) \neq L(\mathcal{A}')$) the length of the shortest timed string $\tau \in L(\mathcal{A}) \bigtriangleup L(\mathcal{A}')$ is bounded by $p(|\mathcal{A}| + |\mathcal{A}'|)$. Let $\mathcal{A}$ and $\mathcal{A}'$ be two automata for which such a function p does not exist and let S_{cs} and S'_{cs} be their polynomial characteristic sets. Let $S = S_{cs} \cup S'_{cs}$ be the input sample for the identification algorithm A for C from Definition 3.4. Since C is not polynomially distinguishable, neither S_{cs} nor S'_{cs} contains a timed string τ such that $\tau \in L(\mathcal{A})$ and $\tau \notin L(\mathcal{A}')$, or vice versa (because no distinguishing string is of polynomial length). Hence, $S = (S_+, S_-)$ is such that $S_+ \subseteq L(\mathcal{A})$, $S_+ \subseteq L(\mathcal{A}')$, $S_- \subseteq L(\mathcal{A})^c$, and $S_- \subseteq L(\mathcal{A}')^c$. The second requirement of Definition 3.3 now requires that A returns both $\mathcal{A}$ and $\mathcal{A}'$, a contradiction. □

Thus, in order to efficiently identify a DTA, we need to be able to distinguish it from every other DTA or vice versa, using a timed string of polynomial length. Combining this with the fact that DTAs are not polynomially distinguishable leads to the main result of this section:

Theorem 1. *The class of DTAs cannot be identified efficiently.*

Proof. By Proposition 3.8 and Lemma 3.9. □

Or more specifically:

Corollary 3.10. *The class of DTAs with two or more clocks cannot be identified efficiently.*

Proof. The corollary follows from the fact that the argument of Proposition 3.6 requires a DTA with at least two clocks. □

An interesting alternative way of proving a slightly weaker non-efficient identifiability result for DTAs is by linking the efficient identifiability property to the *equivalence problem* for automata:

Definition 3.11. *(equivalence problem) The* equivalence problem *for a class of automata C is to find the answer to the following question: Given two automata $\mathcal{A}, \mathcal{A}' \in C$, is it the case that $L(\mathcal{A}) = L(\mathcal{A}')$?*

Lemma 3.12. *If a class of automata C is identifiable from polynomial data, and if membership of a C-language is decidable in polynomial time, then the equivalence problem for C is in* coNP.

Proof. The proof follows from the definition of polynomial distinguishability. Suppose that C is polynomially distinguishable. Thus, there exists a polynomial function p such that for any two automata $\mathcal{A}, \mathcal{A}' \in C$ (with $L(\mathcal{A}) \neq L(\mathcal{A}')$) the length of a shortest timed string $\tau \in L(\mathcal{A}) \triangle L(\mathcal{A}')$ is bounded by $p(|\mathcal{A}| + |\mathcal{A}'|)$. This example τ can be used as a certificate for a polynomial time falsification algorithm for the equivalence problem: On input $((\mathcal{A}, \mathcal{A}'))$:

1. Guess a certificate τ such that $|\tau| \leq p(|\mathcal{A}| + |\mathcal{A}'|)$.

2. Test whether τ is a positive example of $\mathcal{A}$.

3. Test whether τ is a positive example of $\mathcal{A}'$.

4. If the tests return different values, return reject.

The algorithm rejects if there exists a polynomial length certificate $\tau \in L(\mathcal{A}) \triangle L(\mathcal{A}')$. Since C is polynomially distinguishable, this implies that the equivalence problem for C is in coNP. □

The equivalence problem has been well-studied for many systems, including DTAs. For DTAs it has been proven to be PSPACE-complete (Alur and Dill 1994), hence:

Theorem 2. *If* $\mathsf{coNP} \neq \mathsf{PSPACE}$*, then DTAs cannot be identified efficiently.*

Proof. By Lemma 3.12 and the fact that equivalence is PSPACE-complete for DTAs (Alur and Dill 1994). □

Using a similar argument we can also show a link with the *reachability problem*:

Definition 3.13. *(reachability problem) The* reachability problem *for a class of automata C is to find the answer to the following question: Given an automaton $\mathcal{A} \in C$ and a state q of $\mathcal{A}$, does there exist a string that reaches q in $\mathcal{A}$?*

Lemma 3.14. *If a class of models C is identifiable from polynomial data then the reachability problem for C is in* NP*.*

Proof. Similar to the proof of Lemma 3.12. But now we know that for each state there has to be an example of polynomial length in the size of the target automaton. This example can be used as a certificate by a polynomial-time algorithm for the reachability problem. □

Theorem 3. *If* $\mathsf{NP} \neq \mathsf{PSPACE}$*, then DTAs cannot be identified efficiently.*

Proof. By Lemma 3.14 and the fact that reachability is PSPACE-complete for DTAs (Alur and Dill 1994). □

These results seem to shatter all hope of ever finding an efficient algorithm for identifying DTAs. Instead of identifying general DTAs, we therefore focus on subclasses of DTAs that are efficiently identifiable.

3.2.2 DTAs with a single clock are polynomially distinguishable

In the previous section we showed DTAs not to be efficiently identifiable in general. The proof for this is based on the fact that DTAs are not polynomially distinguishable. Since polynomial distinguishability is a necessary requirement for efficient identifiability, we are interested in classes of DTAs that *are* polynomially distinguishable. In this section, we show that DTAs with a single clock are polynomially distinguishable.

A one-clock DTA (1-DTA) is a DTA that contains exactly one clock, i.e., $|X| = 1$. Our proof that 1-DTAs are polynomially distinguishable is based on the following observation:

- If a timed string τ reaches some timed state (q, v) in a 1-DTA $\mathcal{A}$, then for all v' such that $v'(x) \geq v(x)$, the timed state (q, v') can be reached in $\mathcal{A}$.

This holds because when a timed string reaches (q, v) it could have made a larger time transition to reach all larger valuations. This property is specific to 1-DTAs: a DTA with multiple clocks can wait in q, but only bigger valuations can be reached where the difference between the clocks remains the same. It is this property of 1-DTAs that allows us to polynomially bound the length of a timed string that distinguishes between two 1-DTAs. We first use this property to show that 1-DTAs are polynomially reachable. We then use a similar argument to show the polynomial distinguishability of 1-DTAs.

Proposition 3.15. *1-DTAs are polynomially reachable.*

Proof. Given a 1-DTA $\mathcal{A} = \langle Q, \{x\}, \Sigma, \Delta, q_0, F\rangle$, let $\tau = (a_1, t_1) \dots (a_n, t_n)$ be a shortest timed string such that τ reaches some state $q_n \in Q$. Suppose that some prefix $\tau_i = (a_1, t_1) \dots (a_i, t_i)$ of τ ends in some timed state (q, v). Then for any $j > i$, τ_j cannot end in (q, v') if $v(x) \leq v'(x)$. If this were the case, τ_i instead of τ_j could be used to reach (q, v'), and hence a shorter timed string could be used to reach q_n, resulting in a contradiction. Thus, for some index $j > i$, if τ_j also ends in q, then there exists some index $i < k \leq j$ and a state $q' \neq q$ such that τ_k ends in (q', v_0), where $v_0(x) = 0$. In other words, x has to be reset between index i and j in τ. In particular, if x is reset at index i (τ_i ends in (q, v_0)), there cannot exist any index $j > i$ such that τ_j ends in q. Hence:

- For every state $q \in Q$, the number of prefixes of τ that end in q is bounded by the amount of times x is reset by τ.
- For every state $q' \in Q$, there exists at most one index i such that τ_i ends in (q', v_0). In other words, x is reset by τ at most $|Q|$ times.

Consequently, each state is visited at most $|Q|$ times by the computation of $\mathcal{A}$ on τ. Thus, the length of τ is bounded by $|Q| * |Q|$, which is polynomial in the size of $\mathcal{A}$. □

Given that 1-DTAs are polynomially reachable, one would guess that it should be easy to prove the polynomial distinguishability of 1-DTAs. But this turns out to be a lot more complicated. The main problem is that when considering the difference between two 1-DTAs, we effectively have access to two clocks instead of one. Note that, although we have access to two clocks, there are no clock guards that bound both clock values. Because of this, we cannot construct DTAs such as the one in Figure 3.2. Our proof for the polynomial distinguishability of 1-DTAs follows the same line of reasoning as our proof of Proposition 3.15, although it is much more complicated to bound the amount of times that x is reset. We have split the proof of this bound into several proofs of smaller propositions and lemmas, the main theorem follows from combining these.

For the remainder of this section, let $\mathcal{A}_1 = \langle Q_1, \{x_1\}, \Sigma_1, \Delta_1, q_{1,0}, F_1\rangle$ and $\mathcal{A}_2 = \langle Q_2, \{x_2\}, \Sigma_2, \Delta_2, q_{2,0}, F_2\rangle$ be two 1-DTAs. Let $\tau = (a_1, t_1) \dots (a_n, t_n)$ be a shortest string that distinguishes between these 1-DTAs, formally:

Definition 3.16. *(shortest distinguishing string) A* shortest distinguishing string *τ of two DTAs $\mathcal{A}_1$ and $\mathcal{A}_2$ is a minimal length timed string such that $\tau \in L(\mathcal{A}_1)$ and $\tau \notin L(\mathcal{A}_2)$, or vice versa.*

We combine the computations of $\mathcal{A}_1$ and $\mathcal{A}_2$ on this string τ into a single computation sequence:

Definition 3.17. *(combined computation) The* combined computation *of $\mathcal{A}_1$ and $\mathcal{A}_2$ over τ is the sequence:*

$$\begin{aligned}\langle q_{1,0}, q_{2,0}, v_0\rangle &\xrightarrow{t_1} \langle q_{1,0}, q_{2,0}, v_0 + t_1\rangle \ldots\\ \ldots\langle q_{1,n-1}, q_{2,n-1}, v_{n-1} + t_n\rangle &\xrightarrow{a_n} \langle q_{1,n}, q_{2,n}, v_n\rangle\end{aligned}$$

where for all $0 \leq i \leq n$, v_i is a valuation function for both x_1 and x_2, and

$$(q_{1,0}, v_0) \xrightarrow{t_1} (q_{1,0}, v_0 + t_1)\ldots(q_{1,n-1}, v_{n-1} + t_n) \xrightarrow{a_n} (q_{1,n}, v_n)$$

is the computation of $\mathcal{A}_1$ over τ, and

$$(q_{2,0}, v_0) \xrightarrow{t_1} (q_{2,0}, v_0 + t_1)\ldots(q_{2,n-1}, v_{n-1} + t_n) \xrightarrow{a_n} (q_{2,n}, v_n)$$

is the computation of $\mathcal{A}_2$ over τ.

All the definitions of properties of computations are easily adapted to properties of combined computations. For instance, (q_1, q_2) is called a *combined state* and $\langle q_1, q_2, v\rangle$ is called a *combined timed state*. By using the properties of τ its combined computation, we now show the following:

Proposition 3.18. *The length of τ is bounded by a polynomial in the size of $\mathcal{A}_1$, the size of $\mathcal{A}_2$, and the sum of the amount of times x_1 and x_2 are reset by τ.*

Proof. Suppose that for some index $1 \leq i \leq n$, τ_i ends in $\langle q_1, q_2, v\rangle$. Using the same argument used in the proof of Proposition 3.15, one can show that for every $j > i$ and for some v', if τ_j ends in $\langle q_1, q_2, v'\rangle$, then there exists an index $i < k \leq j$ such that τ_k ends in (q'_1, v_0) in $\mathcal{A}_1$ for some $q'_1 \in Q_1$, or in (q'_2, v_0) in $\mathcal{A}_2$ for some $q'_2 \in Q_2$. Thus, for every combined state $(q_1, q_2) \in Q_1 \times Q_2$, the number of prefixes of τ that end in (q_1, q_2) is bounded by the sum of the amount of times r that x_1 or x_2 has been reset by τ. Hence the length of τ is bounded by $|Q_1| * |Q_2| * r$, which is polynomial in r and in the sizes of $\mathcal{A}_1$ and $\mathcal{A}_2$. □

We want to bound the number of clock resets in the combined computation of a shortest distinguishing string τ. In order to do so, we first prove a restriction on the possible clock valuations in a combined state (q_1, q_2) that is reached directly after one clock x_1 has been reset. In Proposition 3.15, there was exactly one possible valuation, namely $v(x) = 0$. But since we now have an additional clock x_2, this restriction no longer holds. We can show, however, that after the second time (q_1, q_2) is reached by τ directly after resetting x_1, the valuation of x_2 has to be smaller than the previous times τ reached (q_1, q_2):

Lemma 3.19. *If there exist (at least) three indexes $1 \leq i < j < k \leq n$ such that τ_i, τ_j, and τ_k all end in (q_1, q_2) directly after a reset of x_1 ($v_i(x_1) = v_j(x_1) = v_k(x_1) = v_0(x_1)$), then the valuation of x_2 at index k has to be smaller than the previous valuations of x_2 at indexes i and j, i.e., $v_i(x_2) > v_k(x_2)$ and $v_j(x_2) > v_k(x_2)$.*

Proof. Without loss of generality we assume that $\tau \in L(\mathcal{A}_1)$, and consequently $\tau \notin L(\mathcal{A}_2)$. Let $\tau_{-k} = (a_{k+2}, t_{k+2})\ldots(a_n, t_n)$ denote the suffix of τ starting at index $k + 2$. Let $a = a_{k+1}$ and $t = t_{k+1}$. Thus, $\tau = \tau_k(a, t)\tau_{-k}$.

We prove the lemma by contradiction. Assume that the valuations v_i, v_j, and v_k are such that $v_i(x_2) < v_j(x_2) < v_k(x_2)$. The argument below can be repeated for the case when $v_j(x_2) < v_i(x_2) < v_k(x_2)$. Let d_1 and d_2 denote the differences in clock values of x_2 between the first and second, and second and third time that (q_1, q_2) is reached by τ, i.e., $d_1 = v_j(x_2) - v_i(x_2)$ and $d_2 = v_k(x_2) - v_j(x_2)$.

We are now going to make some observations about the acceptance property of the computations of $\mathcal{A}_1$ and $\mathcal{A}_2$ over τ. However, instead of following the path specified by τ, we are going to perform a time transition in (q_1, q_2) and then only compute the final part $\tau{-}k$ of τ. Because we assume that (q_1, q_2) is reached (at least) three times, and each time the valuation of x_2 is larger, we can in this way reach the timed state (q_2, v_k) in $\mathcal{A}_2$. Because we reach the same timed state (q_2, v_k) as τ, and because the subsequent timed string $\tau{-}k$ is identical, the acceptance property has to remain the same. We know that $\tau = \tau_k(a,t)\tau_{-k} \notin L(\mathcal{A}_2)$. Hence, it has to hold that $\tau_j(a, t+d_2)\tau_{-k} \notin L(\mathcal{A}_2)$ and that $\tau_i(a, t+d_1+d_2)\tau_{-k} \notin L(\mathcal{A}_2)$. Similarly, since $\tau \in L(\mathcal{A}_1)$, and since τ_i, τ_j, and τ_k all end in the same timed state (q_1, v_0) in $\mathcal{A}_1$, it holds that $\tau_i(a,t)\tau_{-k} \in L(\mathcal{A}_1)$ and that $\tau_j(a,t)\tau_{-k} \in L(\mathcal{A}_1)$. Let us put this type of information in a table (+ denotes true, and − denotes false):

value of d	0	d_1	d_2	$(d_1 + d_2)$
$\tau_i(a, t+d)\tau_{-k} \in L(\mathcal{A}_1)$	+			
$\tau_j(a, t+d)\tau_{-k} \in L(\mathcal{A}_1)$	+			
$\tau_i(a, t+d)\tau_{-k} \in L(\mathcal{A}_2)$				−
$\tau_j(a, t+d)\tau_{-k} \in L(\mathcal{A}_2)$			−	

Since τ is a shortest distinguishing string, it cannot be the case that both $\tau_i(a, t+d)\tau_{-k} \in L(\mathcal{A}_1)$ and $\tau_i(a, t+d)\tau_{-k} \notin L(\mathcal{A}_2)$ (or vice versa) hold for any $d \in \mathbb{N}$. Otherwise, $\tau_i(a, t+d)\tau_{-k}$ would be a shorter distinguishing string for $\mathcal{A}_1$ and $\mathcal{A}_2$. This also holds if we replace i by j, resulting in the following table:

value of d	0	d_1	d_2	$(d_1 + d_2)$
$\tau_i(a, t+d)\tau_{-k} \in L(\mathcal{A}_1)$	+			−
$\tau_j(a, t+d)\tau_{-k} \in L(\mathcal{A}_1)$	+		−	
$\tau_i(a, t+d)\tau_{-k} \in L(\mathcal{A}_2)$	+			−
$\tau_j(a, t+d)\tau_{-k} \in L(\mathcal{A}_2)$	+		−	

Furthermore, since τ_i ends in the same timed state as τ_j in $\mathcal{A}_1$ (directly after a reset of x_1), it holds that for all $d \in \mathbb{N}$: $\tau_i(a, t+d)\tau_{-k} \in L(\mathcal{A}_1)$ if and only if $\tau_j(a, t+d)\tau_{-k} \in L(\mathcal{A}_1)$. The table thus becomes:

value of d	0	d_1	d_2	$(d_1 + d_2)$
$\tau_i(a, t+d)\tau_{-k} \in L(\mathcal{A}_1)$	+		−	−
$\tau_j(a, t+d)\tau_{-k} \in L(\mathcal{A}_1)$	+		−	−
$\tau_i(a, t+d)\tau_{-k} \in L(\mathcal{A}_2)$	+			−
$\tau_j(a, t+d)\tau_{-k} \in L(\mathcal{A}_2)$	+		−	

By performing the previous step again, we obtain:

value of d	0	d_1	d_2	$(d_1 + d_2)$
$\tau_i(a, t + d)\tau_{-k} \in L(\mathcal{A}_1)$	$+$		$-$	$-$
$\tau_j(a, t + d)\tau_{-k} \in L(\mathcal{A}_1)$	$+$		$-$	$-$
$\tau_i(a, t + d)\tau_{-k} \in L(\mathcal{A}_2)$	$+$		$-$	$-$
$\tau_j(a, t + d)\tau_{-k} \in L(\mathcal{A}_2)$	$+$		$-$	$-$

Now, since $\tau_i(a, t+d_1)$ ends in the same timed state as $\tau_j(a, t)$ in $\mathcal{A}_2$, it holds that $\tau_i(a, t + d_1)\tau_{-k} \in L(\mathcal{A}_2)$ if and only if $\tau_j(a, t)\tau_{-k} \in L(\mathcal{A}_2)$ (since they reach the same timed state and then their subsequent computations are identical). Combining this with the previous steps results in:

value of d	0	d_1	d_2	$(d_1 + d_2)$
$\tau_i(a, t + d)\tau_{-k} \in L(\mathcal{A}_1)$	$+$	$+$	$-$	$-$
$\tau_j(a, t + d)\tau_{-k} \in L(\mathcal{A}_1)$	$+$	$+$	$-$	$-$
$\tau_i(a, t + d)\tau_{-k} \in L(\mathcal{A}_2)$	$+$	$+$	$-$	$-$
$\tau_j(a, t + d)\tau_{-k} \in L(\mathcal{A}_2)$	$+$	$+$	$-$	$-$

More generally, it holds that for any time value $d \in \mathbb{N}$, $\tau_i(a, t+d_1+d)\tau_{-k} \in L(\mathcal{A}_2)$ if and only if $\tau_j(a, t + d)\tau_{-k} \in L(\mathcal{A}_2)$. Hence, we can extend the table in the following way:

value of d	$\ldots$	$(d_1 + d_2)$	$2d_1$	$(2d_1 + d_2)$
$\tau_i(a, t + d)\tau_{-k} \in L(\mathcal{A}_1)$	$\ldots$	$-$	$+$	$-$
$\tau_j(a, t + d)\tau_{-k} \in L(\mathcal{A}_1)$	$\ldots$	$-$	$+$	$-$
$\tau_i(a, t + d)\tau_{-k} \in L(\mathcal{A}_2)$	$\ldots$	$-$	$+$	$-$
$\tau_j(a, t + d)\tau_{-k} \in L(\mathcal{A}_2)$	$\ldots$	$-$	$+$	$-$

This extension can be continued infinitely. Thus, for any value $m \in \mathbb{N}$ it holds that $\tau_i(a, t + m * d_1)\tau_{-k} \in L(\mathcal{A}_2)$ and $\tau_i(a, t + m * d_1 + d_2)\tau_{-k} \notin L(\mathcal{A}_2)$. This can only be the case if a different transition is fired for each of these $|\mathbb{N}|$ different values for m. Consequently, $\mathcal{A}_2$ should contain an infinite amount of transitions, and hence $\mathcal{A}_2$ is not a 1-DTA, a contradiction.

We obtain the same proof for the case where $\tau \in L(\mathcal{A}_2)$ by flipping all $+$ symbols to $-$ symbols in the tables and vice versa. The proof for the case where $v_j(x_2) < v_i(x_2) < v_k(x_2)$ can be constructed by modifying the definitions of d_1 and d_2, i.e., $d_1 = v_i(x_2) - v_j(x_2)$ and $d_2 = v_k(x_2) - v_i(x_2)$. □

We have just shown that if τ reaches some combined state (q_1, q_2) twice directly after a reset of x_1, then the valuation of x_2 has to be *decreasing* with respect to the previous time it reached (q_1, q_2). Without loss of generality we can assume that x_1 has already been reset at least twice at previous indexes just before reaching (q_1, q_2).[3] This observation can be used to show that if τ reaches (q_1, q_2) again then:

- x_2 is reset before again reaching (q_1, q_2) and resetting x_1, and
- on the path from (q_1, q_2) to (q_1, q_2), there has to exist at least one transition that cannot be satisfied by a valuation smaller than the one reached by τ.

[3] Reaching every combined state two times extra does of course not influence the polynomial complexity.

The first statement follows from the observation that the valuation of x_2 has to be decreasing. The second statement holds because if there is no such transition, then a timed string τ' exists that reaches a smaller valuation of x_2 than τ when it reaches (q_1, q_2) again. By the argument of Lemma 3.19, it cannot be the case that a non-shortest distinguishing string reaches a smaller valuation than τ. Hence this either leads to a contradiction or τ' is a shortest distinguishing string. In this case the statement holds for the shortest distinguishing string τ'. Formally:

Corollary 3.20. *If for some index i, τ_i ends in (q_1, q_2) directly after a reset of x_1, and if there exists another index $j > i$ such that τ_j ends in (q_1, q_2) directly after a reset of x_1, then there exists an index $i < k \leq j$ such that τ_k ends in $(q_{2,k}, v_{2,0})$ in $\mathcal{A}_2$.*

Proof. By Lemma 3.19, it has to hold that $v_{2,i}(x_2) > v_{2,j}(x_2)$. The value of x_2 can only decrease if it has been reset. Hence there has to exist an index $i < k \leq j$ at which x_2 is reset. □

Corollary 3.21. *If for some index i, τ_i ends in (q_1, q_2) directly after a reset of x_1, and if there exists another index $j > i$ such that τ_j ends in (q_1, q_2) directly after a reset of x_1, then there exists an index $i < l \leq j$ such that τ_l ends in $\langle q_{1,l}, q_{2,l}, v_l \rangle$, where either $v_l(x_1)$ or $v_l(x_2)$ is the lower bound of the clock guard of the last transition fired by τ_l in $\mathcal{A}_1$ or $\mathcal{A}_2$, respectively.*

Proof. Suppose there exists no such index l. In this case, we can subtract 1 from a time value occurring in τ_j to create a timed string τ'_j that follows the same path as τ_j, but ends in $\langle q_1, q_2, v'_j \rangle$, where $v'_j(x_1) = 0$ and $v'_j(x_2) = v_j(x_2) - 1$. Without loss of generality τ'_j is not the prefix of a shortest distinguishing string (otherwise the corollary holds for τ'_j instead of τ_j). We know that τ'_j reaches a smaller valuation than τ_l, i.e., it holds that $v_j(x_2) < v_j(x_2)$. Hence, τ'_j can be used to reach a contradiction using the argument of Lemma 3.19. □

We now use these these two properties of τ to polynomially bound the number of different ways in which x_2 can be reset by τ before reaching (q_1, q_2). By Corollary 3.20, this also polynomially bounds the amount of resets of x_1. In combination with Proposition 3.18 this proves that 1-DTAs are polynomially distinguishable.

Lemma 3.22. *The number of times x_2 is reset by τ before reaching a combined state (q_1, q_2) directly after a reset of x_1 is bounded by a polynomial in the sizes of $\mathcal{A}_1$ and $\mathcal{A}_2$.*

Proof. Suppose x_1 is reset at index $1 \leq i \leq n$ just before reaching (q_1, q_2), i.e., τ_i ends in $\langle q_1, q_2, v_i \rangle$, with $v_i(x_1) = 0$. Thus, by Lemma 3.19, $v_i(x_2)$ is decreasing with respect to previous indexes. By Corollary 3.20, we know that x_2 has reset before index i. Let $k < i$ be the largest index before i where x_2 is reset. Let (q'_1, q'_2) be the combined state that is reached directly after this reset. By Lemma 3.19, we know that $v_k(x_1)$ is decreasing with respect to previous indexes. We also know, by Corollary 3.21, that there exists an index l such that the last transition that is fired τ_l in either $\mathcal{A}_1$ or $\mathcal{A}_2$ has a clock guard g_1 or g_2 with a lower bound equal to $v_l(x_1)$ or $v_l(x_2)$, respectively. This situation is depicted in Figure 3.3. Let us consider these two cases.

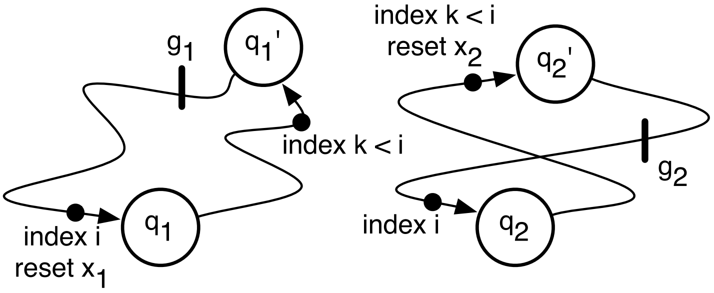


Figure 3.3: Bounding the number of resets of x_2 before a reset of x_1. The left and right figure represent loops followed by τ in $\mathcal{A}_1$ and $\mathcal{A}_2$ from the combined state (q_1, q_2) to the combined state (q'_1, q'_2) and back again. Clock x_1 is reset at index i, directly before entering (q_1, q_2). Clock x_2 is reset at index $k < i$, directly before entering (q'_1, q'_2). Because of this, there exists a clock guard g_1 or g_2 on the computation path from (q'_1, q'_2) to (q_1, q_2) that cannot be satisfied if this loop is performed again (the proof for the path from (q_1, q_2) to (q'_1, q'_2) follows by symmetry).

Suppose $v_l(x_1)$ is the lower bound of g_1. Since $v_k(x_1)$ is decreasing, and since x_1 is not reset between index k and l, it has to be the case that $v_l(x_1)$ is decreasing. Hence, if at later indexes $i < k' < l'$, it again occurs that x_1 is reset at index l' just before reaching (q_1, q_2), k' is the largest index before l' where x_2 is reset, and $\tau_{k'}$ ends in the same combined state as τ_k, then there can be no index l'' such that $v_{l''}$ satisfies g_1. Thus, if the same combined state (q'_1, q'_2) is used to reset x_2 before later reaching the same combined state (q_1, q_2) to reset x_1, there exists at least one transition in $\mathcal{A}_1$ that can no longer be fired on the path from (q'_1, q'_2) to (q_1, q_2). Hence, there are at most $|Q_1| * |Q_2| * |\Delta_1|$ ways in which this can occur in τ.

Alternatively, suppose $v_l(x_2)$ is the lower bound of g_2. Since $v_i(x_2)$ is decreasing, and since x_2 is not reset between index l and i, it has to be the case that $v_l(x_2)$ is decreasing. Hence if at some later index $i < i'$ it occurs again that x_1 is reset at index i' just before reaching (q_1, q_2), then there can be no index l' such that $v_{l'}$ satisfies g_2. Hence, there are at most $|\Delta_2|$ ways in which this can occur.

In conclusion, the number of times that x_2 can be reset by τ before reaching (q_1, q_2) directly after a reset of x_1 is bounded by $|Q_1| * |Q_2| * |\Delta_1| + |\Delta_2|$, which is polynomial in the sizes of $\mathcal{A}_1$ and $\mathcal{A}_2$. □

We are now ready to show the main result of this section:

Theorem 4. *1-DTAs are polynomially distinguishable.*

Proof. By Lemma 3.19, after the second time a combined state (q_1, q_2) is reached by τ directly after resetting x_1, it can only be reached again if x_2 is reset. By Lemma 3.22, the total number of different ways in which x_2 can be reset before reaching (q_1, q_2) and resetting x_1 is bounded by a polynomial p in $|\mathcal{A}_1| + |\mathcal{A}_2|$. Hence the total number of times a combined state (q_1, q_2) can be reached by τ directly after resetting x_1 is bounded by $|Q_1| * |Q_2| * p(|\mathcal{A}_1| + |\mathcal{A}_2|)$. This is polynomial in $|\mathcal{A}_1|$ and $|\mathcal{A}_2|$. By symmetry, this also holds for combined states that are reached directly after resetting x_2. Hence, the total number of resets of x_1 and x_2 by τ is bounded by a polynomial in $|\mathcal{A}_1| + |\mathcal{A}_2|$. Since, by Proposition 3.18, the length of τ is bounded by this number, 1-DTAs are polynomially distinguishable. □

As a bonus we get the following corollaries:

Corollary 3.23. *The equivalence problem for 1-DTAs is in* coNP.

Proof. By Theorem 4 and the same argument as Lemma 3.12. □

Corollary 3.24. *The reachability problem for 1-DTAs is in* NP.

Proof. By Proposition 3.15 and the same argument as Lemma 3.14. □

3.3 DTAs with a single clock are efficiently identifiable

In the preceding section, we proved that DTAs in general cannot be identified efficiently because they are not polynomially distinguishable. In addition, we showed that 1-DTAs are polynomially distinguishable, and hence that they might be efficiently identifiable. In this section we show this indeed to be the case: 1-DTAs are efficiently identifiable. We prove this by showing an algorithm that identifies 1-DTAs efficiently in the limit. In other words, we describe a polynomial time algorithm ID_1-DTA (Algorithm 3.1) for the identification of 1-DTAs and we show that there exists polynomial characteristic sets (Lemma 3.26 and Proposition 3.27) for this algorithm. In order to bound the length of these timed strings (and the size of S_{cs}) we make use of the facts that 1-DTAs are both polynomially reachable (Proposition 3.15) and polynomially distinguishable (Theorem 4). The combination of these results satisfies all the constraints required for efficient identification in the limit (Definition 3.4), and hence shows that 1-DTAs are efficiently identifiable (Theorem 5).

3.3.1 An algorithm for identifying 1-DTAs efficiently

In this section, we describe our ID_1-DTA algorithm for identifying 1-DTAs from an input sample S. The main value of this algorithm is that:

- given any input sample S, ID_1-DTA returns in polynomial time a 1-DTA $\mathcal{A}$ that is consistent with S, i.e., such that $S_+ \subseteq L(\mathcal{A})$ and $S_- \subseteq L(\mathcal{A})^{\mathrm{c}}$,

- and if S contains a characteristic subsample S_{cs} for some target language L_t, then ID_1-DTA returns a correct 1-DTA $\mathcal{A}$, i.e., such that $L(\mathcal{A}) = L_t$.

In other words, ID_1-DTA identifies 1-DTAs efficiently in the limit. Note that, in a 1-DTA identification problem, the size of the 1-DTA is not predetermined. Hence, our algorithm has to identify the complete structure of a 1-DTA, including states, transitions, clock guards, and resets. Our algorithm identifies this structure one transition at a time: it starts with an empty 1-DTA $\mathcal{A}$, and whenever an identified transition requires more states or additional transitions, these will be added to $\mathcal{A}$. In this way, ID_1-DTA builds the structure of $\mathcal{A}$ piece by piece. Since we claim that ID_1-DTA identifies 1-DTAs efficiently, i.e., from polynomial time and data, we require that, for any input sample S for any target language L_t, the following four properties hold for this identification process:

Property 1. Identifying a single transition δ requires time polynomial in the size of S *(polynomial time per δ)*.

Property 2. The number of such transitions is polynomial in the size of S *(convergence in polynomial time)*.

Property 3. For every transition δ, there exists an input sample S_{cs} of size polynomial in the size of the smallest 1-DTA for L_t such that when included in S, S_{cs} guarantees that δ is identified correctly *(polynomial data per δ)*.

Property 4. The number of such correct transition identifications that are required to return a 1-DTA $\mathcal{A}$ with $L(\mathcal{A}) = L_t$ is polynomial in the size of the smallest 1-DTA for L_t *(convergence from polynomial data)* .

With these four properties in mind, we develop our ID_1-DTA algorithm for the efficient identification of 1-DTAs. This algorithm is shown in Algorithm 3.1. In this section, we use an illustrative example to show how this algorithm identifies a single transition, and to give some intuition why the algorithm satisfies these four properties. In the next section, we prove that our algorithm indeed satisfies these four properties and thus prove that it identifies 1-DTAs efficiently in the limit.

Example 3.2. Suppose that after having identified a few transitions, our algorithm has constructed the (incomplete) 1-DTA $\mathcal{A}$ from Figure 3.4. Furthermore, suppose that S contains the following timed strings: $\{(a,4)(a,6),(a,5)(b,6),(b,3)(a,2),(a,4)(a,1)(a,3),(a,4)(a,2)(a,2)(b,3)\} \subseteq S_+$ and $\{(a,3)(a,10),(a,4)(a,2)(a,2),(a,4)(a,3)(a,2)(b,3),(a,5)(a,3)\} \subseteq S_-$. Our algorithm has to identify a new transition δ of $\mathcal{A}$ using information from S. There are a few possible identifiable transitions: state q_1 does not yet contain any transitions for b, or for a and valuations smaller than 9, and state q_2 does not yet contain any transitions at all. Our algorithm first chooses which transition to identify, i.e., it selects the source state, label, and valuations for a new transition. Then our algorithm actually identifies the transition, i.e., it uses S in order to determine the target state, clock guard, and reset of the transition.

As can be seen from the example, the first problem our algorithm has to deal with is to determine which transition to identify. Our algorithm makes this decision using a fixed predetermined order (independent of the input sample). The order used by our algorithm is very straightforward: first a state q is selected in the order of identification (first identified first), second a transition label l is selected

Algorithm 3.1 Efficiently learning 1-DTAs from polynomial data: ID_1-DTA

```
Require: An input sample $S = (S_+, S_-)$ for a language $L_t$, with alphabet $\Sigma$
Ensure: $\mathcal{A}$ is a 1-DTA consistent with $S$, i.e., $S_+ \subseteq L(\mathcal{A})$ and $S_- \subseteq L(\mathcal{A})^{\mathrm{c}}$, in addition,
  if it holds that $S_{cs} \subseteq S$, then $L(\mathcal{A}) = L_t$
  $\mathcal{A} := \langle Q = \{q_0\}, x, \Sigma, \Delta = \emptyset, q_0, F = \emptyset \rangle$
  if $S_+$ contains the empty timed string $\lambda$ then set $F := \{q_0\}$
  while there exist a reachable timed state $(q, v)$ and a symbol $a$ for which there exists
  no transition $\langle q, q', a, g, r \rangle \in \Delta$ such that $v$ satisfies $g$ do
    for all states $q \in Q$ and symbols $a \in \Sigma$ do
      $v_{\min} := \min\{v \mid (q, v)$ is reachable $\}$
      $c' := \max\{v \mid \neg\exists \langle q, q', a, g, r \rangle \in \Delta$ such that $v$ satisfies $g\}$
      while $v_{\min} \leq c'$ do
        create a new transition $\delta := \langle q, q' := 0, a, g := v_{\min} \leq x \leq c', r \rangle$, add $\delta$ to $\Delta$
        $V := \{v \mid \exists \tau \in S : \tau$ fires $\delta$ with valuation $v\}$
        $r := true$ and $c_1 :=$ lower_bound$(\delta, V \cup \{v_{\min}\}, \mathcal{A}, S)$
        $r := false$ and $c_2 :=$ lower_bound$(\delta, V \cup \{v_{\min}\}, \mathcal{A}, S)$
        if $c_1 \leq c_2$ then set $r := true$ and $g := c_1 \leq x \leq c'$
        else set $r :=$ false and $g := c_2 \leq x \leq c'$
        for every state $q'' \in Q$ (first identified first) do
          $q' := q''$
          if consistent$(\mathcal{A}, S)$ is $true$ then break else $q' := 0$
        end for
        if $q' = 0$ then
          create a new state $q''$, set $q' := q''$, and add $q'$ to $Q$
          if $\exists \tau \in S_+$ such that $\tau$ ends in $q'$ then set $F := F \cup \{q'\}$
        end if
        $c' := \min\{v \mid v$ satisfies $g\} - 1$
      end while
    end for
  end while
```

according to an alphabetic order, and third the highest possible upper bound c' for a clock guard in this state-label combination is chosen. This fixed order makes it easier to prove the existence of characteristic sets (satisfying property 3). In our example, our algorithm will try to identify a transition $\delta = \langle q, q', l, c \leq x \leq c', r \rangle$, where $q = q_1$, $l = a$, and $c' = 9$ (since there exists a transition with a clock guard that is satisfied by a valuation $v = 10$) are all fixed. Thus, our algorithm only needs to identify: (i) the target state q', (ii) the lower bound of the clock guard c, and (iii) the clock reset r.

Note that fixing q, a, and c' in this way does not influence which transitions will be identified by our algorithm. Since we need to identify a transition with these values anyway, it only influences the order in which these transitions are identified. We now show how our algorithm identifies c, r, and q'.

The lower bound c. Our algorithm first identifies the lower bound c of the clock guard g of δ. The smallest possible lower bound for g is the smallest reachable valuation $v_{\min}$ in q (q_1 in the example). This valuation $v_{\min}$ is equal to the smallest lower bound of a transition with q as target. In the example, $v_{\min}$ is 4. Thus, the

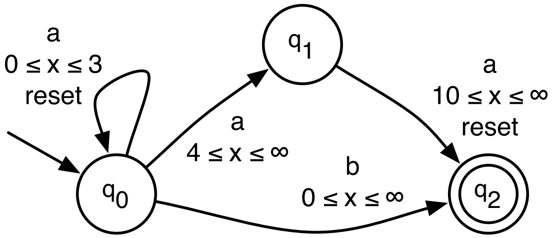


Figure 3.4: A partially identified 1-DTA. The transitions from state q_0 have been completely identified. State q_1 only has one outgoing transition. State q_2 has none.

Algorithm 3.2 Checking for consistency: consistent

```
Require: An 1-DTA $\mathcal{A}$ and an input sample $S$
Ensure: Returns true if $\mathcal{A}$ is consistent with $S$
  for every positive example $\tau = (a_1, t_1) \dots (a_n, t_n)$ from $S_+$ and
  every negative example $\tau' = (a'_1, t'_1) \dots (a'_m, t'_m)$ from $S_-$ and
  every pair of indices $1 \leq i \leq n$ and $1 \leq j \leq m$ do
    if $\tau_i$ ends in $(q, v)$ and $\tau'_j$ ends in $(q, v')$ and
    $\tau_{i+1} = \tau_i(a, t)$, $\tau'_{j+1} = \tau'_j(a, t + v - v')$ and
    $(a_{i+2}, t_{i+2}) \dots (a_n, t_n)$ equals $(a'_{j+2}, t'_{j+2})$ then
      Return false
    end if
  end for
  Return true
```

lower bound c has to be a value with the set $\{c \mid v_{\min} \leq c \leq c'\}$. One approach for finding c would be to try all possible values from this set and pick the best one. However, since time values are encoded in binary in the input sample S, iterating over such a set is exponential in the size of these time values, i.e., it is exponential in the size of S (contradicting property 1). This is why our algorithm only tries those time values that are actually used by timed strings from S. We determine these in the following way. We first set the lower bound of g to be $v_{\min}$. There are now examples in S that fire δ. The set of valuations V that these examples use to fire δ are all possible lower bounds for g, i.e., $V := \{v \mid \exists \tau \in S : \tau$ fires δ with valuation $v\}$. In our example, we have that $\{(a,4)(a,1)(a,3)\} \subseteq S_+$ and $\{(a,5)(a,3),(a,4)(a,2)(a,2)\} \subseteq S_-$. In this case, $V = \{4+1 = 5, 5+3 = 8, 4+2 = 6\}$. Since for every time value in V there exists at least one timed string in S for every such time value, iterating over this set is polynomial in the size of S (satisfying property 1).

From the set $V \cup \{v_{\min}\}$ our algorithm selects the smallest possible consistent lower bound. A lower bound is consistent if the 1-DTA resulting from identifying this bound is consistent with the input sample S. A 1-DTA $\mathcal{A}$ is called *consistent* if S contains no positive example that inevitably ends in the same state as a negative example, i.e., if the final result $\mathcal{A}$ can be such that $S_+ \in L(\mathcal{A})$ and $S_- \in L(\mathcal{A})^c$.

Whether $\mathcal{A}$ is consistent with S is checked by testing whether there exist no two timed strings $\tau \in S_+$ and $\tau' \in S_-$ that reach the same timed state (possibly after making a partial time transition) and afterwards their suffixes are identical. The algorithm for checking this is shown in Algorithm 3.2. This check can clearly be done in polynomial time (satisfying property 1). Our algorithm finds the smallest consistent lower bound by trying every possible lower bound $c \in V \cup \{v_{\min}\}$, and testing whether the result is consistent. This lower_bound routine is shown in Algorithm 3.3. This routine ensures that at least one timed string from S will fire δ, and hence that our algorithm only identifies a polynomial amount of transitions (satisfying property 2). In our example, setting c to 5 makes $\mathcal{A}$ inconsistent since now both $(a,4)(a,1)(a,3)$ and $(a,4)(a,2)(a,2)$ reach $(q',6)$, where q' is any possible target for δ, and afterwards they have the same suffix $(a,2)$. However, setting c to 6 does not make $\mathcal{A}$ inconsistent. Since 6 is the smallest value in $V \cup \{v_{\min}\}$ greater than 5, $c = 6$ is the smallest consistent lower bound for g.

Algorithm 3.3 Obtaining the lower bound: lower_bound

```
Require: A new transition $\delta = \langle q, q', a, g, R\rangle$
Require: A set of possible lower bounds $V \cup \{v_{\min}\}$ for the clock guard of a
  transition $\delta = \langle q, q', a, g = v_{\min} \leq x \leq c', r\rangle$, and a 1-DTA $\mathcal{A}$ that is consistent
  with an input sample $S$
Ensure: Returns the smallest consistent lower bound $v \geq v_{\min}$ for $\delta$
  $v := c'$
  for all valuations $v' \in V \cup \{v_{\min}\}$ do
    $g := v' \leq x \leq c'$
    if consistent($\mathcal{A}, S$) is true and $v' < v$ then set $v := v'$
  end for
  Set $g$ to its original value.
  Return $v$
```

Our main reason for selecting the smallest consistent lower bound for g is that this selection can be used to force our algorithm to make the correct identification (required by property 3). Suppose that if our algorithm identifies c^*, and if all other identifications are correct, then the result $\mathcal{A}$ will be such that $L(\mathcal{A}) = L_t$. Hence, our algorithm should identify c^*. In this case, there always exist examples that result in an inconsistency when our algorithm selects any valuation smaller than c^*. The reason is that an example that fires δ with valuation $c^* - 1$ should actually fire a different transition, to a different state, or with a different reset value. Hence, the languages after firing these transitions are different. Therefore, there would exist two timed strings $\tau \in L_t$ and $\tau' \in L_t^c$ (that can be included in S) that have identical suffixes after firing δ with valuations c^* and $c^* - 1$ respectively. Moreover, any pair of string that fire δ with valuations greater or equal to c^* cannot lead to an inconsistency since their languages after firing δ are the same.

The reset r. After having identified the lower bound c of the clock guard g of δ, our algorithm needs to identify the reset r of δ. One may notice that the identification of g depends on whether δ contains a clock reset or not: the value of r determines the valuations that are reached by timed strings after firing δ (the clock can be reset to 0), hence this value determines whether $\mathcal{A}$ is consistent after trying a particular lower bound for g. In our example, $(a,4)(a,1)(a,3)$ and $(a,4)(a,1)(a,2)$ reach $(q',1)$ and $(q',0)$ respectively before their suffixes are identical if $r = true$. Because of this, our algorithm identifies the clock reset r of δ at the same time it identifies the clock guard g. The method it uses to identify r is very simple: first set $r = true$ and then find the smallest consistent lower bound c_1 for g, then set $r = false$ and find another such lower bound c_2 for g. The value of r is set to *true* if and only if the lower bound found with this setting is smaller than the other one, i.e., iff $c_1 \leq c_2$. There always exist timed strings that ensure that the smallest consistent lower bound for g such that when the clock reset is set incorrectly, it is larger than when it is set correctly (satisfying property 3). In our example the timed strings that ensure this are $(a,4)(a,2)(a,2)(b,3) \in S_+$ and $(a,4)(a,3)(a,2)(b,3) \in S_-$. Because these examples reach the same valuations in state q' only if the clock is reset, they create an inconsistency when r is set to *true*. In general, such strings always exists since the difference of 1 time value is sufficient for such an inconsistency: a difference of 1 time value can always be the difference between later satisfying and not satisfying some clock guard.[4]

The target state q'. Having identified both the clock guard and the reset of δ, our algorithm still needs to identify the target state q' of δ. Since we need to make sure that our algorithm is capable of identifying any possible transition (required by property 3), we need to try all possible settings for q', and in order to make it easier to prove the existence of a characteristic set (required by property 3), we do so in a fixed order. The order our algorithm uses is the order in which our algorithm identified the states, i.e., first q_0, then the first additional identified state, then the second, and so on. The target state for δ is set to be the first consistent target state in this order. In our example, we just try state q_0, then state q_1, and finally state q_3. When none of the currently identified states result in a consistent 1-DTA $\mathcal{A}$, the target is set to be a new state. This new state is set to be a final state only if there exists a timed string in S_+ that ends in it. It should be clear that since the languages after reaching different states are different, there always exist timed strings that ensure that our algorithm identifies the correct target (satisfying property 3). In our example, there exist no timed strings that make $\mathcal{A}$ inconsistent when our algorithm tries the first state (state q_0), and hence our algorithm identifies a transition $\langle q_1, q_0, a, 6 \leq x \leq 9, \text{false}\rangle$.

This completes the identification of δ and (possibly) q'. This identification of a single transition δ essentially describes the main part of our algorithm (see Algorithm 3.1). However, we still have to explain how our algorithm iterates over the transitions it identifies. The algorithm consists of a main loop that iterates in a fixed order over the possible source states and labels for new transitions. For every combination of a source state q and a label a, our algorithm first sets two values: $v_{\min}$ and c'. The first is the smallest reachable valuation in q. The second is the fixed upper bound of the delay guard of a new transition. Because our model is

[4] This holds unless the clock guard can only be satisfied by a unique valuation, i.e., unless $g = c \leq x \leq c$. However, in this case any setting for r is correct since both can lead to results such that $L(\mathcal{A}) = L_t$.

deterministic, this is set to be the largest reachable valuation for which there exists no transition with q as source state and a as label. After identifying a transition δ with these values, our algorithm updates c' to be one less than the lower bound of the clock guard of δ. If c is still greater than $v_{\min}$, there are still transitions to identify for state q and label a. Thus, our algorithm iterates and continues this iteration until c' is strictly less than $v_{\min}$. Our main reason for adding this additional iteration is that it makes it easier to prove the convergence of our algorithm (property 4). The main loop of our algorithm continuously identifies new transitions and possibly new target states until there are no more new transitions to identify, i.e., until there exists a transition for every reachable timed state in $\mathcal{A}$. This is necessary because identifying a transition δ can create new identifiable transitions. This happens when the smallest reachable valuation $v_{\min}$ in some state is decreased, or when a new state is identified, by the identification of δ.

3.3.2 Polynomial characteristic sets for 1-DTAs

We described a polynomial-time algorithm for the identification of 1-DTAs. The algorithm is consistent, i.e., given an input sample $S = (S_+, S_-)$, it always returns a 1-DTA $\mathcal{A}$ such that $S_+ \subset L(\mathcal{A})$ and $S_- \subset L(\mathcal{A})^c$. But it this result also desired? There are infinitely many consistent 1-DTAs. Given an input sample obtained from a 1-DTA language L_t (the target language), the desired result is a 1-DTA $\mathcal{A}$ such that $L(\mathcal{A}) = L_t$. In this section, we show that our algorithm returns the desired result *in the limit*. Moreover, it does so *efficiently*, i.e., it only requires a polynomial amount of examples in the size of the smallest 1-DTA model for the target language L_t. These are the two properties required for efficient identification in the limit of 1-DTAs and we prove this by showing the existence of polynomial characteristic sets (Definition 3.3). We show this by proving that the four properties mentioned in the previous section hold for our algorithm:

- First, we prove that the ID_1-DTA algorithm is a polynomial-time algorithm. This satisfies properties 1 and 2.

- Second, we prove that in every iteration of the ID_1-DTA algorithm, there exists a polynomial amount of timed strings that can force our algorithm to identify the correct δ (Lemma 3.26). The union of all of these timed strings form a characteristic set S_{cs} for the ID_1-DTA algorithm.

- Third, we prove that the ID_1-DTA algorithm converges efficiently, i.e., that only a polynomial amount of (correctly identified) transitions are required to construct a 1-DTA $\mathcal{A}$ such that $L(\mathcal{A}) = L_t$ (Proposition 3.27).

In order to bound the length of the timed strings in S_{cs} (and hence the size of S_{cs}) we make use of the facts that 1-DTAs are both polynomially reachable (Proposition 3.15) and polynomially distinguishable (Theorem 4). The combination of these results satisfies all the constraints required for efficient identification in the limit (Definition 3.4), and hence shows that 1-DTAs are efficiently identifiable (Theorem 5).

Proposition 3.25. *ID_1-DTA is a polynomial-time algorithm (properties* 1 *and* 2*).*

Proof. In Algorithm 3.1, the main loop stops when the inner while loop cannot iterate anymore. Therefore, the running time of Algorithm 3.1 is bounded by the running time of the iterations on the inner while loop (plus a constant factor for determining $v_{\min}$ and c'). We have to show that the running time of the inner while loop can be bounded by a polynomial in the size of the input sample S.

In every iteration of the inner while loop of Algorithm 3.1, the algorithm identifies (constructs) one new transition. In the lower_bound subroutine, this transition is identified using a set V of valuations that is constructed using timed strings from S. It can be the case that V is empty but this case can be neglected since it occurs rarely and does not cause any additional iterations of the inner while loop. Since V is non-empty, the new transition is fired by at least one timed string τ from the input sample S. Every timed string fires a number of transitions equal to or less than its length. Hence, every example $\tau \in S$ can create at most $|\tau|$ iterations of the inner while loop in the worst case. Thus, in total there are at most $\sum_{\tau \in S} |\tau|$ iterations of the inner while loop. This is polynomial in the input size.

The lower_bound subroutine iterates $|V|$ times, which is bounded by $|S|$, and hence is bounded by a polynomial of the input size. Constructing V boils down to checking for every prefix τ_i of every timed string $\tau \in S$, whether τ_i fires δ. Thus, we need to make $|S|$ checks. These checks can be performed by running $\mathcal{A}$ over τ_i while increasing the value of i. This way, we can perform each check in $O(1)$ time. Constructing V thus requires $O(|S|)$ running time.

The loop for identifying the target state iterates $|Q|$ times. Since the algorithm can in the worst case identify a new state in every iteration of the inner while loop, this number is also bounded by a polynomially of the input size. The consistency check (consistent) can be implemented by trying all combinations of indexes of positive and negative examples. Therefore its worst-case complexity is $\sum_{\tau \in S_+, \tau' \in S_-} |\tau| * |\tau'|$, which is polynomially bounded in the input size $|S|$.

Since polynomials are closed under composition, the running time of the inner while loop can be bounded by a polynomial in the size of the input sample. Hence, Algorithm 3.1 is a polynomial-time algorithm. □

The above proposition shows that our algorithm is time-efficient. More specifically, given any input sample S, ID_1-DTA returns in polynomial time a 1-DTA $\mathcal{A}$ that is consistent with S, i.e., such that $S_+ \subseteq L(\mathcal{A})$ and $S_- \subseteq L(\mathcal{A})^{\mathrm{c}}$. We now show that the ID_1-DTA algorithm is also data-efficient. We first show that it requires a polynomial amount of data for a single transitions. Then we show that it converges after a polynomial amount of transitions.

Lemma 3.26. *There exist polynomial characteristic sets of the transitions of 1-DTAs for ID_1-DTA (property* 3*).*

Proof. First, our algorithm identifies whether q_0 is a final state. The example that ensures correct identification is the empty timed string λ: $\lambda \in S_+$ if $q_0 \in F$ and $\lambda \in S_-$ otherwise. Including this example in S makes our algorithm identify q_0 as a final state only when it should in fact be a final state. In a similar way, we now show that there exists a polynomial amount of polynomial-sized timed strings that ensure the correct identification of the transitions of $\mathcal{A}$. We prove this by showing that there exists a polynomial characteristic set for every transition δ such that our algorithm will identify the lower bound c, the reset r, and the target state q' of δ correctly.

The lower bound c. We need to ensure that our algorithm identifies the correct lower bound c. The lower_bound subroutine selects a valuation v that is the smallest valuation from V that leads to a consistent 1-DTA $\mathcal{A}$. Thus, we need to find examples that guarantee that the correct valuation is an element of V, that $\mathcal{A}$ is consistent when this valuation is selected as a lower bound, and that $\mathcal{A}$ is inconsistent when any smaller valuation is selected.

We can guarantee the correct valuation to be an element of V using a single example $\tau(a, c - v_{\min})$, where τ is a timed string that ends in $(q, v_{\min})$. Since $(q, v_{\min})$ is reachable, this example is guaranteed to exist. Moreover, when the algorithm constructs V, this example will end in (q, c). This ensures that c is an element of V. Naturally, it should be the case that $\tau(a, c - v_{\min}) \in S_+$ if and only if $\tau(a, c - v_{\min}) \in L_t$.

In order to ensure that $\mathcal{A}$ is consistent when c is selected we do not require any examples. This consistency is guaranteed by definition since our algorithm should identify c, and since the initial target of δ is a new state. Since this new state can be reached by no transition other than δ, the fact that S is an input sample for L_t guarantees that there can be no pair of timed strings in S that lead to an inconsistency when the correct lower bound c is selected.

Both a positive and a negative example are required to ensure that $\mathcal{A}$ is inconsistent when any smaller valuation is selected. These examples $\tau \in L_t$ and $\tau' \notin L_t$ should be such that if a smaller valuation is selected, then after some prefixes τ_i and τ'_j of τ and τ', they both reach the same timed state, and their subsequent computations are identical. The valuations of the timed states in which τ_i and τ'_j end (after firing δ) depend on whether δ contains a reset or not. We later show the existence of examples which ensure that δ contains a reset only if δ_t contains a reset. Now, we therefore only need to show the examples that are required, depending on whether $r = \mathit{true}$ or $r = \mathit{false}$.

In the case that $r = \mathit{true}$, the two examples we require are $\tau(a, c - v_{\min})\tau'$, and $\tau(a, c - v_{\min} - 1)\tau'$, where τ is a timed string that ends in $(q, v_{\min})$, and τ' is a timed string such that $\tau(a, c - v_{\min})\tau' \in L_t$ and $\tau(a, c - v_{\min} - 1)\tau' \notin L_t$, or vice versa. Because $(q, v_{\min})$ is reachable, and because our algorithm should select c, these examples are guaranteed to exist. Essentially, τ' is a string that distinguishes between the two languages $L_1 = \{\tau_1 \mid \tau(a, c - v_{\min})\tau_1 \in L_t\}$ and $L_2 = \{\tau_2 \mid \tau(a, c - v_{\min} - 1)\tau_2 \in L_t\}$. Since our algorithm should select c, it holds that $L_1 \neq L_2$. Otherwise, our algorithm might as well select $c - 1$, if all other identifications are performed correctly the result will still be such that $L(\mathcal{A}) = L_t$. This would contradict the fact that our algorithm should select c.

In the case that $r = \mathit{false}$, the two examples are $\tau(a, c - v_{\min})(b, t)\tau'$ and $\tau(a, c - v_{\min} - 1)(b, t+1)\tau'$, where τ is a timed string, b is a symbol, t is a time value, and τ' is a timed string, such that τ ends in $(q, v_{\min})$ and $\tau(a, c - v_{\min})(b, t)\tau' \in L_t$ and $\tau(a, c - v_{\min} - 1)(b, t + 1)\tau' \notin L_t$, or vice versa. These examples are similar to the ones we require when $r = \mathit{true}$. The only difference being that in order to reach the same valuation in q' (and hence being capable of creating an inconsistency), the second example has to wait one additional time value. The examples are guaranteed to exists because our algorithm should select c.

The reset r. The correct identification of the clock reset r of δ is ensured by similar examples. In the case that $r = \mathit{false}$, the two examples we require are: $\tau(a, c - v_{\min})\tau'$ and $\tau(a, c - v_{\min} + 1)\tau'$. In the case that $r = \mathit{true}$, we require: $\tau(a, c - v_{\min})(b, t)\tau'$ and $\tau(a, c - v_{\min} + 1)(b, t - 1)\tau'$. Because these examples reach the same valuations in q' only if x is reset while it should not be (or the other way around), these examples can be used to create a inconsistency. The difference of 1 time value is sufficient for such an inconsistency since 1 time value can always be the difference between satisfying and not satisfying a clock guard. Hence, these inconsistencies are guaranteed to exist.

In the case that r is set incorrectly, the timed strings that guarantee the correct identification of r ensure that there is an inconsistency when our algorithm selects the correct lower bound c. Hence, these examples ensure that setting r incorrectly results in a higher lower bound than setting r correctly. Thus, with these examples our algorithm is guaranteed to identify the correct reset.

The target state q'. We still need to ensure the identification of the correct target state q' of δ. This is achieved by ensuring inconsistencies for every incorrect target state $q'' \neq q'$: $\tau(a, c - v_{\min})(b, t)\tau'$ and $\tau''(b, t')\tau'$, where τ'' ends in q'', and t and t' are such that $\tau(a, c - v_{\min})(b, t)$ and $\tau''(b, t')$ both end in in the same valuation (but not the same state). Naturally, these examples are guaranteed to exist, otherwise q' is identical to q''.

The states Q of $\mathcal{A}$ are identified correctly since our algorithm only adds new states when none of the old ones is a consistent target. We ensure the correct identification of the final states F by requiring for every state an example that ends in it when the state is identified. This completes the specification of all the examples we require for the correct identification of δ by our algorithm.

The amount of examples required for one transition is clearly polynomial. Moreover, since all of the examples consist of a prefix that reaches some specific timed state and a suffix that is a distinguishing string, the fact that 1-DTAs are polynomially distinguishable guarantees that all of the examples are of polynomial length. Moreover, because the order in which our algorithm identifies transitions is independent of S, it is impossible to add additional examples to S such that our algorithm no longer returns $\mathcal{A}$. This proves the lemma. □

We have just shown that our algorithm is capable of returning a correct transition efficiently. We still have to show that it in fact will return a correct 1-DTA efficiently, i.e., that it converges after identifying a polynomial amount of transitions.

Lemma 3.27. *ID_1-DTA converges after identifying a polynomial amount of transitions (property 4).*

Proof. By the previous lemma, our algorithm is capable of making only correct identifications. Notice that, since a 1-DTA is a finite model, only a finite number of such identifications are necessary to make until our algorithm converge to the correct 1-DTA. We have to show that the number of these identifications is polynomial in the size of the correct 1-DTA.

Let $\mathcal{A}_t = \langle Q, x, \Sigma, \Delta, q_0, F \rangle$ be a 1-DTA such that $L_t = L(\mathcal{A}_t)$. Clearly, only the reachable parts of $\mathcal{A}_t$ matter for the acceptance of timed strings. Since 1-DTAs are polynomially distinguishable, there exists some polynomial p such that these parts can be reached by timed strings of length $p(|\mathcal{A}_t|)$. In a single complete run (over all states and symbols) of the main loop, our algorithm identifies new transitions for every newly reachable valuation in any timed state. Hence, the main loop is run at most $p(|\mathcal{A}_t|)$ times before the smallest reachable valuation in any state of $\mathcal{A}_t$ can be identified by our algorithm. In one iteration of the main loop, if all transitions are identified correctly so far, then at most $|\Delta|$ new correct transitions can be identified. Hence, in total at most $p(|\mathcal{A}_t|) * |\Delta|$ transitions are identified before the transition for the smallest reachable valuation can be identified in any state of $\mathcal{A}_t$. This is clearly polynomial in $|\mathcal{A}_t|$. Once the transitions for the smallest reachable valuation can be identified in every state, every transition can be identified. Hence, by the previous proposition, every transition can be identified correctly. Thus, our algorithm can return a 1-DTA $\mathcal{A}$ such that $L(\mathcal{A}) = L_t$ by identifying a number of transitions polynomial in $|\mathcal{A}_t|$. □

The two lemmas above show that our algorithm converges from polynomial data. In other words, if S contains a characteristic subsample S_{cs} for some target language L_t, then ID_1-DTA returns a correct 1-DTA $\mathcal{A}$, i.e., such that $L(\mathcal{A}) = L_t$. Combined with the time efficiency, this is sufficient to prove the efficient identifiability of 1-DTAs:

Theorem 5. *1-DTAs are efficiently identifiable in the limit.*

Proof. By Proposition 3.25 and Lemma 3.27, if all the examples from Lemma 3.26 are included in S, our algorithm returns a 1-DTA $\mathcal{A}$ such that $L(A) = L_t$ in polynomial time and from polynomial data. We conclude that Algorithm 3.1 identifies 1-DTAs efficiently in the limit. □

3.4 Identifying multi-clock DTAs

In the preceding sections, we have proven several results regarding the complexity of identifying DTAs from labeled data. These results have important consequences for anyone who is interested in identifying timed automata. In general, our results show that for the purpose of identifying a timed system, 1-DTAs are more preferred models than multi-clock DTAs (n-DTAs) because they are more efficient to identify. For identifying n-DTAs, we need a huge (exponential) amount of data before we can guarantee convergence, and hence any n-DTA identification algorithm is inefficient by definition.

This is unfortunate, but it does not justify completely rejecting the possibility of identifying an n-DTA instead of a 1-DTA. We could still write an algorithm for identifying n-DTAs and test its performance on some data sets. In fact, in this section we show that by making some straightforward adaptations to the ID_1-DTA algorithm (Algorithm 3.1), we can construct a nearly polynomial-time algorithm for identifying n-DTAs (Section 3.4.1). The resulting algorithm is only exponential in the number of clocks, which will be small in practice. However, while the algorithm runs in near polynomial time, it still requires a huge amount of data in order to identify n-DTAs.

In many practical settings, we do not have access to such amounts of data.[5] Hence, in practice, we will not be able to give correctness guarantees for an n-DTA identification algorithm. Our results show that for a 1-DTA identification algorithm we should in practice be able to give guarantees regarding its correctness. We therefore claim that, in practice, identifying a 1-DTA is better than identifying an n-DTA.

To strengthen this claim, we show that it is possible to identify a 1-DTA that accepts a language that is equivalent to an actual n-DTA language (Section 3.4.2). In other words, we never actually need an n-DTA in order to identify an n-DTA language correctly, a 1-DTA always suffices. Essentially, identifying 1-DTAs instead of n-DTAs reduces the search space (the amount of possible models). The benefit of this is that it now becomes possible to learn a specific type of models (1-DTAs) efficiently. On the downside, identifying n-DTAs can lead to (exponentially) smaller models for the same languages. But this is a small price to pay since identifying these smaller models requires a huge amount of data.

In this section, we first give our algorithm for identifying n-DTAs, and then prove that n-DTAs and 1-DTAs are language equivalent. We conclude with a small discussion regarding the consequences of our results on existing work on the identification of event-recording automata (ERAs) (defined in Section 2.3.2). ERAs are a special class of n-DTAs, and hence they cannot be identified efficiently.

3.4.1 An n-DTA identification algorithm

Although n-DTAs cannot be identified efficiently, it is possible to write a nearly polynomial-time algorithm for the identification of n-DTAs. Very roughly, this algorithm can be constructed by adapting Algorithm 3.1 in the following way:

- Instead of only trying to reset clock x, try to reset every possible combination of clocks from the set of clocks X. Like the ID_1-DTA algorithm, reset the combination that achieves the smallest lower bound.

- Restrict the identification of clock guards to reachable valuations. This ensures that we cannot identify a transition for a valuation that cannot be reached by any timed string (this would lead to incorrect behavior). The reachable valuations can be represented using a so-called clock-zone (see e.g., (Alur and Dill 1994)). Figure 3.5 shows a clock zone.

- When choosing a lower bound for a clock guard, use a total order over the space of valuations. Using this, the algorithm can always select the smallest valuation, even if the valuations are incomparable with respect to the standard order, i.e., if for two valuations v and v' there exist two clocks $x, y \in X$ such that $v(x) > v'(x)$ and $v(y) < v'(y)$.

[5] If we do have access, parsing the data will already take too long for most practical purposes.

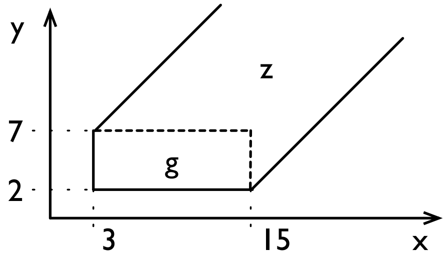


Figure 3.5: A clock zone represents the set of reachable valuations in a state q. The guard $g = 2 \leq y \leq 7 \wedge 3 \leq x \leq 15$ is the guard of an incoming transition for q. Every valuation that satisfies g can be reached in q. In addition, a DTA can perform time transitions, increasing the value of every clock by the same amount. Thus, all the valuations represented by the area z are also reachable.

Proposition 3.28. *The adapted algorithm runs in time polynomial in the size of the input, and exponential in the amount of clocks.*

Proof. By Proposition 3.25, the ID_1-DTA algorithm is a polynomial-time algorithm. Since constructing and using a clock zone and a total order is easy (definitely polynomial), only the first adaptation influences the time complexity of the algorithm. In the step proposed in the first adaptation, the algorithm tries all possible $2^{|X|}$ combinations of clock resets, which is exponential in the amount of clocks. Since this amount is not part of the input sample, this step takes constant time in the size of the input. Hence, the adapted algorithm requires exponential time, but only in the amount of clocks. □

Ensuring the convergence of this adapted algorithm can be done using characteristic sets similar to the ones we used to show the convergence of the original algorithm. However, since DTAs are not polynomially distinguishable (Proposition 3.8), these sets are not of size polynomial in the size of the smallest n-DTA for the target language.

3.4.2 1-DTAs and n-DTAs are language equivalent

In practice, should we use an n-DTA or a 1-DTA identification algorithm? We think it is best to identify 1-DTAs since these can be identified efficiently. Moreover, we show in this section that, besides resulting in smaller models, an n-DTA identification algorithm does not add anything: any n-DTA language can be represented using an identified 1-DTA. More generally, n-DTAs and 1-DTAs are *language equivalent* (also called *notational variants*):

Definition 3.29. *(language equivalence) Two classes of automata $\mathcal{C}_1$ and $\mathcal{C}_2$ are* language equivalent *if for any $\mathcal{A}_1 \in \mathcal{C}_1$, there exists an $\mathcal{A}_2 \in \mathcal{C}_2$ such that $L(\mathcal{A}_1) = L(\mathcal{A}_2)$ and vice versa.*

We show the language equivalence of 1-DTAs and n-DTAs by transforming any n-DTA to a 1-DTA by applying a modified region construction method (see

Section 2.3.2) to all but 1 of the clocks of the n-DTA. In every state of the 1-DTA, the values of the removed clocks can be represented using constant deviations from the value of the remaining clock. This transformation is such that the accepted languages remain the same.

Theorem 6. *The classes of n-DTAs and 1-DTAs are language equivalent.*

Proof. Let $\mathcal{A}_1 = \langle Q, \Sigma, X = \{x_1, \ldots, x_n\}, \Delta, q_0, F\rangle$ be an n-DTA, and let $c \in \mathbb{N}$ be the largest constant occurring in any clock guard of $\mathcal{A}_1$. We define the following 1-DTA $\mathcal{A}_2 = \langle Q^*, \Sigma, \{x_1\}, \Delta^*, q_0^*, F^*\rangle$, where

- $Q^* = \{\langle q, t_2, \ldots, t_n\rangle \mid q \in Q \text{ and } t_i \in [-c+1, c+1] \text{ for } 2 \leq i \leq n\}$
- $\Delta^* = \{\langle \langle q, t_2, \ldots, t_n\rangle, \langle q', t'_2, \ldots, t'_n\rangle, a, x_1 \leq t_1 \wedge x_1 \geq t_1, R^*\rangle \ \mid$
  - $t_1 \in [0, c+1]$
  - $t'_i = \begin{cases} t_1 + t_i & \text{if } x_1 \in R \text{ and } x_i \notin R \\ -t_1 & \text{if } x_1 \notin R \text{ and } x_i \in R \\ 0 & \text{if } x_1 \in R \text{ and } x_i \in R \\ t_i & \text{otherwise} \end{cases}$
  - $R^* = \begin{cases} \{x_1\} & \text{if } x_1 \in R \\ \emptyset & \text{otherwise} \end{cases}$
  - and there exists a transition $\langle q, q', a, g, R\rangle \in \Delta$ such that g is satisfied by a valuation v, where $v(x_1) = t_1$ and $v(x_i) = t_1 + t_i$ for $2 \leq i \leq n$ $\}$
- $q_0^* = \langle q_0, 0, \ldots, 0\rangle$, and
- $F^* = \{\langle q, t_2, \ldots, t_n\rangle \mid \langle q, t_2, \ldots, t_n\rangle \in Q^* \text{ and } q \in F\}$.

We now claim that for every timed string $\tau \in \Sigma \times \mathbb{N}$ it holds that $\tau \in L(\mathcal{A}_1)$ if and only if $\tau \in L(\mathcal{A}_2)$. We prove this by showing that the following invariant holds for any prefix of τ: if τ_i ends in (q, v) in $\mathcal{A}_1$, then τ_i ends in $(\langle q, v(x_2) - v(x_1), \ldots, v(x_n) - v(x_1)\rangle, v^*)$ in $\mathcal{A}_2$, where $v^*(x_1) = v(x_1)$.

For the initial case, $\mathcal{A}_1$ and $\mathcal{A}_2$ start in (q_0, v_0) and $(q_0^* = \langle q_0, 0, \ldots, 0\rangle, v_0^*)$, respectively, where $v_0^*(x_1) = 0$. Thus, if $\tau_0 = \lambda$ ends in $(q = q_0, v = v_0)$ in $\mathcal{A}_1$, then τ_0 ends in $(\langle q = q_0, v(x_2) - v(x_1) = 0 - 0 = 0, \ldots, v(x_n) - v(x_1) = 0\rangle, v^*)$ in $\mathcal{A}_2$, where $v^*(x_1) = v(x_1) = 0$.

For the arbitrary case, let (a, t) be the ith symbol-time value pair of τ, i.e., $\tau_i = \tau_{i-1}(a, t)$. We assume without loss of generality that there exists a computation of $\mathcal{A}_1$ over τ. Thus, τ_{i-1} ends in some timed state (q, v) in $\mathcal{A}_1$, and there exists a transition $\delta = \langle q, q', a, g, R\rangle \in \Delta$ such that g is satisfied by $v + t$. Hence, by definition of $\mathcal{A}_2$, there exists a transition $\delta^* = \langle q^*, \langle q', t'_2, \ldots, t'_n\rangle, a, x_1 \leq t_1 \wedge x_1 \geq t_1, R^*\rangle \in \Delta^*$, where $t_1 = v(x_1) + t$, $R^* = \{x_1\}$ if and only if $x_1 \in R$,

$$\begin{aligned} q^* &= \langle q, (v(x_2) + t) - t_1, \ldots, v'(x_n) - t_1\rangle \\ &= \langle q, v(x_2) + t - (v(x_1) + t), \ldots, v(x_n) + t - (v(x_1) + t)\rangle \\ &= \langle q, v(x_2) - v(x_1), \ldots, v(x_n) - v(x_1)\rangle, \end{aligned}$$

and

$$t'_i = \begin{cases} t_1 + t_i = v(x_1) + t + v(x_i) - v(x_1) = t + v(x_i) & \text{if } x_1 \in R \text{ and } x_i \notin R \\ -t_1 = -v(x_1) - t & \text{if } x_1 \notin R \text{ and } x_i \in R \\ 0 & \text{if } x_1 \in R \text{ and } x_i \in R \\ t_i = v(x_i) - v(x_1) & \text{otherwise.} \end{cases}$$

Due to the existence of δ, and the fact that τ_{i-1} ends in some timed state (q, v) in $\mathcal{A}_1$, we know that τ_i ends in (q', v') in $\mathcal{A}_1$, where

$$v'(x) = \begin{cases} v(x) + t & \text{if } x \notin R \\ 0 & \text{if } x \in R. \end{cases}$$

Suppose for the sake of induction that τ_{i-1} ends in $(\langle q, v(x_2) - v(x_1), \ldots, v(x_n) - v(x_1)\rangle, v^*)$ in $\mathcal{A}_2$, where $v^*(x_1) = v(x_1)$. Due to the existence of δ^*, this implies that τ_i ends in $(\langle q', t'_2, \ldots, t'_n\rangle, v^{*'})$ in $\mathcal{A}_2$, where

$$v^{*'}(x_1) = \begin{cases} v(x_1) + t = v'(x_1) & \text{if } x \notin R^*\text{, hence if } x_1 \notin R \\ 0 = v'(x_1) & \text{if } x \in R^*\text{, hence if } x_1 \in R. \end{cases}$$

and

$$v'_i = \begin{cases} t + v(x_i) = v'(x_1) - 0 = v'(x_i) - v'(x_1) & \text{if } x_1 \in R \text{ and } x_i \notin R \\ -v(x_1) - t = 0 - v'(x_1) = v'(x_i) - v'(x_1) & \text{if } x_1 \notin R \text{ and } x_i \in R \\ 0 = 0 - 0 = v'(x_i) - v'(x_1) & \text{if } x_1 \in R \text{ and } x_i \in R \\ v(x_i) - v(x_1) = (v'(x_i) - t) - (v'(x_1) - t) = v'(x_i) - v'(x_1) & \text{otherwise.} \end{cases}$$

By induction, we conclude that holds that if τ_i ends in (q, v) in $\mathcal{A}_1$, then τ_i ends in $(\langle q, v(x_2) - v(x_1), \ldots, v(x_n) - v(x_1)\rangle, v^*)$ in $\mathcal{A}_2$. We are now ready to prove that $\tau \in L(\mathcal{A}_1)$ if and only if $\tau \in L(\mathcal{A}_2)$:

($\Rightarrow$) If $\tau \in L(\mathcal{A}_1)$, then τ ends in some final state $q \in F$ in $\mathcal{A}_1$. Therefore, τ ends in $\langle q, v(x_2) - v(x_1), \ldots, v(x_n) - v(x_1)\rangle \in F^*$ in $\mathcal{A}_2$, and hence $\tau \in L(\mathcal{A}_2)$.

($\Leftarrow$) If $\tau \notin L(\mathcal{A}_1)$, then τ ends in some non-final state $q \notin F$ in $\mathcal{A}_1$. Therefore, τ ends in $\langle q, v(x_2) - v(x_1), \ldots, v(x_n) - v(x_1)\rangle \notin F^*$ in $\mathcal{A}_2$, and hence $\tau \notin L(\mathcal{A}_2)$. □

Theorem 6 tells us that we can represent any n-DTA language using an (exponentially larger) 1-DTA or, more tailored to the problem of identifying n-DTAs:

Corollary 3.30. *Given an n-DTA target language L_t, there exists a 1-DTA $\mathcal{A}$ such that $L(\mathcal{A}) = L_t$.*

Proof. The statement follows directly from Theorem 6. □

Hence, we never actually need an n-DTA in order to identify an n-DTA language correctly, a 1-DTA always suffices. Because 1-DTAs can be identified efficiently, and n-DTAs cannot, this supports our claim that it is better to identify 1-DTAs in practice.

3.4.3 Identifying other classes of DTAs

In related work, a query learning algorithm is described for identifying event recording automata (ERAs) (Grinchtein et al. 2006). An ERA is a TA where each symbol from the alphabet a is associated with exactly one clock x_a. Each transition of an ERA resets only the clock its label is associated with. Thus during a computation of an ERA, the value of a clock x_a is always equal to the time since the last time a transition with label a was fired.

An important property of ERAs is that they are *determinizable* (Alur et al. 1999). A class of automata C is determinizable if for every non-deterministic automaton $\mathcal{A} \in C$ it is possible to construct a deterministic automaton $\mathcal{A}' \in C$, such that they accept the same language, i.e., $L(\mathcal{A}) = L(\mathcal{A}')$. This is important for identification because the language inclusion problem is decidable for classes of determinizable TAs. For TAs in general, the language inclusion problem is undecidable, and hence it is difficult (if not impossible) to bound the amount of data required to distinguish one TA from another TA. Since the problem of identifying an automaton is basically the same as distinguishing automata (from each other) based on some data, this seems to indicate that deterministic ERAs (DERAs) form a class of automata that are well-suited for identification. However, a class of automata can only be identified efficiently from queries if it is also efficiently identifiable in the limit from labeled data (Parekh and Honavar 2000). Thus, our results show that DERAs can never be identified efficiently since an ERA has access to multiple clocks (an ERA can be used to prove Proposition 3.8).

We believe it would be interesting, and very valuable for real-world applications, to adapt the timed query learning algorithm to the class of 1-DTAs. Our results indicate that this may result in an efficient query learning algorithm for timed systems. However, in (Grinchtein et al. 2006), it is shown that the timed query learning algorithm only requires an amount of queries polynomial in the length of the shortest counterexample (i.e., distinguishing string). If it can be shown that this counterexample is of length polynomial in the size of the smallest 1-DTA for the same language, then this algorithm already query-learns 1-DTAs efficiently while representing them using DERAs.

3.5 The expressive power of one-clock and multi-clock DTAs

In general, our lemmas and theorems are statements about the expressive power of one-clock DTAs (1-DTAs) and multi-clock DTAs (n-DTAs). These statements are important by themselves, i.e., not necessarily restricted to just the identification problem. We believe that there can be other problems (such as reachability analysis) that may benefit from our results. In this section we give an overview of the consequences of our results in general.

First of all, Theorem 4 tells us that:

> The length of the shortest string in the *symmetric difference* $L(\mathcal{A}_1) \triangle L(\mathcal{A}_2)$ between the languages of any two 1-DTAs $\mathcal{A}_1$ and $\mathcal{A}_2$ is of length bounded by a polynomial p in the sizes of $\mathcal{A}_1$ and $\mathcal{A}_2$.

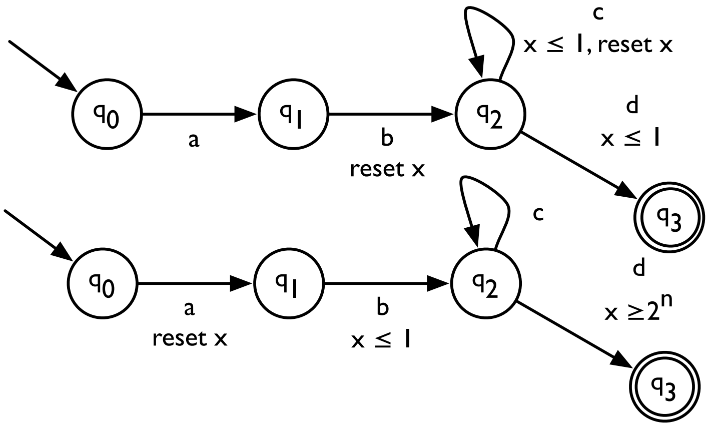


Figure 3.6: Two 1-DTAs. Taking the intersection of the languages of these 1-DTAs results in a language equivalent to the language of the 2-DTA of Figure 3.2.

In the timed automata (formal methods) field, it is well-known that there exists a language-preserving transformation from any DTA with n clocks to n 1-DTAs (Asarin, Caspi and Maler 2001). In fact, this transformation is quite straightforward: given an n-DTA, create a 1-DTA copy for every clock and set all occurrences of different clocks in clock guard to *true*. The intersection of the languages of these n 1-DTAs is (with some additional transformations) equal to the language of the original DTA. An example of two such 1-DTAs is shown in Figure 3.6. Figure 3.2 is the intersected n-DTA version of Figure 3.6.

The combination of this fact with Proposition 3.8 tells us that:

> The shortest string in the *intersection* $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$ between the languages of two 1-DTAs $\mathcal{A}_1$ and $\mathcal{A}_2$ *cannot* be bounded by a polynomial p in the sizes of $\mathcal{A}_1$ and $\mathcal{A}_2$.

The two statements above tell us something important regarding the expressive power of 1-DTAs and n-DTAs. We also know that the complement $L(\mathcal{A})^C$ of the language of a 1-DTA $\mathcal{A}$ can be easily computed (polynomially) by changing the final and non-final states of $\mathcal{A}$. Since 1-DTA are polynomially reachable (Proposition 3.15), this implies that:

> The shortest string in the (non-symmetric) *difference* $L(\mathcal{A}_1) \setminus L(\mathcal{A}_2) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)^C$ between the languages of any two 1-DTAs $\mathcal{A}_1$ and $\mathcal{A}_2$ *cannot* be bounded by a polynomial p in the sizes of $\mathcal{A}_1$ and $\mathcal{A}_2$.

In fact, using the basic set operations, the above statements, and some algebra, we can show many of such statements. These statements are summarized in Figure 3.7. In general, they tell us that the intersection of the languages of two 1-DTAs is strictly more expressive than either of these 1-DTAs: by Theorem 6 it can be represented using a 1-DTA, but at the cost of exponential blowup. Since this

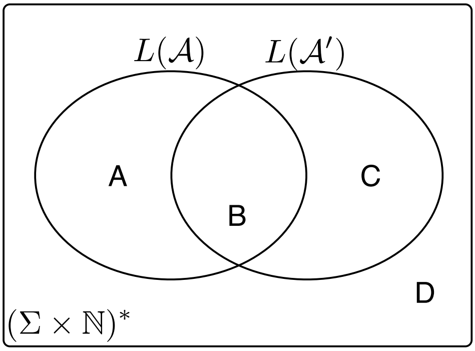


Figure 3.7: The power of two 1-DTAs depicted in a Venn diagram. $\mathcal{A}$ and $\mathcal{A}'$ are two 1-DTAs. In the Venn diagram there are 4 spaces. The language $L(\mathcal{A})$ of $\mathcal{A}$ is equal to $A \cup B$. The language $L(\mathcal{A}')$ of $\mathcal{A}'$ is equal to $C \cup B$. B is the intersection of $L(\mathcal{A})$ and $L(\mathcal{A})$. Thus, B is a 2-DTA language, and hence we cannot polynomially bound the size of the smallest timed string in B (Proposition 3.8). Using complementation and again intersection, the same holds for A, C, and D. However, we can polynomially bound the size of the smallest timed string in $X \cup Y$ where $X, Y \in \{A, B, C, D\}$ (Proposition 3.15 and Theorem 4).

tells us something about the power of clocks in DTAs in general, we believe that it should be possible to apply our theorems to for example reachability analysis in timed automata. Perhaps they can be used to reduce the complexity of reachability analysis depending on which type of combination (using set operations) of the languages of 1-DTAs has to be analyzed.

3.6 Discussion

In this chapter we have shown the following main results:

1. Polynomial distinguishability (Definition 3.7) is a necessary condition for efficient identification in the limit (Lemma 3.9).
2. DTAs with two or more clocks are not polynomially distinguishable, and thus not efficiently identifiable (Theorem 1 and Corollary 3.10).
3. DTAs with one clock (1-DTAs) are polynomially distinguishable (Theorem 4).
4. 1-DTAs are efficiently identifiable using our ID_1-DTA algorithm (Algorithm 3.1 and Theorem 5).
5. 1-DTAs and n-DTAs (DTAs with n clocks) are language equivalent (Theorem 6).

In addition, we described roughly how the ID_1-DTA algorithm could be adapted in order to identify n-DTAs, albeit inefficiently (Section 3.4.1).

Since DFAs can be identified efficiently (Oncina and Garcia 1992), the reason for our inefficiency results has to be due to the explicit representation of time (using numbers) that is used in DTAs. We know that for any DTA, there exists a DFA that models the exact same language (is language equivalent) but with an implicit representation of time (using states). The standard method of creating such a DFA is the region construction (see Section 2.3.2). This construction results in a DFA of size exponential in both the binary encoding of time values used in the DTA and the amount of clocks of the DTA. Therefore, it is not unexpected that DTAs cannot be identified efficiently. In fact, a straightforward way to identify a DTA is to first transform the input to make it suitable for an implicit representation of time. This can be done using for example a *sampling* method: every timed symbol (a, t) is replaced by $a \circlearrowright \ldots \circlearrowright$, where $\circlearrowright$ is a special symbol denoting a time increase and $\mid \circlearrowright \ldots \circlearrowright \mid = t$. From a transformed input a DFA can be identified that models a language that is equivalent to the language of any DTA that could be identified from the original input. However, since both the transformed input and the resulting DFA are exponentially larger than their timed counterparts, such a method is highly inefficient.

It comes as a surprise that 1-DTAs can be identified efficiently. This is surprising since the region construction still results in an exponentially larger DFA when applied to a 1-DTA: time is still represented in binary (instead of unary). The main reason that 1-DTAs can be identified efficiently is an important lemma regarding their modeling power (Lemma 3.19). This lemma states a restriction on the shortest timed strings in the symmetric difference of two 1-DTA languages. This restriction is then used to prove that 1-DTAs can be distinguished using a timed string of polynomial length. We believe however, that this restriction and the results that come from this restriction have consequences that are of importance outside the scope of the DTA identification problem (see Section 3.5).

Our results and algorithm are of importance for anyone interested in identifying timed systems (and DTAs in particular). Most importantly, the efficiency results tell us that identifying a 1-DTA from timed data is more efficient than identifying an equivalent DFA. Furthermore, the results show that anyone who needs to identify a DTA with two or more clocks should either be satisfied with sometimes requiring an exponential amount of data, or he or she has to find some other method to deal with this problem, for instance by identifying a subclass of DTAs (such as 1-DTAs).

CHAPTER 4

Identifying deterministic real-time automata

This chapter is based on work published in Proceedings of the Belgiun-Dutch Conference on Artificial Intelligence (Verwer, de Weerdt and Witteveen 2005) and Proceedings of the Annual Machine Learning Conference of Belgium and the Netherlands (Verwer, de Weerdt and Witteveen 2007).

4.1 Introduction

In this chapter, we present a novel method for the automatic identification (learning) of a real-time system from positive and negative data. This data consists of sequences that could have been generated by a real-time system. A positive sequence characterizes the (correct) behavior of the system, and a negative sequence does not (or characterizes faulty behavior). In order to obtain such data, the data collected from observations will need to be labeled, i.e., an expert will need to decide for some data sequences whether the sequence is an example of the system's behavior or not. From such data, we try to find a model that agrees with all the positive examples, and none of the negative ones. Moreover, this model is preferably smallest amongst all possible models that are consistent with the data. The identified model can be used to determine whether new event sequences should be classified as being generated by the real-time system.

A well-known automaton model for characterizing systems is the *deterministic finite automaton* (DFA, see Section 2.3.1). A DFA is a language model. Hence, its identification (or inference) problem has been well studied in the grammatical inference field (de la Higuera 2005). Knowing this, we want to take an established method to learn a DFA and apply it to our event sequences. However, when observing a *real-time* system, there often is more information than just the sequence of symbols: the time at which these symbols occur is also available. By itself, the

DFA model is too limited to handle this timed information. A straightforward way to make use of this timed information is to *sample* the timed data. For instance, a symbol that occurs 3 seconds after the previous event can be modeled as 3 special time tick symbols followed by the symbol. A disadvantage of this approach is that it results in an exponential blowup of both the input data and the resulting automaton size. In this chapter, we propose a new algorithm that uses the time information directly in order to produce a timed model, i.e., a timed automaton (TA, see Section 2.3.2).

Unfortunately, we showed in the previous chapter that, because TAs can model combinations of time constraints between any two events, regular TAs are very difficult to identify from data. In particular, we showed that TAs cannot be identified efficiently in the limit from labeled data (Theorem 1). This means that we require at least an exponential amount of data before we can ensure the convergence of a TA identification algorithm. In many practical settings, however, we do not have access to such amounts of data, and even if we do have access, parsing this data will already take too long for most practical purposes. Hence, in practice, we will not be able to guarantee the correctness of a TA identification algorithm. This makes it difficult to apply a TA identification algorithm.

Fortunately, in the same chapter, we also showed that if we restrict ourselves to deterministic TAs with at most one clock (1-DTAs), then we can identify these efficiently in the limit (Theorem 5). Hence, in practice, we will be able to guarantee the correctness of a 1-DTA identification algorithm. In this chapter, we focus on identifying a simple type of 1-DTA, known as a deterministic real-time automaton (DRTA, see Section 4.2). A DRTA models only the time constraints between two *consecutive* events, instead of between two arbitrary events. We restrict ourselves to DRTAs and not to the full class of 1-DTAs because they are expressive enough for many interesting applications, including for instance modeling truck driver behavior (see Figure 4.1), which is the motivating example application of our techniques (see Chapter 6). Furthermore, since DFA identification already is a difficult problem, it makes sense to first focus on simple extensions of DFAs.

We provide an algorithm for identifying DRTAs from labeled data-sets (Section 4.3). We call this algorithm RTI, which stands for *real-time identification*. The RTI algorithm is based on the currently best-performing algorithm for the identification of DFAs, called evidence-driven state-merging (ESDM, see Section 2.4.1). The only difference between DFAs and DRTAs are the time constraints. Although this seems a small difference, the problem of identifying a DRTA is much more difficult than the problem of identifying a DFA: we show the problem of identifying only the time constraints of a DRTA to be already NP-complete (Section 4.3.1). More specifically, we show that identifying the correct time constraints is as difficult as solving the satisfiability problem. In spite of the complexity of this additional problem, RTI is a polynomial-time, correct, and complete algorithm that converges efficiently to the correct DRTA in the limit (Section 4.3.4).

A big difference between EDSM and RTI is that RTI has to deal with timed data. Therefore, RTI requires a different evidence measure than the one used by EDSM, which is based on non-timed data. We provide a few measures for timed data that can be used in RTI (Section 4.4). We then experimentally compare each of these different evidence measures for RTI and a sampling approach (Section 4.5).

The sampling approach first replaces the time values in timed strings by special time tick symbols, and then runs EDSM to obtain a DFA representation of a DRTA. We perform the experiments on a large set of artificial data-sets generated from a randomly generated DRTA. The main conclusion of these experiments in that RTI significantly outperforms the sampling approach.

In this chapter, we thus show the following:

- We show that it is possible to efficiently identify DRTAs from labeled data: RTI identifies DRTAs efficiently in the limit.
- We show that it is useful to identify a DRTA: by identifying a DRTA we obtain a better performance than identifying a DFA from sampled data.

We conclude this chapter with a summary and a discussion of the RTI algorithm, including possible applications and ideas for future work (Section 4.6).

4.2 Real-time automata

In a real-time system, each occurrence of a symbol (event) is associated with a time value, i.e., its time of occurrence. As before, we model these time values using the *natural numbers* $\mathbb{N}$. This is sufficient because in practice we always deal with a finite precision of time, e.g. milliseconds. Timed automata (see Section 2.3.2) can be used to accept or generate a sequence $\tau = (a_1, t_1)(a_2, t_2)(a_3, t_3) \dots (a_n, t_n)$ of symbols $a_i \in \Sigma$ paired with time values $t_i \in \mathbb{N}$, called a *timed string*. Every time value t_i in a timed string represents the time (delay) until the occurrence of symbol a_i since the occurrence of the previous symbol a_{i-1}.

In timed automata, timing conditions are added using a finite number of *clocks* and a *clock guard* for each transition. In this section, we describe the class of timed automata that we use in this chapter, known as *real-time automata* (RTAs) (Dima 2001). An RTA has only one clock that represents the time delay between two *consecutive events*. The clock guards for the transitions are then constraints on this time delay. When trying to identify an RTA from a timed input sample S, one can always determine an upper bound on the possible time delays by taking the maximum observed delay from S. Therefore, we represent a *delay guard* (constraint) $[n, n']$ by a *closed interval* in $\mathbb{N}$. We say that $[n, n']$ is *satisfied* by a time value $t \in \mathbb{N}$ if $t \in [n, n']$. An RTA is defined as follows:

Definition 4.1. *(RTA) A* real-time automaton *(RTA) is a 5-tuple $\mathcal{A} = \langle Q, \Sigma, \Delta, q_0, F \rangle$, where Q is a finite set of states, Σ is a finite set of symbols, Δ is a finite set of transitions, q_0 is the start state, and $F \subseteq Q$ is a set of accepting states.*

A transition $\delta \in \Delta$ in an RTA is a tuple $\langle q, q', a, [n, n'] \rangle$, where $q, q' \in Q$ are the source and target states, $a \in \Sigma$ is a symbol, and $[n, n']$ is a delay guard.

Due to the complexity of learning non-deterministic (timed) automata (see Section 3.1.1), we only consider deterministic RTAs (DRTAs). An RTA $\mathcal{A}$ is called *deterministic* if $\mathcal{A}$ does not contain two transitions with the same symbol, the same source state, and overlapping delay guards. Like timed automata, in DRTAs, it is possible to make time transitions in addition to the normal state transitions used in deterministic finite state automata (DFAs). In other words,

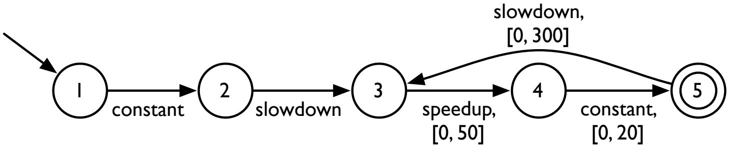


Figure 4.1: The harmonica driving behavior modeled as a DRTA. The numbers used in the delay guards are amounts of tenths of a second.

during its execution a DRTA can remain in the same state for a while before it generates the next symbol. The time it spends in every state is represented by the time values of a timed string. In a DRTA, a state transition is possible (can fire) only if its delay guard is satisfied by the time spent in the previous state. A transition $\langle q, q', a, [n, n'] \rangle$ of a DRTA is thus interpreted as follows: whenever the automaton is in state q, reading a timed symbol (a, t) such that $t \in [n, n']$, then the DRTA will move to the next state q'.

Example 4.1. Figure 4.1 shows an example DRTA that models a specific driving behavior known as 'harmonica driving'. This often occurs when a truck is driving at a somewhat higher speed than the vehicle directly in front of it. The driver slows down a bit, waits until there is enough distance between him and the vehicle in front, and then speeds up again, closing in on the vehicle. This whole process often repeats itself a couple of times before the driver finally adjusts the speed of the truck to match the vehicle in front of him. The result of this whole process is unnecessary fuel consumption.

The complete behavior of a DRTA is defined by the *computation* of a DRTA:

Definition 4.2. *(DRTA computation) A finite* computation *of a DRTA $\mathcal{A} = \langle Q, \Sigma, \Delta, q_0, F \rangle$ over a (finite) timed string $\tau = (a_1, t_1) \dots (a_n, t_n)$ is a finite sequence*

$$q_0 \xrightarrow{(a_1,t_1)} q_1 \cdots q_{n-1} \xrightarrow{(a_n,t_n)} q_n$$

such that for all $1 \leq i \leq n$, $\langle q_{i-1}, q_i, a_i, [n_i, n'_i] \rangle \in \Delta$, where $t_i \in [n_i, n'_i]$. A computation of a DRTA is called accepting *if $q_n \in F$.*

We say that a timed string τ *ends* in a DRTA $\mathcal{A}$ in the last state occurring in the computation of $\mathcal{A}$ over τ, i.e., τ ends in q_n. A DRTA $\mathcal{A}$ accepts a timed string τ if τ ends in a final state, i.e., if $q_n \in F$. The language of a DRTA $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of timed strings τ that the computation of $\mathcal{A}$ over τ is accepting.

Since DRTAs can be modeled using one-clock deterministic timed automata (1-DTAs), Theorem 5 implies that DRTAs are efficiently identifiable in the limit. Moreover, we could use ID_1-DTA (Algorithm 3.1) to identify them efficiently. This algorithm, however, is mainly useful for proving this efficiency result. In practice, it will often fail to identify the correct DRTA because it relies on there being a specific set of timed strings (a characteristic set). This set of timed strings is not very likely to occur in practice, and thus it is usually better to use some form of heuristic evidence to guide the identification process. In the next section, we describe an evidence-driven approach for the identification of DRTAs.

4.3 Identifying real-time automata

Given a timed input sample $S = (S_+, S_-)$ (a pair of sets of positive and negative timed strings), we want to identify the smallest DRTA $\mathcal{A}$ that is consistent with S, i.e., the smallest DRTA $\mathcal{A}$ such that $S_+ \subseteq L(\mathcal{A})$ and $S_- \subseteq L(\mathcal{A})^c$. This problem adds a difficult subproblem to the DFA identification problem: in addition to identifying the correct DFA structure, the algorithm needs to identify the correct delay guards. Preferably, we would like an algorithm that solves this subproblem optimally. It should then be possible to use this algorithm as a subroutine of a DFA identification algorithm in order to identify DRTAs. Unfortunately, we start this section by proving that identifying only these timed properties of a DRTA is already NP-complete. Thus, we will not be able to solve this subproblem efficiently unless P = NP (Section 4.3.1).

Therefore, instead of treating the identification of delay guards as a subproblem, we provide a new algorithm that identifies delay guards in a way similar to the way it identifies states and transitions. We call this algorithm RTI (Algorithm 4.5), which stands for *real-time identification*. The RTI algorithm is based on the evidence-driven state-merging (EDSM) algorithm, which is one of the most successful algorithms for identifying DFAs (see Section 2.4.1). RTI can perform all of the traditional state-merging routines (Algorithms 4.1 and 4.3). In addition, our DRTA identification algorithm is capable of identifying delay guards by *splitting* transitions into two (Algorithm 4.2).

Both the transition-splitting and the modified state-merging routines are explained in Section 4.3.2. Afterwards, we present the RTI algorithm for identifying DRTAs in Section 4.3.3. In Section 4.3.4, we end this chapter by proving that RTI runs in polynomial time, is correct, is complete (Propositions 4.3 to 4.5), and that it converges efficiently to the correct DRTA (Proposition 4.6).

4.3.1 Identifying delay guards of a DRTA is NP-complete

Identifying the correct delay guards is a hard problem. In fact, in the very favorable circumstance where the correct non-deterministic finite state automaton (NFA) structure is already given, the problem of identifying the correct delay guards is still NP-complete:

Theorem 7. *Given a timed input sample S and an NFA $\mathcal{A} = \langle Q, \Sigma, \Delta, q_0, F\rangle$, the problem of finding for every transition $\delta = \langle q, q', a\rangle \in \Delta$ a delay guard $[n, n']$ such that $\langle Q, \Sigma, \{\langle q, q', a, [n, n']\rangle \mid \langle q, q', a\rangle \in \Delta, n, n' \in \mathbb{N}\}, q_0, F\rangle$ is a DRTA consistent with S is NP-complete.*

Proof. Our proof is by reduction from 3-SAT (for a definition see e.g. (Sipser 1997)). We first give the intuition behind this reduction, then we give the formal proof.

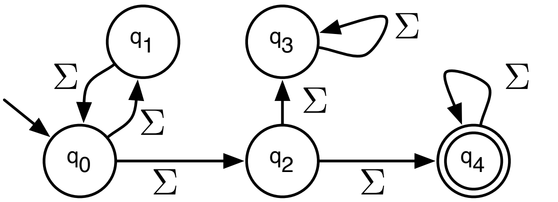


Figure 4.2: The incomplete DRTA (without delay guards) resulting from the reduction in the proof of Theorem. The alphabet Σ consists of the variables V of the 3-SAT instance.

The reduction consists of two parts: constructing the NFA $\mathcal{A}$, and constructing the input sample S. The NFA $\mathcal{A}$ has a structure that is mostly independent of the 3-SAT instance; only the alphabet Σ contains one symbol a for every variable $a \in V$ in the 3-SAT instance. This structure is depicted in Figure 4.2. The main idea of this structure is that the accepting state q_4 can only be reached on an even index. For every clause c in the 3-SAT instance, a positive timed string τ of length 6 is constructed. Hence, every such string τ has 3 opportunities to reach state q_4, once for each literal l in c. Whether they reach state q_4 in one such opportunity depends on the delay guards $[n_1, n_2]$ and $[n_3, n_4]$ of the transitions that have as label the atom a of l:

- if the sign of l is positive, then τ reaches q_4 if $1 \in [n_1, n_2]$ and $1 \in [n_3, n_4]$;
- if the sign of l is negative, then τ reaches q_4 if $0 \in [n_1, n_2]$ and $0 \in [n_3, n_4]$.

The negative timed strings $\tau' \in S_-$ are strings of length 2 that reach state q_4 on if for all labels $a \in \Sigma$, the delay guards $[n_1, n_2]$ and $[n_3, n_4]$ of the transitions with label a contain both 0 and 1. Thus, the identification of the guards of this NFA $\mathcal{A}$ such that all positive examples reach q_4, and none of the negative reach q_4, represents an assignment of truth values to the variables of that 3-SAT instance that satisfies all clauses. We now present the formal proof.

Let $I = (V = \{v_1, \dots, v_n\}, C = \{c_1, \dots, c_m\})$ be a 3-SAT instance. We construct the following instance of our delay guard identification problem $(S = (S_+, S_-), \mathcal{A} = \langle Q, \Sigma, \Delta, q_0, F \rangle)$ ($\mathcal{A}$ is depicted in), where:

- S_+ contains a timed string $\tau_c = (a_1, t_1)(a_1, t_1)(a_2, t_2)(a_2, t_2)(a_3, t_3)(a_3, t_3)$ for every clause $c = \{l_1, l_2, l_3\} \in C$, where for all $1 \leq i \leq 3$, a_i is the atom variable of literal l_i, and $t_i = 1$ if l_i is a positive literal, $t_i = 0$ otherwise;
- S_- contains two timed strings $\tau_v = (a, 0)(a, 1)$ and $\tau'_v = (a, 1)(a, 0)$ for every variable $a \in V$;
- $Q = \{q_0, q_1, q_2, q_3, q_4\}$;
- $\Sigma = V$;

- $\Delta = \bigcup_{a \in V} \left\{ \begin{array}{c} \langle q_0, q_1, a\rangle, \langle q_1, q_0, a\rangle, \langle q_0, q_2, a\rangle, \langle q_2, q_3, a\rangle, \\ \langle q_2, q_4, a\rangle, \langle q_3, q_3, a\rangle, \langle q_4, q_4, a\rangle \end{array} \right\}$;
- $F = \{q_4\}$.

We now claim that there exists a delay guard for every transition $\delta \in \Delta$ such that the resulting DRTA is consistent with S if and only if the original 3-SAT instance is satisfiable.

($\Rightarrow$) Let $m : V \to \{\mathit{true}, \mathit{false}\}$ be a certificate for the 3-SAT instance I, i.e., setting all variables $v \in V$ to $m(v)$ makes I satisfied. Using m we create the following delay guards for the transitions of $\mathcal{A}$:

- for all $\langle q_0, q_1, a\rangle \in \Delta$ and all $\langle q_2, q_3, a\rangle \in \Delta$ create a guard $[t, t]$, where $t = 1$ if $m(a) = \mathit{true}$, $t = 0$ otherwise;
- for all $\langle q_1, q_0, a\rangle \in \Delta$, all $\langle q_3, q_3, a\rangle \in \Delta$, and all $\langle q_4, q_4, a\rangle \in \Delta$ create a guard $[0, 1]$;
- for all $\langle q_0, q_2, a\rangle \in \Delta$ and all $\langle q_2, q_4, a\rangle \in \Delta$ create a guard $[t', t']$, where $t' = 1$ if $m(a) = \mathit{true}$, $t' = 0$ otherwise.

All the transitions that directly lead to q_4 from the start state q_0 in the DRTA we just constructed have the same delay guard. Because of this, no negative example $\tau_v \in S_-$ can end in the final state q_4. Instead, they end in the non-final state q_3.

For every positive example $\tau_c \in S_+$, there are two transitions τ_c can fire at the start of the computation of $\mathcal{A}'$: a transition $\langle q_0, q_1, a\rangle$ to q_1, or a transition $\langle q_0, q_2, a\rangle$ to q_2. It fires a transition to state q_2 if either $m(a) = \mathit{true}$ and the first literal in c is a, or $m(a) = \mathit{false}$ and the first literal in c is $\neg a$. Otherwise τ fires a transition to state q_1.

Suppose the fired transition is to state q_2. Because the delay guard created for all transitions $\langle q_2, q_4, a\rangle$ to state q_4 is equal to the guards created for the transitions $\langle q_0, q_2, a\rangle$, the construction of τ_c guarantees that the next transition it fires is to state q_4. After reaching state q_4, the transitions $\langle q_4, q_4, a\rangle$ with delay guard $[0, 1]$ guarantee that it ends in q_4.

Suppose the fired transition is to state q_1. The next transition it fires is a transition $\langle q_1, q_0, a\rangle$ with delay guard $[0, 1]$ back to q_0. From q_0 it again fires a transition to either q_1 or q_2 depending on the value of $m(a)$ and whether the next literal is positive or not. Since there exists at least one literal in c that is satisfied by m, τ will at some point fire the transition to q_2. Thus, it will at some point end in q_4. Hence, all positive examples $\tau_c \in S_+$ end in q_4. Consequently, it holds that $S_+ \subseteq L(\mathcal{A}')$ and $S_- \subseteq L(\mathcal{A}')^{\mathrm{c}}$.

($\Leftarrow$) Let $\mathcal{A}'$ be a DRTA constructed from $\mathcal{A}$ by adding delay guards to transitions such that $S_+ \subseteq L(\mathcal{A}')$ and $S_- \subseteq L(\mathcal{A}')^{\mathrm{c}}$. Because $\mathcal{A}$ accepts all of the positive examples and none of the negative examples, the transitions $\langle q_0, q_2, a, [n_1, n_2]\rangle$ and $\langle q_2, q_4, a, [n_3, n_4]\rangle$ in $\mathcal{A}'$ (with the same label) are such that $0 \in [n_1, n_2] \cup [n_3, n_4]$ or $1 \in [n_1, n_2] \cup [n_3, n_4]$, but not $\{0, 1\} \in [n_1, n_2] \cup [n_3, n_4]$. We construct the following solution for the 3-SAT problem: for all $a \in V$, set $m(a) = \mathit{true}$ if $1 \in [n_1, n_2] \cup [n_3, n_4]$, set $m(a) = \mathit{false}$ if $0 \in [n_1, n_2] \cup [n_3, n_4]$. By construction of S_+, and since it holds that $S_+ \subseteq L(\mathcal{A}')$, m is a mapping that satisfies at least one

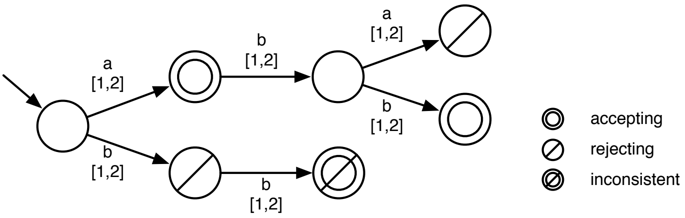


Figure 4.3: A timed APTA for the timed input sample: $(S_+ = \{(a,1),$ $(a,1)(b,2)(b,1),(b,2)(b,1)\}, S_- = \{(a,1)(b,1)(a,1),(b,2),(b,1)(b,1)\})$. The minimum delay $t_{\min}$ in the sample is 1, the maximum delay $t_{\max}$ is 2.

literal in every clause of the original 3-SAT problem. Hence, the 3-SAT instance is satisfiable.

The reduction function is clearly polynomial. Checking whether a DRTA is consistent can clearly be done in polynomial time by running the DRTA on every example from the input set. Hence the problem is a member of NP. This completes the proof. □

Despite the above theorem, we still want to identify both the delay guards and the structure of a DRTA. We will not be able to do so efficiently, but we might be able to *converge* efficiently to the correct DRTA when given more and more data, i.e., in the limit. In fact, we know we are able to do so by Theorem 5. In the following, we describe our algorithm for identifying a DRTA. The algorithm identifies a small DRTA in polynomial time from a timed input sample, and converges efficiently to the correct DRTA in the limit.

4.3.2 Timed state-merging and transition-splitting

Our DRTA identification algorithm, called RTI, is an EDSM algorithm that uses the red-blue framework (see Section 2.4.1). Most of the conventional state-merging routines are modified in order to deal with timed data and DRTAs. Moreover, the algorithm now contains a new routine, called *transition-splitting*, that enables it to identify the delay guards of a DRTA. This routine identifies time constraints by first assuming them to contain every time value, and then splitting these constraints into two new non-overlapping constraints for two new transitions. In the previous chapter, we described a one-clock deterministic timed automaton identification algorithm ID_1-DTA (Algorithm 3.1) that identifies clock guards by simply selecting the smallest consistent one. We use this splitting routine instead of this smallest first order because this routine makes it possible to use an evidence measure similar to the one used in the original EDSM algorithm. In fact, it is difficult to define a suitable evidence measure for the identification methods in the ID_1-DTA algorithm. In the following we discuss all of the changes we made to the original EDSM algorithm.

A timed APTA Like the conventional state-merging algorithm, RTI starts with an augmented prefix tree acceptor (APTA) $(\mathcal{A}, R)$, i.e., a prefix tree DRTA $\mathcal{A}$, augmented with a set of rejecting states R. In the conventional APTA, two strings end in the same state only if they are identical. Two timed strings are identical if at every index both their symbols *and* their time values are the same. It rarely occurs that an input sample S contains (prefixes of) timed strings that have exactly the same time values. Hence, were we to construct an APTA in the conventional way, we would with high probability obtain an automaton such that every state will be reached by only a single timed string from S. Starting from such an automaton, the conventional state-merging algorithm could be used to merge the delay guards of transitions into larger delay guards. In other words, we could identify the delay guards in a *bottom-up* way. However, like our ID_1-DTA algorithm, it is very difficult to come up with a good evidence value for such an bottom-up method. The conventional evidence value clearly fails since the determinization routine (see Section 2.4.1) will only combine states that are reached using the same symbol *and* the same time value(s). In other words, it will usually merge no states at all. Consequently, the evidence value is usually either 1 or 0, and hence it contains little information.

A bottom-up approach for identifying states and transitions of the conventional state-merging algorithm clearly creates some problems for identifying delay guards. Because of this, we identify the delay guards using a *top-down* approach, i.e., by initially setting them to be as general (large) as possible and specializing them (making them smaller) if necessary. The states and transitions are still identified using the conventional state-merging approach.

In our timed APTA (see Figure 4.3), two timed strings end in the same state if their *untimed* strings (the strings obtained by disregarding the time values) are identical. We set the initial values of the lower and upper bounds of all delay guards of the timed APTA to be the minimum $t_{\min}$ and maximum $t_{\max}$ time values from the input sample S, respectively. This timed APTA is identical to the conventional APTA constructed from the untimed versions of the timed strings from S. We show a timed version of the APTA construction in Algorithm 4.1.

Our timed APTA construction allows for the possibility of *inconsistent* states. These are created when the untimed versions of a positive and a negative example are identical. For example, in Figure 4.3, string bb is the untimed version of $(b, 2)(b, 1) \in S_+$ and $(b, 1)(b, 1) \in S_-$. Our algorithm can get rid of the aforementioned inconsistencies by *splitting* (specializing) a transition at some time value $t \in [t_{\min}, t_{\max} - 1]$.

Splitting a transition A *split* with time value t of a transition δ divides the part of the DRTA pointed to by transition δ into two parts. The first part is reached by the timed strings that fire δ with a delay value less or equal to t. The second part is reached by timed strings for which this value is greater than t. An example split is depicted in Figure 4.4.

The exact result of a split depends on which timed strings from the input sample fire δ. Every such string can be written as $\tau(a, t')\tau'$, where τ is the prefix before firing transition δ, (a, t') is a pair containing the symbol a of δ and a time value t that satisfies the delay guard $[n, n']$ of δ, and τ' is the suffix after firing δ. We call

Algorithm 4.1 Construct the timed APTA: tapta

```
Require: an input sample $S = \{S_+, S_-\}$ with alphabet $\Sigma$ and minimum and maximum
  delay values $t_{\min}$ and $t_{\max}$
Ensure: $(\mathcal{A}, R)$ is the timed APTA for $S$ (a DRTA $\mathcal{A}$ augmented with a set of rejecting
  states $R$)
  Set $\mathcal{A} := \langle Q = \{q_0\}, \Sigma, \Delta = \emptyset, q_0, F = \emptyset\rangle$
  Set $R := \emptyset$
  for each timed string $\tau = (a_1, t_1), \ldots, (a_n, t_n)$ from $S$ do
    Set state $q := q_0$
    for every index $0 < i \leq n$ of $\tau$ do
      if there exist no $\langle q, q', a_i, [n, n']\rangle \in \Delta$ for any $q'$ and $[n, n']$ then
        Add a new state $q'$ to $\mathcal{A}$
        Add a new transition $\langle q, q', a_i, [t_{\min}, t_{\max}]\rangle$ to $\mathcal{A}$
      end if
      Set $q := q'$
    end for
    if $\tau \in S_+$ then set $F = F \cup \{q\}$
    if $\tau \in S_-$ then set $R = R \cup \{q\}$
  end for
  Return $(\mathcal{A}, R)$
```

Algorithm 4.2 Splitting a transition: split

```
Require: an augmented DRTA $(\mathcal{A} = \langle Q, \Sigma, \Delta, q_0, F\rangle, R)$, a transition $\delta = \langle q, q', a,$
  $[n, n']\rangle$, a time value $t \in [n, n']$, and an input sample $S$
Ensure: $\delta$ is split at time $t$ and $\mathcal{A}$ is changed accordingly
  Remove $\delta$ from $\mathcal{A}$
  Remove the part of the APTA with $q'$ as root from $\mathcal{A}$
  Add two new states $q_1$ and $q_2$ to $\mathcal{A}$.
  Add a new transition $\delta_1 := \langle q, q_1, a, [n, t]\rangle$ to $\mathcal{A}$.
  Add a new transition $\delta_2 := \langle q, q_2, a, [t+1, n]\rangle$ to $\mathcal{A}$.
  Set $q_1$ to be the start state of $\mathcal{A}_1 :=$ tapta$(S^{\delta_1})$
  Set $q_2$ to be the start state of $\mathcal{A}_2 :=$ tapta$(S^{\delta_2})$
```

the timed string $\tau^\delta = (a, t')\tau'$ the *tail* of $\tau(a, t')\tau'$ for δ. Such a tail is said to be positive (negative) if $\tau\tau^\delta$ is positive (negative). We use S^δ to denote the subsample of all tails from S for δ, i.e., $S^\delta = (S^\delta_+ = \{\tau^\delta \mid \tau \in S_+\}, S^\delta_- = \{\tau^\delta \mid \tau \in S_-\})$. In a split the two divided parts of the DRTA are reconstructed using these tails. Because we use the red-blue framework, RTI can only split transitions that have blue states as their target. The splitting algorithm is shown in Algorithm 4.2.

Let δ be a transition to a blue node. Suppose S^δ contains two tails τ^δ_1 and τ^δ_2 that are equal if we disregard their time values. Initially, these two tails end in the same state due to the timed APTA construction. After a split of δ, however, it is possible that S^{δ_1} contains τ^δ_1 while S^{δ_2} contains τ^δ_2 (or vice versa). In other words, τ^δ_1 and τ^δ_2 no longer end in the same state. In this way, a split can remove both consistent and inconsistent states from a DRTA. Our algorithm decides where to split a transition based on the amounts of removed consistent and inconsistent states. This is explained further in Section 4.4.

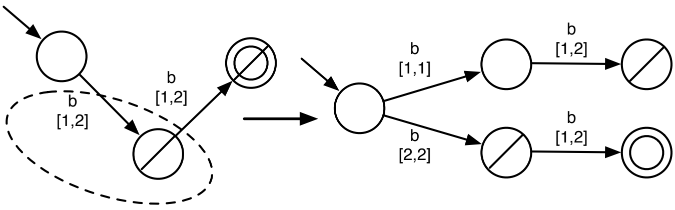


Figure 4.4: A split of a part of the APTA from Figure 4.3. On the left, the original DRTA is shown. The guard and target node that are to be split are surrounded by a dashed ellipse. On the right, the result of the split is shown. The split is called using time value $t = 1$.

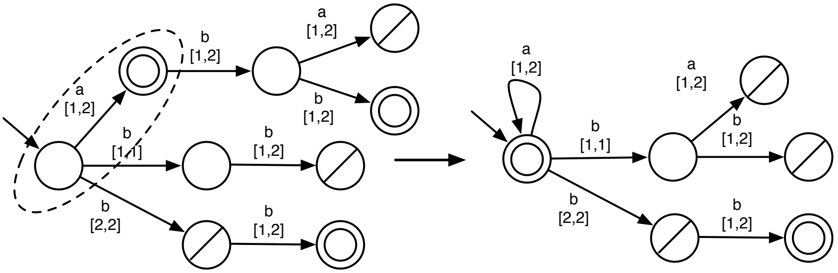


Figure 4.5: A merge of a part of the APTA from Figure 4.3 after the split from Figure 4.4. On the left, the original DRTA is shown. The states that are to be merged are surrounded by a dashed ellipse. On the right, the result of the merge is shown.

Merging states in a DRTA Besides the timed APTA construction and the split operation, our DRTA identification algorithm is still somewhat different from a conventional state-merging algorithm: the merge operation is modified in order to deal with the delay guards of a DRTA. Suppose that RTI wants to merge a blue state q and a red state q'. In the conventional state-merging algorithm, a new red state q'' will be created that has all the incoming and outgoing transitions of both q and q'. In the DRTA case, however, we cannot just use the outgoing transitions of both q and q' as the outgoing transitions for q'' because their guards can be partially overlapping. For example, suppose that a DRTA $\mathcal{A}$ contains the following transitions: $\langle q, q_1, a, [0, 10] \rangle$, $\langle q', q_2, a, [0, 4] \rangle$, and $\langle q', q_3, a, [5, 10] \rangle$, depicted in Figure 4.6. If RTI merges q and q' into a new state q'', the transitions of q'' will be: $\langle q'', q_1, a, [0, 10] \rangle$, $\langle q'', q_2, a, [0, 4] \rangle$, and $\langle q'', q_3, a, [5, 10] \rangle$. Thus, there is a non-deterministic choice for a delay value in $[0, 4]$, with targets q_1 and q_2, and there is a different non-deterministic choice for a delay value in $[5, 10]$, with targets q_1 and q_3. In fact, because the guard of the transition to q_1 overlaps with both of the other transitions, a call to the determinization routine will merge all three of the states q_1, q_2, and q_3 into one single state. In other words, the determinization

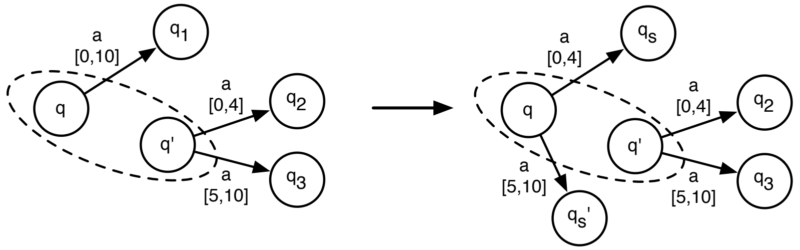


Figure 4.6: When merging two states q and q', first the outgoing transitions are split in order to resolve non-deterministic choices that are due to differences in delay guards.

routine also merges the deterministic choice with targets q_2 and q_3. We solve this issue by splitting $\langle q, q_1, a, [0, 10]\rangle$ into $\langle q, q_s, a, [0, 4]\rangle$ and $\langle q, q'_s, a, [5, 10]\rangle$ before merging q with q'. The guards of the outgoing transitions of q and q' are now identical and we can simply merge the two states. The determinization routine resolves the resulting non-determinism in the normal way.

Example 4.2. Figure 4.5 shows an example of a merge that includes a split operation before determinization. The second transition fired by $(a, 1)(b, 2)(b, 1)$ and $(a, 1)(b, 1)(a, 1)$ (labeled with b) is split first before starting the merge procedure. Notice that the accepting state gets merged with the bottom path (reached by the transition with guard $[2, 2]$) because it is reached by $(a, 1)(b, 2)(b, 1)$. The rejecting state gets merged with the top path because it is reached by $(a, 1)(b, 1)(a, 1)$.

In the red-blue framework, we can only merge a blue state with a red state. Consequently, the determinization routine only merges uncolored states with other states. Since we only split the outgoing transitions of red states, in any merge, all of the outgoing transitions of one of the two states is guaranteed to have the initial delay guard $([t_{\min}, t_{\max}])$. Hence, we can always solve the determinization issue by simply splitting these outgoing transitions at the values of the delay guards of the other state. Algorithm 4.3 shows the merge and determinization routines that include these splits.

4.3.3 The RTI algorithm for identifying DRTAs

In the previous section, we constructed routines for timed state-merging and transition-splitting. These routines can be inserted into the EDSM algorithm within the red-blue framework in order to construct an algorithm for identifying DRTAs. We call this algorithm RTI, which stands for Real-Time Identification. Algorithm 4.5 shows this algorithm. This algorithm tries all possible merges, splits, and colorings, in every iteration. It computes an evidence value (described in Section 4.4) for every possible action. The algorithm then performs the action that scores highest, but only if the result is not permanently inconsistent (Algorithm 4.4). An augmented DRTA is *permanently inconsistent* if either an inconsistency occurs in the red states, or if there exist an identical pair of a pos-

Algorithm 4.3 Merging two states: merge

```
Require: an augmented DRTA ($(\mathcal{A} = \langle Q, \Sigma, \Delta, q_0, F \rangle, R)$), two states $q, q'$ from $\mathcal{A}$ such
  that $q'$ is not red, and an input sample $S$
Ensure: $q$ and $q'$ are merged, $\mathcal{A}$ is updated accordingly, and true is returned, false is
  returned otherwise
  Add a new state $q''$ to $\mathcal{A}$ that is neither accepting nor rejecting
  if it holds that $q \in F$ or $q' \in F$, then set $F = F \cup \{q''\}$
  if it holds that $q \in R$ or $q' \in R$, then set $R = R \cup \{q''\}$
  for all outgoing transitions $\delta = \langle q', q^*, a, [n, n'] \rangle \in \Delta$ from $q'$ do
    if it holds that $n' \neq t_{\max}$ then call split($(\mathcal{A}, R), \delta, n', S$)
  end for
  for all transitions $\delta = \langle q_1, q_2, a, [n, n'] \rangle \in \Delta$ do
    if it holds that $q_1 = q$ or $q_1 = q'$ then set $q_1 := q''$
    if it holds that $q_2 = q$ or $q_2 = q'$ then set $q_2 := q''$
  end for
  while $\Delta$ contains two non-deterministic transitions $\langle q'', q_1, a, [n, n'] \rangle$ and $\langle q'', q_2, a,$
  $[n, n'] \rangle$ such that $q_2$ is not red do
    Boolean $b$ = merge($(\mathcal{A}, R), q_1, q_2, S$)
    if $b$ equals false then
      Undo the merge of $q$ with $q'$ and return false
    end if
  end while
```

Algorithm 4.4 Checking permanent inconsistency: inconsistent

```
Require: A timed input sample $S$ and an augmented DRTA ($\mathcal{A} = \langle Q, \Sigma, \Delta, q_0, \rangle, R$)
Ensure: Returns true if $(\mathcal{A}, R)$ can still be made consistent with $S$
  for all red states $q$ of $Q$ do
    if it holds that $q \in F$ and $q \in R$ then return false
    for all transitions $\langle q, q', a, [n, n'] \rangle \in \Delta$ such that $q'$ is not red do
      Let $S^{\delta} = (S^{\delta}_{+}, S^{\delta}_{-})$ be the set of tails of $S$ for $\delta$
      if $S^{\delta}_{+} \cap S^{\delta}_{-} \neq \emptyset$ then return false
    end for
  end for
  Return true
```

itive and a negative tail in the transitions to blue nodes. These tails can never be pulled apart by any subsequent split operation. If some actions score equally, the algorithm gives preference to first a merge, then a split, and finally a color operation.

It might be the case (and in fact it often is the case) that the evidence value gives the same score to a possible range of splits between two time values $t_1 < t_2$. We deal with such a situation by setting t equal to t_1. The motivation behind this is that if we should actually split it at some other time value $t < t' < t_2$, then the set of time values $[t + 1, t_2]$ is as large as possible. Hence, if at some later iteration of RTI there is some additional information (by performing merges and creating loops), then we can still identify the correct split later using as much information as possible. Setting t to $\lfloor \frac{t_1+t_2}{2} + 0.5 \rfloor$ might seem to make more sense, but this minimizes the additional information we can get in order to still identify the correct split if this split is incorrect. Note that incorrect splits can still be resolved by correct merges in subsequent iterations of RTI.

4.3.4 Properties of RTI

We now prove the following important properties of this algorithm: it is efficient, it is correct (sound), and it is complete. In addition, we show that under an appropriate evidence measure, the algorithm is a special case of the ID_1-DTA algorithm, and hence converges efficiently to the correct DRTA.

Algorithm 4.5 Identifying DRTAs: RTI

```
Require: A timed input sample $S$, with alphabet $\Sigma$, and minimum and maximum time
  values $t_{\min}$ and $t_{\max}$
Ensure: $\mathcal{A}$ is a small DRTA that is consistent with $S$
  Set $(\mathcal{A}, R) = (\langle Q, \Sigma, \Delta, q_0, F \rangle, R) = \mathsf{tapta}(S, \Sigma, t_{\min}, t_{\max})$
  Color the start state $q_0$ of $\mathcal{A}$ red
  while $\mathcal{A}$ contains non-red states do
    for all transitions $\langle q_r, q', a, [n, n'] \rangle \in \Delta$ such that $q_r$ is red and $q'$ is not do
      Color $q'$ blue
    end for
    for all transitions $\delta = \langle q, q_b, a, [n, n'] \rangle \in \Delta$ such that $q_b$ is blue do
      for all red states $q_r$ in $Q$ do
        Call $\mathsf{merge}((\mathcal{A}, R), q_r, q_b, S)$
        if $\mathsf{inconsistent}((\mathcal{A}, R))$ is *false* then
          Calculate the evidence value $v_m$
        end if
        Undo the merge of $q_r$ and $q_b$
      end for
      Let $S^\delta$ be the set of tails for $\delta$
      for all tails $\tau^\delta = (a, t)\tau' \in S^\delta$ do
        Call $\mathsf{split}((\mathcal{A}, R), \delta, t, S)$
        Calculate the evidence value $v_s$
        Undo the split of $\delta$
      end for
      Color $q_b$ red
      if $\mathsf{inconsistent}((\mathcal{A}, R))$ is *false* then
        Calculate the evidence value $v_c$
      end if
      Color $q_b$ blue
    end for
    if A merge has the highest evidence value $v_m$ then redo the merge
    else if A split has the highest evidence $v_s$ then redo the split
    else redo the coloring with the highest evidence value $v_c$
  end while
  Return $\mathcal{A}$
```

Proposition 4.3. *RTI is time efficient, i.e., it requires runtime polynomial in the size of the input sample S.*

Proof. RTI starts with the construction of a timed APTA $\mathcal{A}$. This construction (and the resulting APTA) is clearly polynomial in the size of the input. Then, in every iteration of RTI, either a transition is split, two states are merged, or a state is colored red in $\mathcal{A}$. The split operation requires three sets of timed strings S^{δ}, S^{δ}_1, and S^{δ}_2, which can all be constructed (or maintained in an efficient way using binary search trees) in polynomial time by running $\mathcal{A}$ on the input sample. The split operation uses two calls of the polynomial-time APTA construction in order to compute its result. The merge operation (including determinization) combines an amount of states that is bounded by the current size of $\mathcal{A}$. In doing so, it sometimes calls the polynomial split operation. Hence, the merge operation is also polynomial in the size of the input sample. Coloring a state only requires $O(1)$ time. This proves that each individual operation requires no more than polynomial time. What remains is to show that a single iteration of RTI requires no more than polynomial time and that the algorithm ends after a polynomial amount of iterations.

RTI only performs splits that separate the paths of two timed strings from the input sample. Hence, the amount of possible splits is polynomial in the size of the input sample, i.e., this amount is bounded by $|S| \times |S|$, where $|S| = \sum_{\tau \in S} |\tau|$. The amount of possible merges is bounded by the amount of possible pairs of states of the APTA, i.e., this amount is also bounded by $|S| \times |S|$. The number of timesRTI colors a state red is bounded by the amount of possible states, i.e., it is bounded by $|S|$. Thus, in a single iteration, RTI can in the worst case try $2|S| \times |S| + |S|$ possible polynomial operations before deciding which action to take.

Once a state is colored red, a pair of states has been merged, or two timed strings have been split, the same action cannot occur again. Since one of these actions is performed in every iteration, this bounds the amount of possible iterations of RTI by $2|S| \times |S| + |S|$. The proposition follows. □

Proposition 4.4. *RTI is correct (sound), i.e., given any input sample $S = (S_+, S_-)$, it returns a DRTA $\mathcal{A}$ that is consistent with S (such that $S_+ \in L(\mathcal{A})$ and $S_- \in L(\mathcal{A}^{\mathrm{C}})$).*

Proof. By definition S is such that $S_+ \cap S_- = \emptyset$. Thus, although initially it is possible that $\mathcal{A}$ is inconsistent with S, it is not permanently inconsistent. RTI can make $\mathcal{A}$ consistent with S by splitting transitions at the appropriate time values. In addition, a color or merge operation is only performed if $\mathcal{A}$ is not permanently inconsistent afterwards. Hence, during the execution of RTI, $\mathcal{A}$ is never permanently inconsistent. More specifically, there exists no red state q_r such that $q_r \in F$ and $q_r \in R$. The timed APTA construction ensures that every timed string $\tau \in S_+$ ends in a state $q \in F$, and that every timed string $\tau \in S_-$ ends in a state $q \in R$.

RTI terminates only if all states are colored red. Hence, when it terminates, every timed string $\tau \in S_+$ ends in a red state $q_r \in F$, every timed string $\tau \in S_-$ ends in a red state $q_r \in R$, and there exists no red state q_r such that $q_r \in F$ and $q_r \in R$. In other words, when it terminates, $\mathcal{A}$ is consistent with S. Because the algorithm terminates after a polynomial number of iterations (Proposition 4.3), the proposition follows. □

Proposition 4.5. *RTI is complete, i.e., for any DRTA languages L_t, there exists a data sample S such that RTI returns a DRTA $\mathcal{A}$ with $L(\mathcal{A}) = L_t$.*

Proof. For any DRTA language L_t, there exists a DRTA $\mathcal{A}_t$ such that $L_t = L(\mathcal{A}_t)$. We show that RTI is correct by proving that it performs actions such that the following invariant holds: the red states and the transitions between them are correct, i.e., $\mathcal{A}_t$ contains all of the red states and transitions.

For the initial case, the start state q_0 of the timed APTA $\mathcal{A}$ is colored red. $\mathcal{A}$ also contains a start state. For the arbitrary case, assume for the sake of induction that at the start of the current iteration $\mathcal{A}_t$ contains all the red states in $\mathcal{A}$ as well as all the transitions between them. We distinguish three cases:

Case 1. One of the outgoing transitions of a red state in $\mathcal{A}_t$ has a different delay guard in $\mathcal{A}$;

Case 2. The first case does not hold and there exists a transition between two of those red states in $\mathcal{A}_t$, but not between the red states in $\mathcal{A}$;

Case 3. None of the above holds.

In case 1, RTI should identify a delay guard for this transition δ. Note that, when some delay guard is incorrect, there always exists at least one transition such that only the upper bound is incorrect. We assume without loss of generality that the upper bound of the delay guard of δ is correct. Since δ is a reachable transition, there exist timed strings that fire $\delta = \langle q, q', a, [n, n'] \rangle$ in $\mathcal{A}_t$. Moreover, there exist timed strings that fire the transition $\delta' = \langle q, q'', a, [n'+1, n''] \rangle$ in $\mathcal{A}_t$. Since all the red states in $\mathcal{A}$ are correct so far, all of these timed strings currently fire a transition to a blue state $\delta'' = \langle q, q''', a, [n, n''] \rangle$ in $\mathcal{A}$. Within the set of timed strings that can fire δ'' in $\mathcal{A}$, there exist two that are of the form: $\tau(a, n')\tau'$ and $\tau''(a, n'+1)\tau'''$. When these two strings are included in the input sample S, RTI will try to split the transition δ'' with time value n', resulting in the correct delay guard for δ. We left the precise definition of the evidence value open, but clearly there exist timed strings such that the evidence value is highest when the correct guard is identified. These timed strings can be included in S in order to force the correct identification. These correct identifications can be iterated (a finite number of times) until every outgoing transition of a red state has the correct delay guard.

In case 2, RTI should identify a new transition $\delta = \langle q, q', a, [n, n'] \rangle$ between two red states. It does so by merging a blue state with a red state. Like before, there exist timed strings that fire $\delta' = \langle q, q'', a, [n, n'] \rangle$ in $\mathcal{A}$, where q'' is a blue state. Because RTI tries all possible merges, there exists an evidence value such that this blue state q'' will be merged correctly with the red state q'. Moreover, there exist no timed strings that create a permanent inconsistency if this merge is performed since this merge creates a transition that is a part of the correct DRTA $\mathcal{A}_t$.

In case 3, RTI should color a blue node red. Because the previous two cases do not hold, any coloring of a blue state that does not result in a permanent inconsistency is a correct identification of a transition and a red state.

We conclude that if all red states and the transitions between them are correct, then it is possible that, given the right data, after an iteration of RTI, the red states and transitions between them are still correct. By induction, all the red states and transitions between them can remain correct during the entire execution of RTI. Since RTI has to identify a finite number of correct delay guards, transitions, and states, and RTI can identify one of these in every iteration, RTI ends after a finite number of iterations. States are added to the set of final states if there exists a positive example that ends in that state. Clearly, for every final state there exists some example that ends in it. Hence, given the right data, the final states are also identified correctly. In conclusion, the result $\mathcal{A}$ of RTI is such that $L(\mathcal{A}) = L_t$ after a finite number of iterations. This proves the proposition. □

Proposition 4.6. *RTI converges efficiently in the limit to a correct DRTA $\mathcal{A}_t$.*

Proof. RTI is efficient (Proposition 4.3) and complete (Proposition 4.5). Thus, we only need to show that there exists an evidence value and polynomial characteristic sets that force the algorithm to converge to the correct DRTA. We prove this defining an evidence value that makes RTI simulate ID_1-DTA Algorithm 3.1 in the case that the target language is a DRTA language.

Our evidence value is such that it gives a higher score to operations that are first in the order of the ID_1-DTA algorithm. More specifically, it gives highest values to first merge, then color, then split operations. In addition, higher values are given to operations on transitions to blue states that are reachable by smaller (by length and alphabet) timed strings.

Using such an evidence value, splits are considered only if the transition to a blue state that RTI is currently trying to identify cannot be merged or colored. Since we want RTI to simulate ID_1-DTA, the evidence value should give the highest score to the smallest consistent split, where consistent is defined in terms of ID_1-DTA. More specifically, the evidence value gives preference to splits that result in a new blue state that can be merged or colored. In addition, since ID_1-DTA identifies transitions with larger constants in the clock guards first, the highest value is given to the split such that this blue state is the target of the newly created transition with the highest constants in its clock guard.

The merge and color operations of RTI correspond to the identification of the target state in ID_1-DTA. In both algorithms, these operations are only performed if the result is not permanently inconsistent.

Using the evidence value we just described, RTI identifies the exact same transitions as ID_1-DTA in the case that the clock of the 1-DTA should be reset at every transition. Moreover, they do so in the exact same order. Thus, when the correct 1-DTA is a DRTA, then the two algorithms result in the exact same DRTAs. Since ID_1-DTA identifies the full class of 1-DTAs efficiently in the limit, this proves that RTI converges efficiently to the correct DRTA (given the right evidence value). □

In conclusion, RTI, which is a timed version of EDSM, runs in polynomial-time, is a correct and complete algorithm, and converges efficiently to the correct DRTA in the limit. A big difference between EDSM and RTI is that RTI deals with timed data. As a consequence, RTI requires a different evidence measure than the one used by EDSM, which is based on non-timed data. In the next section, we provide a few of these measures.

4.4 Heuristics for RTI

Our algorithm uses an evidence (score) value (measure) in order to determine which action to take. The action that results in the highest evidence value, i.e., the one that agrees most with the input sample, is selected to be performed. Thus, the evidence value can be thought of as a *heuristic* that guides RTI.[1] Since RTI stops when it has reached a local optimum, i.e., when it cannot perform any actions, RTI can be thought of as a greedy procedure. The result is a small, hopefully the smallest, DRTA that is consistent with the input sample.

In this section, we describe the four heuristics that we use in our experiments. In addition, we explain a simple search variant of RTI. This search procedure uses the size of the DRTA resulting form an entire run of RTI as an evidence value. We compared the performance of this search variant in our experiments in order to give some insight into the effects of searching for a smaller solution.

4.4.1 Four evidence values

Since RTI is based on the EDSM algorithm for the identification of DFAs, we also based all of our heuristics on the EDSM evidence value. The intuition behind the EDSM evidence value is that the labels of the states that are combined during the determinization procedure of a merge can be seen as statistical tests for testing whether the merge is good or bad. A good merge is one where the states are instances of the same state in the target DFA, in a bad merge they are instances of different states. In the case of a good merge, none of the labels of the states that are merged by the determinization procedure can result in an inconsistency. In the case of a bad merge, some of these labels can result in an inconsistency. Thus, the greater the number of merged labeled states, the more confident we are that the merge is a good merge.

In addition to the normal non-timed (EDSM) evidence, RTI has access to information in the form of time values. Naturally, we would like RTI to make use of this timed information. Moreover, we would like to show that using this additional information leads to improved performance.

We created four heuristics that make use of the EDSM evidence value and the additional time information in different ways. We now first explain the two simple ones that do not make use of time information, then one that does make use of time information by giving more power to tails that are closer together, and finally one that tries to penalize the evidence value based on the amount of splits necessary in order to remove all inconsistencies.

Pure EDSM In order to give more insight into whether it is beneficial to make use of the time information in the input sample, we included the exact score value used by EDSM in our experiments. The evidence value used by EDSM is defined

[1] In contrast, in the proof of Proposition 4.6 we constructed a fixed order evidence value. Such an evidence value makes it easy to prove theoretical results, but in practice it will not perform well. In practice, there can be a lot more evidence then permanent inconsistencies. Moreover, permanent inconsistencies are created by very specific pairs of timed strings. These strings will only occur in very large input samples.

as:

$$\mathsf{pure} = \#\mathsf{merge}(pos, pos) \ + \ \#\mathsf{merge}(neg, neg)$$

where $\#\mathsf{merge}(pos, pos)$ (or $\#\mathsf{merge}(neg, neg)$) is the number of merges of pairs of positive (or negative) states performed by the determinization procedure.

Consistent EDSM A big difference between EDSM and our DRTA identification algorithm is that in our case, we allow for the possibility of inconsistent states. The split operation can be used to remove such inconsistencies. Intuitively, these inconsistencies should of course have some negative influence on the evidence value. We included a variant of EDSM that uses the inconsistencies in a very simple way: it simply subtracts the number of inconsistent merges from the number of consistent ones. It is defined as:

$$\mathsf{consistent} = \#\mathsf{merge}(pos, pos) \ + \ \#\mathsf{merge}(neg, neg) \ - \ \#\mathsf{merge}(pos, neg)$$

When using this evidence measure it is possible that a color operation gets the highest score. This occurs when all possible splits and merges score negatively. This is different from conventional EDSM within the red-blue framework where a color operation is only applied when no merge is possible. Our algorithm simply picks the highest scoring operation, including color operations. If some operations score equally, we give preference to first merge, then split, and finally color operations.

Impact EDSM The first timed measure we use is based on the idea that tails that lie far away from each other are more likely to be pulled apart by a future split operation than tails that lie close to each other. For example, suppose that RTI merges two states in a DRTA, each containing just one tail (each state is reached by just one timed string). Let these tails starting from these two states be something like: $(a, 1)(b, 3)(c, 5)$ and $(a, 2)(b, 3)(c, 4)$. These tails lie close to each other in time and should intuitively get a higher impact on the score than say: $(a, 1)(b, 3)(c, 5)$ and $(a, 5)(b, 6)(c, 7)$.

We calculate this impact value using the distance in time between every pair of tails τ and τ' that are identical if we disregard their time values, i.e., if $\mathsf{untime}(\tau) = \mathsf{untime}(\tau')$. These two tails currently end in the same state in the augmented DRTA $(\mathcal{A} = \langle Q, \Sigma, \Delta, q_0, F\rangle, R)$. We define the impact of τ and τ' as the probability that τ and τ' still end in the same state if we were to choose a split point uniformly at random in every non-red transition. Since RTI cannot split transitions between two red states, these transitions are excluded from this definition. The impact of two tails is calculated using Algorithm 4.6. The evidence value we use is computed as the sum over all timed strings from S of the maximum impact with any other consistent timed string minus the maximum impact with any other inconsistent timed string. All pairs of timed strings that end in a red state get an impact of 1.0 because they can never be separated by any split. Let Q_r denote the red states of Q, and let Δ_b denote the transitions to blue states of Δ. S^δ denotes the sample of tails (suffixes) of timed strings from S that fire δ. Using these sets, the evidence value is:

Algorithm 4.6 The probability of not separating two timed strings: IMPACT

```
Require: Two tails $\tau$ and $\tau'$ for a transition to a blue state $\delta$, and the minimum and
  maximum time value $t_{\min}$ and $t_{\max}$
Ensure: Returns the probability that $\tau$ and $\tau'$ end in the same state if we split all
  non-red transitions uniformly at random
  Set probability $p := 1.0$
  for all integers $1 \leq i \leq |\tau|$ do
    Let $t_1$ and $t_2$ be the $i$th time values in $\tau$ and $\tau'$ respectively
    Set $p := p \times (1.0 - \frac{|t_1 - t_2|}{t_{\max} - t_{\min}})$
  end for
  Return $p$
```

$$\begin{aligned}\mathsf{impact} = \sum_{q \in Q_r} \mathsf{pure}(q) - &\sum_{\delta \in \Delta_b} \max\{\mathsf{impact}(\tau, \tau') \mid \tau \in S_+^\delta \text{ and } \tau' \in S_-^\delta\} \\ + &\sum_{\delta \in \Delta_b} \max\{\mathsf{impact}(\tau, \tau') \mid \tau \neq \tau', \tau \in S_+^\delta \text{ and } \tau' \in S_+^\delta\} \\ + &\sum_{\delta \in \Delta_b} \max\{\mathsf{impact}(\tau, \tau') \mid \tau \neq \tau', \tau \in S_-^\delta \text{ and } \tau' \in S_-^\delta\}\end{aligned}$$

Splits EDSM Like our second evidence value, our last evidence measure tries to include the inconsistencies in an EDSM-like score. However, it tries to do so in a way that is a bit more sophisticated than simply subtracting the amount of inconsistencies. The main idea is that, since the EDSM evidence value does not include inconsistencies, we should first get rid of all inconsistencies, and then use the EDSM evidence measure on the DRTA without inconsistencies. We remove these inconsistencies by splitting the transitions of the timed APTA.

The size of the resulting consistent APTA, and hence the resulting EDSM score, is determined by which and how many splits the algorithm performs. Hence, we would like to determine where to split the APTA in an optimal way. Unfortunately, the hardness proof for determining the correct delay guards (the proof of Theorem 7) can easily be adapted to show that this problem is NP-hard. Thus, we cannot hope to solve this problem optimally every time we need to compute an evidence value. Instead, we use a simple approximation of this problem using Algorithm 4.7. This algorithm computes the minimum amount of times we have to split a single transition in order to make one node of the APTA consistent. We use this algorithm to compute the following recursion for every state q:

$$\#\mathsf{splits}(q, S) = \begin{cases} \max(\mathsf{splits}(q, S, \mathcal{A}),\ \#\mathsf{splits}(q', S)) & \text{if } q \text{ is not red} \\ 0 & \text{otherwise} \end{cases}$$

where q' is the source state of the transition to q, i.e., q' is such that there exists a transition $\langle q', q, a, [n, n] \rangle$ in $\mathcal{A}$. The recursion computes the number of splits required to make a state consistent for every state of an APTA. Intuitively, we want this measure to reflect how many states a state q' will be split into if we

Algorithm 4.7 Making a state consistent with splits: splits

```
Require: A state non-red $q$ of a colored DRTA $\mathcal{A}$ and a timed input sample $S$
Ensure: Returns an approximation of the minimum amount of times we need to split a
  transition in order to make $q$ consistent
  Let $\delta$ be transition to the root of the APTA that contains $q$
  Let $S^q = (S^q_+, S^q_-)$ denote the subsample of $S^\delta$ of tails for $\delta$ that end in $q$
  for all integers $1 \leq i \leq n$, where $n$ is the length of the tails of $S^q$ do
    Let $T_+$ denote the set of $i$th time values in tails from $S^q_+$
    Let $T_-$ denote the set of $i$th time values in tails from $S^q_-$
    if $T_+ \cap T_- = \emptyset$ then
      Set $s := 0$
      for all time values $t \in T_+$ do
        If $\min(\{t' \in T_- \mid t' > t\}) < \min(\{t' \in T_+ \mid t' > t\})$ then set $s := s + 1$
      end for
      for all time values $t \in T_-$ do
        If $\min(\{t' \in T_+ \mid t' > t\}) < \min(\{t' \in T_- \mid t' > t\})$ then set $s := s + 1$
      end for
    end if
  end for
  Return the minimum value for $s$
```

were to remove all inconsistencies from $\mathcal{A}$. If the parent state q of q' requires more splits than q', we assume that q' will also be split into the number of states q will be split in. This is why we define this measure as the maximum of splits(q', S) and #split(q, S). A red state requires no splits and will also not be split as the consequence of any other split we need to perform. Thus, the measure is 0 for red states.

Given the amount splits$(q, S, \mathcal{A})$ of states that q' will be split into, it is straightforward to compute the number of remaining merges of pairs of positive and negative states, i.e., the amount of remaining consistent merges: simply subtract splits$(q, S, \mathcal{A})$ from the original amount of consistent merges of states. Because the number of inconsistent merges in q can be greater than the number of consistent ones, the resulting value can become less than 0. However, the number of remaining consistent merges can of course never become less than 0. Therefore, we use 0 as the minimum value. For the complete augmented DRTA $(\mathcal{A} = \langle Q, \Sigma, \Delta, q_0, F \rangle, R)$, with input sample S_+, the evidence value then becomes:

$$\mathsf{splits} = \sum_{q \in Q} \max(\mathsf{pure}(q) - \#\mathsf{split}(q, S, \mathcal{A}), 0)$$

where pure(q) is the pure evidence value for state q. This split evidence value does not compute the actual EDSM evidence that remains if we were to split our DRTA in an optimal way, but it tries to approximate this value (from above).

4.4.2 A simple search procedure

In addition to these four heuristics, we tested a simple search process that we wrapped around RTI in order to test how much we can increase the performance

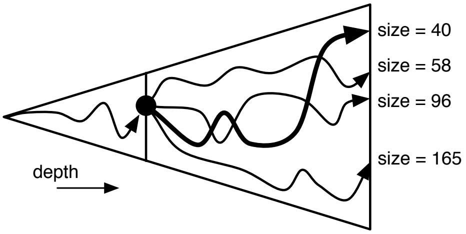


Figure 4.7: The search tree of our simple search process depicted as a triangle, the search depth increases from left to right. From the current node of the search tree, the search procedure computes a complete execution of RTI for every possible action, and chooses the action that results in the smallest DRTA (size = 40).

by searching. Our search process uses an evidence measures in an indirect way (see Figure 4.7):

- For every possible action (split/merge), the search procedure first computes a complete (greedy) execution of RTI with an evidence measure. The result is a DRTA $\mathcal{A}$.

- The search process then chooses the action that results in the smallest DRTA, i.e., such that the number of transitions in $\mathcal{A}$ is smallest.

The intuition behind this process is that actions taken by RTI that lead to a small DRTA are more likely to be correct. More specifically, although an evidence measure m assigns a small value to a specific action a, if a is in fact a good (correct) action, then the result of performing a and then using m to determine the subsequent actions can still lead to a better result than an action with a high value. In other words, the action that scores highest using m is not always the action that leads to the best result after a complete greedy run using m. Like our original algorithm, once the algorithm has chosen an action, it cannot be changed by subsequent iterations. The main benefits of this simple search procedure compared to a search procedure that uses an evidence value directly are:

- The procedure is an anytime algorithm. We can remember the smallest resulting DRTA and return it if the process is interrupted.

- The procedure produces results that are at least as good as the non-search (greedy) algorithm.

- The procedure requires only polynomial time in the size of the input sample: There are at most a polynomial amount of possible actions and the non-search algorithm is a polynomial time algorithm.

- The procedure ends without having to search through all possible DRTAs smaller than the result, i.e., it is incomplete but efficient.

Although our search procedure is a polynomial time algorithm, it takes a lot more time than the execution of our original algorithm since it can call this original algorithm tens of thousands of times as a subroutine.

4.5 Experiments

In this section, we experimentally compare each of the different evidence measures (including the search procedure) for RTI described in the previous section. In addition, we compare all of these implementations of RTI with a sampling approach that first samples the timed data and then runs EDSM to obtain a DFA representation of a DRTA. We perform the experiments on a large set of artificial data-sets generated from a randomly generated DRTA. In order to give insight into which method performs best in which setting, we generated these DRTAs with different number of states, transitions, delay guards, time values, etc. We provide plots of the performance of RTI and the sampling approach for each of these settings.

This section is structured as follows. We first describe the sampling approach in Section 4.5.1. Then, we explain the experimental setup and provide the algorithms we used to produce the artificial data-sets in Section 4.5.2. We list our expectations with respect to these experiments in Section 4.5.3. Finally, we show, describe, and explain our results in Section 4.5.4.

4.5.1 Sampled finite automata

As mentioned in the introduction, it is possible to sample timed data into an equivalent untimed format. Suppose we have a timed string $\tau = (a_1, t_1) \dots (a_n, t_n)$. The equivalent untimed format for this timed string is a string $s_1 a_1 \dots s_n a_n$, where $s_i = \circlearrowright\circlearrowright \dots \circlearrowright$ is a string consisting of special time tick symbols $\circlearrowright$ of length t_i.

Similarly, there also exists an equivalent DFA representation for any DRTA language. In fact, there exists a DFA representation for any DTA language. Given a DTA, a DFA representation can be constructed using the region construction method (see Section 2.3.2). Essentially, this method creates additional states for every possible combination of a state and a time value. The same state with a higher time value can be reached by a series of transitions labeled with the time tick symbol $\circlearrowright$. In Figure 4.8, we give the DFA that results from the region construction when applied to the DRTA of Figure 4.1.

Using the region construction it is always possible to construct a DFA that accepts exactly the same language as a DRTA. Hence, is never really necessary to identify a DRTA, there always exists a DFA that recognizes exactly the same language. The downside to trying to identify this DFA is that it is exponentially larger in the size of the DRTA. More specifically, since all constants of a DRTA are written down in binary, the number of states is exponential in the size of these constants. If we want the DFA returned by a DFA identification algorithm to be correct, we naturally require that every state in this DFA is reached by at least one string from the input data. Hence, using a DFA identification algorithm in order to identify a DRTA language is inefficient since it requires an exponential amount of data in order to return the correct result. In other words, it requires exponential time and space in order to converge. In contrast, by Proposition 4.6,

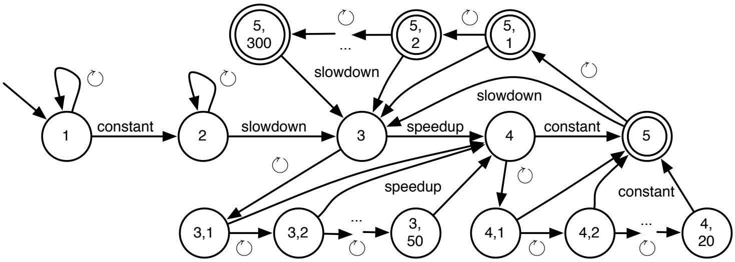


Figure 4.8: A DFA equivalent to the harmonica DRTA of Figure 4.1.

it is possible to converge to the correct DRTA using only a polynomial amount of data.

In theory, identifying a DRTA model for a DRTA language seems to be a better idea than identifying a DFA model for a sampled DRTA language. However, in practice one may argue that it makes no sense to sample the timed data using single time ticks. If the timed data has millisecond precision, it probably does not hurt to sample the timed data every 100 or even 1000 time ticks. Hence, in practice, the sampling transformation will usually contain an additional argument r that denotes the *sampling rate*. The result of sampling a timed string $\tau = (a_1, t_1)(a_2, t_2)(a_3, t_3) \ldots (a_n, t_n)$ with rate r is a string $s_1 a_1 s_2 a_2 s_3 a_3 \ldots s_n a_n$, where $s_i = \circlearrowright\circlearrowright \ldots \circlearrowright$ is a string of length $\lfloor \frac{t_i}{r} + 0.5 \rfloor$. Using such a sampling rate, it is of course possible that the region construction no longer works. In other words, it is possible that there exists no equivalent DFA for a DRTA when the timed strings are sampled with rate r. However, the blowup in automaton size is not so great when this rate r is used (the blowup is exponential in $\frac{c}{r}$). In practice, it is not a big problem that the resulting DFA is not equivalent to the actual DRTA. We are usually satisfied when it only approximates (in terms of a high percentage of overlap in accepted timed strings) the actual DRTA.

In the next section, we describe how we compare the RTI algorithm with the approach of first sampling the timed data using some fixed frequency and then using a DFA learning algorithm.

4.5.2 Experimental setup

In order to test our DRTA identification algorithms, we generate data using the process shown in Algorithms 4.8 and 4.9. We first generate a DRTA uniformly at random (Algorithm 4.8) and then we use this DRTA to generate timed strings uniformly at random (Algorithm 4.9). The length of these timed strings follows an exponential distribution with an average length of 10.

We generate random DRTAs with 4, 8, 16, and 32 states and alphabets of sizes 2, 4, and 8. To each of these DRTAs, we apply the split routine 4, 8, 16, and 32 times at random time values in order to create clock guards. The minimum time value of the clock guards is set to 0. The maximum time values of the clock guards is either 100 or 1000. Every state of the DRTA has a chance of 0.5 to be

Algorithm 4.8 Generating a random DRTA: rand_drta

```
Require: Values for the amount of states $n$, splits $m$, possible time values $o$, and the size of the alphabet $p$.
Ensure: $\mathcal{A}$ is a random DRTA for the specified parameters.
  $Q := \{q_0, q_1, \ldots, q_{n-1}\}$, $q_n$ is a garbage state.
  $\Sigma := \{a_1, a_2, \ldots, a_p\}$.
  $\Delta := \bigcup_{0 \leq i < n} \bigcup_{1 \leq j \leq n} \{\langle q_i, q_n, a_j, 0 \leq x \leq o \rangle\}$.
  $F := \emptyset$.
  for $1 \leq i \leq m$ do
    $\delta \in \Delta$ is a transition chosen uniformly at random.
    $t \in [0, o]$ is a time value chosen uniformly at random.
    Call split$(\delta, t)$.
  end for
  for every transition $\delta = \langle q, q_n, a, g \rangle \in \Delta$ do
    $q' \in Q$ is a state chosen uniformly at random.
    Set $\delta = \langle q, q', a, g \rangle$.
  end for
  while $F = \emptyset$ or $F = Q$ do
    for every state $q \in Q$ do
      Set $F := F \cup \{q\}$ with probability 0.5.
    end for
  end while
  Return $\mathcal{A} = \langle Q, \Sigma, \Delta, q_0, F \rangle$
```

an accepting state. We disallow the case that all or none of the states were chosen to be accepting.

From these DRTAs, we randomly generate input samples consisting of either 1000 or 2000 timed strings. These input samples are used as data-sets for our algorithms. We also generate test-sets consisting of 50000 newly generated timed strings. Each of these timed strings has a probability of 0.1 to stop in each state it visits. Whether a string is positive or negative was determined by the state it ended in.

For every unique combination of the parameters of the DRTA generating algorithm (Algorithm 4.8), we generated 10 different DRTAs. This resulted in $4 \times 3 \times 4 \times 2 \times 2 \times 10 = 1920$ data- and test-sets. In order to compare RTI with a sampling approach, we sampled these data- and test-sets using a fixed sampling size of 10. We replaced every symbol-time value pair (a, t) with an a symbol and $\frac{t}{10}$ special time increase symbols. Rounding was used to get rid of fractions. We used a sampling size of 10 for both the 100 and 1000 maximum values of the clock guards.

We run RTI on the constructed data-sets, and used the test-set to evaluate its performance. For the search process, we used the consistent EDSM measure because it is simple to compute, and because it turned out to perform not (much) worse than the timed evidence measures. The algorithm we use for identifying a DFA from the sampled data is the red-blue algorithm, which we downloaded from the Abbadingo web-site.[2] In the following, we first present the expected results,

[2] Abbadingo One: DFA Learning Competition: http://abbadingo.cs.unm.edu/.

Algorithm 4.9 Generating an input sample

```
Require: Values for the amount of states $n$, splits $m$, possible time values $o$, stopping
  probability $s$, and the sizes of the alphabet $p$ and the input sample $v$.
Ensure: $S$ is an input sample of size $v$ for the identification problem of a random DRTA
  for the specified parameters.
  Call $\mathcal{A} = \langle Q, \Sigma, \Delta, q_0, F \rangle = \text{RAND\_DRTA}(n, m, o, p)$.
  $S := (S_+ = \emptyset, S_- = \emptyset)$
  for $1 \leq i \leq v$ do
    $q := q_0$.
    $\tau := \lambda$.
    while true do
      break with probability $s$.
      $a \in \Sigma$ is a symbol chosen uniformly at random.
      $t \in [0, o]$ is a time value chosen uniformly at random.
      $\delta = \langle q, q', a, g \rangle \in \Delta$ such that $g$ is satisfied by $t$.
      $q := q'$.
      $\tau := \tau(a, t)$.
    end while
    if $q \in F$ then
      $S_+ := S_+ \cup \tau$.
    else
      $S_- := S_- \cup \tau$.
    end if
  end for
  Return $S$
```

then we provide and discuss the actual results of these algorithms.

4.5.3 Expectations

The main goal of the experimental setup described in the previous section, is to determine whether RTI performs better than the sampling approach. Because sampling results in a blow-up in both the amount of data and the size of the resulting automaton, we expect that RTI will perform better. In addition, the sampling transformation results in a small loss of information (otherwise there would be an even greater blow-up). However, since the sampling approach is a straightforward application of the current state-of-the-art in DFA identification, we expect that this approach also performs reasonably well.

With respect to the four different evidence values, we expect that both the splits and impact EDSM values will perform best because they make use of the time values of timed strings. The pure and consistent values do not consider these values. Between the two of them, we expect the consistent value to perform better because it also uses the inconsistent merges as information. We do not know what to expect regarding the performance difference of the impact and splits values. The splits value is more similar to the traditional EDSM measure. The impact value uses a distance measure between timed strings to determine their impact on the score. Timed strings (with identical symbols) that are closer together in time have a higher probability of ending in the same state. Both values make sense, we leave it to the experiments to show which one is superior.

When increasing the amount of delay guards (splits) of the original DRTA that was used to produce the data-sets, we expect the decrease in performance of the timed evidence values to be less than the non-timed evidence values. This makes sense because additional splits cause more differences in behavior between timed strings with the same symbols, but different time values. Similarly, when increasing the amount of states, we expect the non-timed evidence values to decrease more slowly than the timed ones, because the behavior is then not influenced by differences in time values. The same holds for additional symbols in the alphabet.

We expect that all methods perform better when more data is available, and that all perform worse when more time points are used. The search version of RTI should perform best, since this one runs a greedy version of RTI many times in order to decide between a single merge or split. We hope that, using the search procedure, we will be able to identify DRTAs that are sufficiently large enough for practical applications.

4.5.4 Results

We run RTI on all of the 1920 data-sets using each of the different evidence values. In addition, we run the sampled and the search approaches on these data-sets. We use two indicators for the performance of the DRTAs (or DFAs in case of the sampling method) resulting from these runs. The first is the size, measured in number of transitions of the DRTA (DFA), which we like to be as small as possible. The second is the percentage of correctly classified timed strings of the test-set, which we like to be as large as possible. In Figure 4.9, we show both of these performance indicators for every method over all data-sets as *boxplots*. The plots show the lower quartile, median, and upper quartile as a box. This box covers the entire *inter-quartile range*, i.e., the middle fifty percent of all data values. In addition, the *whiskers* (dashed lines) extend to the largest and smallest values that have a distance of at most 1.5 of the inter-quartile range from the edges of the box. Any values outside of these whiskers are considered *outliers* and are plotted using dots. Around the median of each boxplot, there is a small triangular cut in the box, called a *notch*. This notch depicts (roughly) the 95% confidence interval of the median. Thus, if the median of another boxplot is outside of this range, then the median of these two plots differ significantly. All of the plots in this section were made using the R statistical package.[3]

From Figure 4.9, it is clear that the sampling method performs much worse (in both performance measures) than our DRTA identification algorithm. In addition, the search procedure does help to increase the performance of RTI (again in both measures). Furthermore, the consistent evidence value results in smaller DRTAs on average than any of the other values. We used a paired t-test for independence to test the significance of this difference between the consistent and splits evidence values. The result of this test is a p-value of less than 2.2×10^{-16}, which is the minimum p-value used in the R package. Thus, this difference is significant. On average, the consistent evidence value finds DRTAs that contain 36 transitions less than the splits evidence value.

[3] See http://www.r-project.org/.

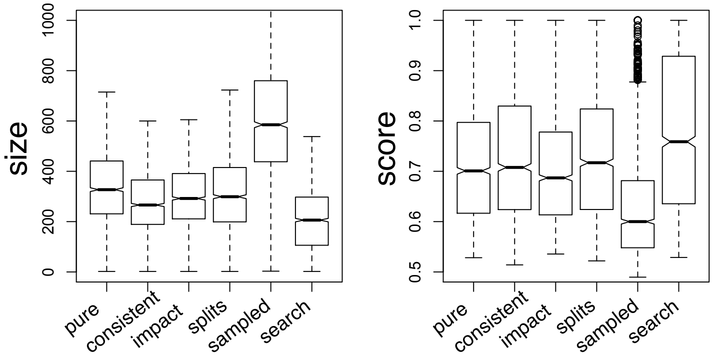


Figure 4.9: Boxplots of the sizes and scores of every method over all data-sets.

It seems that the score of the consistent value is smaller on average than the score of the splits value, but this score difference is not significant: a paired t-test applied to these score values results in a p-value of about 0.666. The pure and impact values clearly perform the worst of the four evidence values. The score of the impact value is even worse than the very simple pure evidence value. Thus, unfortunately, we can conclude that the impact value does not use the time information in a very good way.

Figure 4.10 shows the difference in score between the consistent value, the splits value, the sampling approach, and the search procedure in more detail. In this figure, the score of the method using the consistent value is plotted against each of the other methods. Each individual dot in this plot represents a single data-set. The x- and y-values of a dot are the scores of the respective methods on this data-set. In the first plot, we can see clearly that the sampling approach usually scores a lot worse than the consistent value: there are just a few cases in which the sampling approach is better than the consistent value, however the other way around there are many. From the second plot, it is more difficult to draw a good conclusion. There are some cases in which one clearly outperforms the other, but about an equal amount in both ways. The third plot clearly shows the value of searching: although sometimes the score becomes slightly worse, usually it either remains the same or becomes a lot better.

When looking at the boxplots in Figure 4.9 a question that comes to mind is whether it is better to return smaller DRTAs. The results of the search method seem to indicate this (also from Figure 4.10): searching for smaller solutions only performs a little worse in a few cases, but in much more cases it performs a lot better. More specifically, searching performs worse in 440 of all 1920 cases. The average decrease in score of these 440 cases is 0.01. However, in the 1480 remaining cases the average score increase is 0.05. To give more insight into the relation between score increase and size decrease, we created Figure 4.11. Figure 4.11 shows

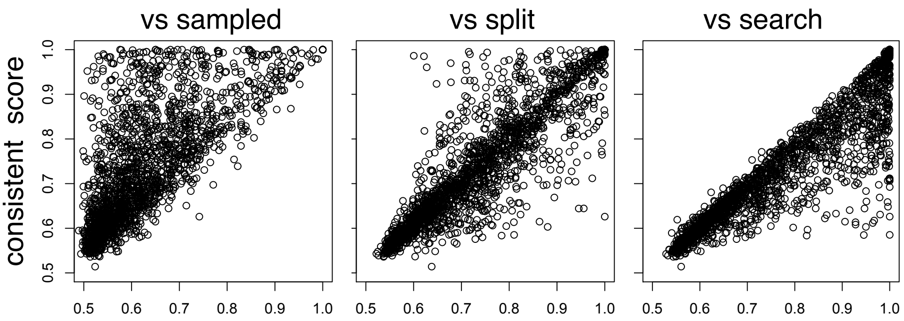


Figure 4.10: The score of the consistent measure plotted against the sampling method, the splits measure, and the search procedure.

the decrease in size when searching plotted against the score increase. Again, each individual dot in this plot represents a single data-set. From this plot we conclude that finding smaller DRTAs helps a lot, but only if the size decrease is more than approximately 25 transitions.

In the remainder of this section, we show plots of the performance of every method on different settings of each individual parameter that was used to generate the data-sets. This can help to give insight into which method should be used in which setting. It is especially interesting to find out with what settings the splits value outperforms the consistent value and the other way around. In addition, these results show how each of the parameters influences the difficulty of the identification problem.

Different sized alphabets Figure 4.12 shows the performance of each of the individual methods on data-sets with the same alphabet size. In these plots, we can clearly see a large performance decrease when the size of the alphabet is increased. Furthermore, we can see that on alphabets of size 2, the splits value outperforms the consistent value both on the resulting size and the score values. On alphabets of size 8, this is the other way around. This suggest that the splits value performs better on the smaller problems and the consistent value performs better on the larger problems. All methods perform bad on the size 8 alphabet problems. Even the search procedure has a median score value just above 0.6. Considering that a score value of 0.5 can be achieved by random guessing, this is not very high.

Different number of states Figure 4.13 shows the performance of each of the individual methods on data-sets with the same number of states in the original DRTA that was used to generate the data-set. In these plots, we again see that all methods perform bad on the large problems. Furthermore, the consistent value produces consistently smaller results than the splits value. However, there is almost no difference in scores between these two methods. The search procedure

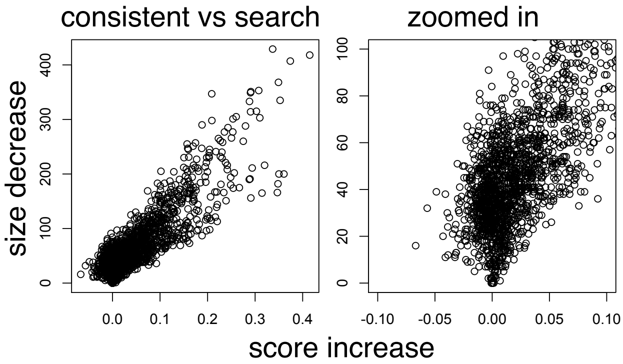


Figure 4.11: The size decrease plotted against the score increase when searching. The plot on the right shows a zoomed-in version of the left plot.

again performs very good on the small problems. It is almost as bad as the consistent value (algorithm without search) on the large problems. As expected, the sampling method performs the worst overall.

Different number of splits Figure 4.14 shows the performance of each of the individual methods on data-sets with the same number of splits in the original DRTA that was used to generate the data-set. Hence, this should really show the difficulty of identifying the correct splits. Intuitively, when the number of splits is high, the evidence value that make use of time information should perform better than those that do not. This is exactly what we observe. With few splits, the scores of both the splits and the impact values score are lower than the score of the consistent value. With increasing amounts of splits, however, the difference in score becomes less. At 16 splits, the splits value already outperforms the consistent value. At 32 splits, even the impact value starts to outperform the consistent value. In fact, this is the only plot we have where the impact value outperforms any of the other values. Note, however, that the sizes of the solutions found by the consistent value are still smaller than the sizes of the solutions found by the splits value. Another interesting observation is that the first two plots show a very large variance in score values. However, this is not that surprising since these include problems with large alphabets and many states. Moreover, a large alphabet creates much larger problems if the amount of states is larger. That is why we do not see this large variance in the different number of states plots. Another consequence of this is that all methods perform better in the case of many splits, than in the case of many states.

Different number of time values and data-set sizes Figure 4.15 shows the performance of each of the individual methods on data-sets while varying the amount of examples, and the amount of time values. From these plots we draw

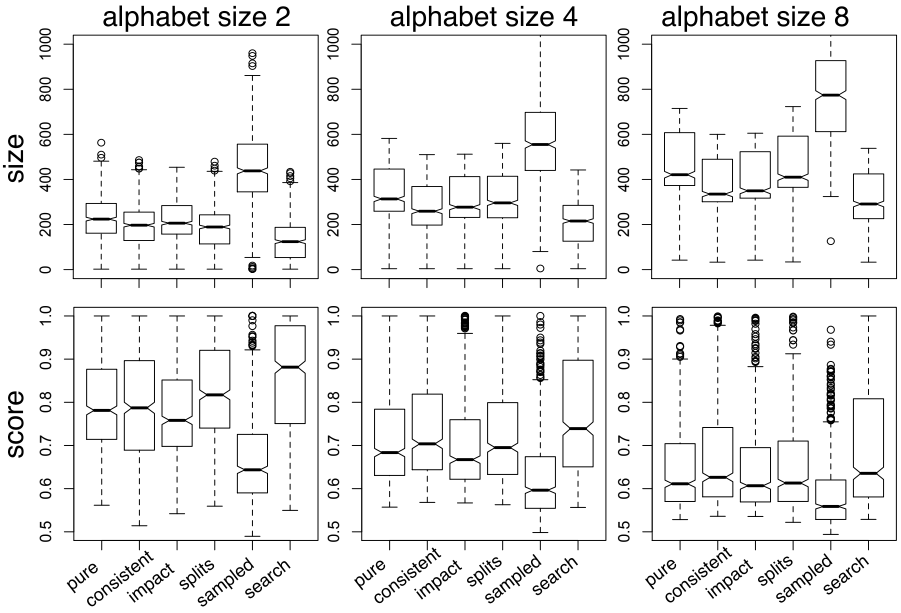


Figure 4.12: Boxplots of the sizes and scores of every method on different sized alphabets.

two important conclusions:

- More data leads to larger DRTAs, but the score of these DRTAs is significantly larger. This also holds for the sampling approach.
- An increase in the number of time points does not affect the results of the DRTA identification algorithms much, but the performance of the sampling approach decreases significantly.

The first conclusion is very interesting since it seems to contradict the expectation that smaller DRTAs lead to better performance. However, as we can see from the plots, searching still helps, i.e., it decreases the size and increases the score. Thus, there seems to be a dependence on the sizes of DRTAs that perform well and the size of the data sample. Actually, this is not that surprising because the availability of more data makes it possible to infer more transitions correctly, even (or especially) in larger DRTAs. But this does not explain why RTI produces larger DRTAs when more data is available. We believe that the reason this occurs is that more data also creates more possible inconsistencies (a positive example that ends in the same state as a negative example). To solve these inconsistencies, RTI requires larger DRTAs. This especially holds if RTI makes a (small) mistake, i.e., it performs an incorrect merge or split. In this case, when there is more data, more timed strings will end up in wrong states. Hence, more possible inconsistencies are

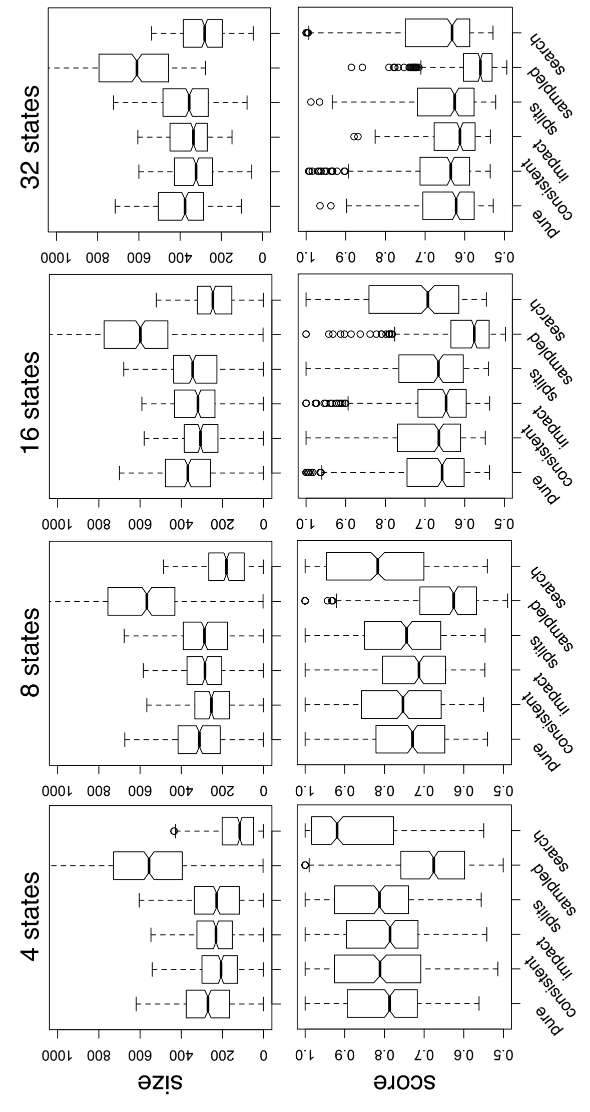


Figure 4.13: Boxplots of the sizes and scores of every method on different number of states of the original DRTA.

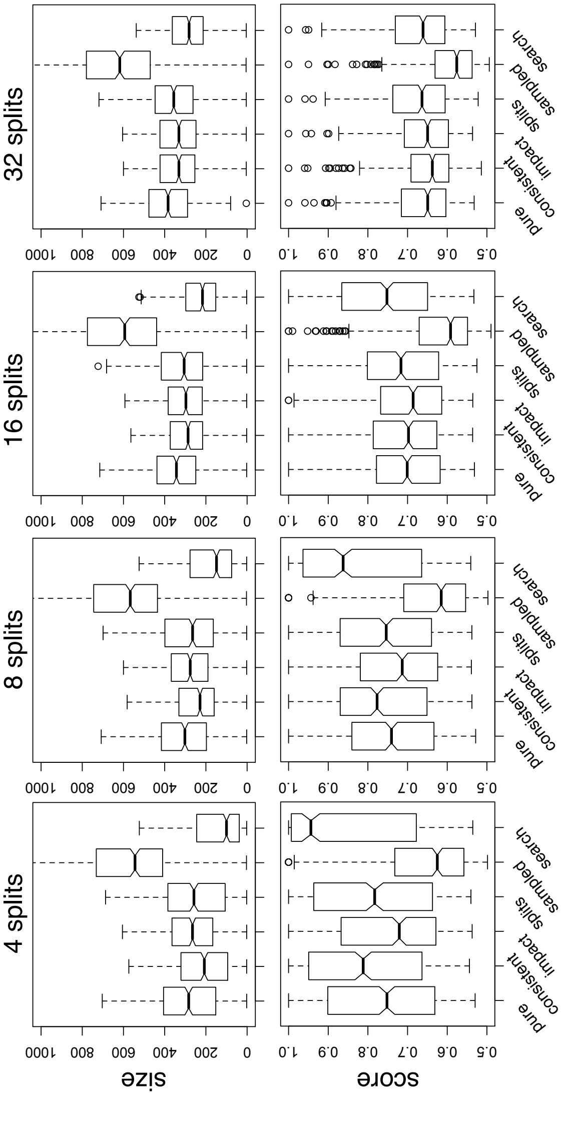


Figure 4.14: Boxplots of the sizes and scores of every method on different number of splits of the original DRTA.

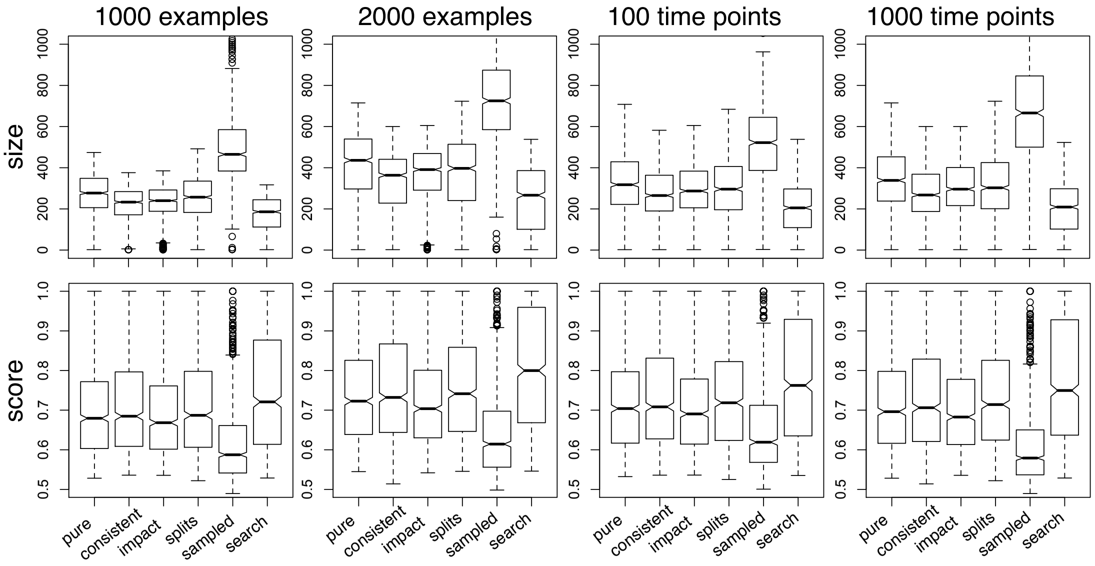


Figure 4.15: Boxplots of the sizes and scores of every method when varying the amount of possible time values and the size of the data-sets.

created. Note that this phenomenon is not the same as the overtraining problem in machine learning, see e.g. (Mitchell 1997). Overtraining occurs when a large number of parameters (states/splits) are fitted too closely to the data-set. In such a case, the performance on the test-set decreases. In our case, this performance increases.

The second conclusion is also interesting. However, the fact that the sampling method performs worse is not very surprising. The sampling method uses a fixed sampling rate (of length 10). Thus, when there are more time points, the blowup in the length of the examples will be bigger. In this case, it is ten times bigger. Longer strings contain less useful data (it can create less inconsistencies) for the DFA identification algorithm, and hence the performance decreases. More interesting is the fact that it does not seem to matter much for our DRTA identification algorithm. In the DRTA case, an increase in the number of time points will make more splits possible, and hence results in a larger search space. When the amount of possible splits is larger, it makes sense that the chance that RTI chooses an incorrect split increases, and hence that the score decreases. Fortunately, in the boxplots of Figure 4.15, this seems not to occur.

Different sized DRTAs Figure 4.16 to 4.18 show the performance of each of the individual methods on data-sets with same sized original DRTAs. For every combination of alphabet size, states, and splits the figures show boxplots of both the size and the score of each method. The figures mostly show some of the conclusions we already obtained from the previous figures:

- The sampling method performs consistently worse than RTI, this is independent of the used heuristic.
- Among the four evidence values, the consistent measure results in the smallest DRTAs.
- The splits value obtains better scores than the consistent measure when there are many splits.
- The impact value does not perform well.
- The search procedure performs best.

In addition we can draw the following conclusions from Figures 4.16 to 4.18:

- For the smallest size DRTAs, the search procedure solves the problems almost optimally.
- The number of splits has a big influence on the difficulty of a problem, especially in the case of small DRTAs.
- The search procedure scores sufficiently good for real-world problems (about 80% accuracy) for DRTAs with up to 16 states, 16 splits, and alphabet of size 2.
- The search procedure scores sufficiently good for DRTAs with up to 16 states, 8 splits, and alphabet of size 4.

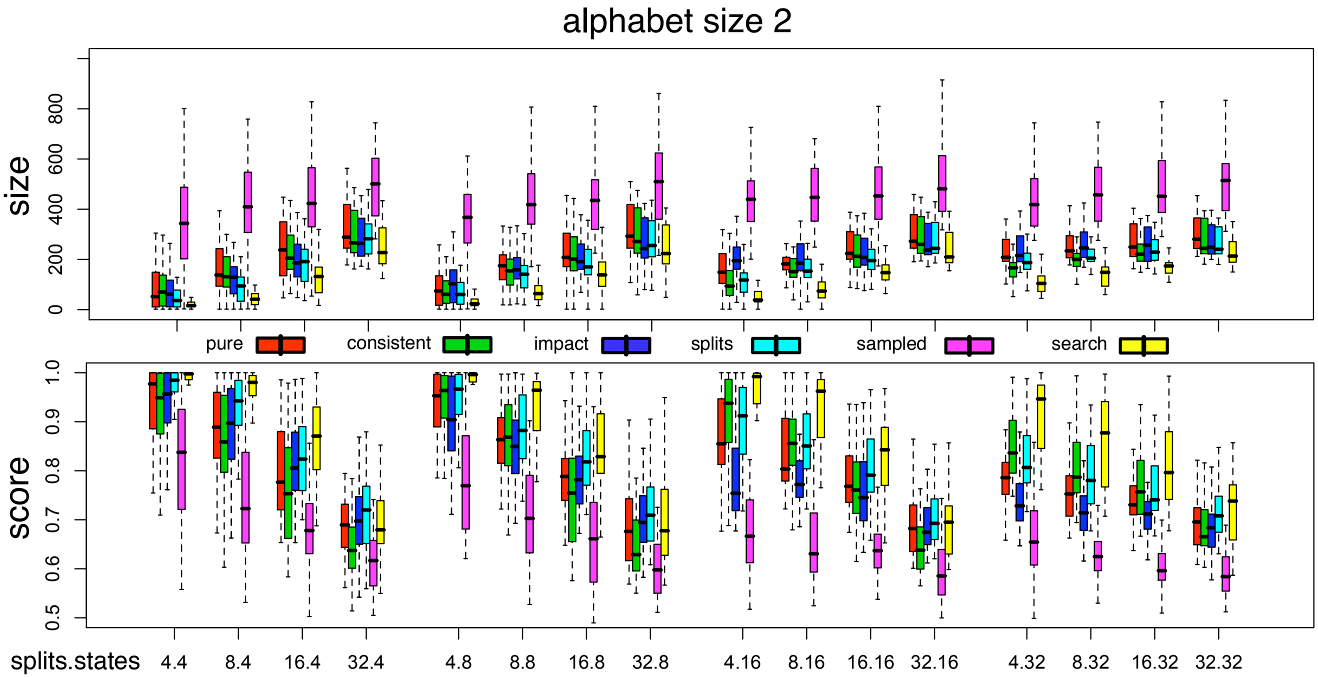


Figure 4.16: Boxplots of the sizes and scores of every method on the data-sets with an alphabet size of 2 and when varying the amount of states and splits of the original DRTA.

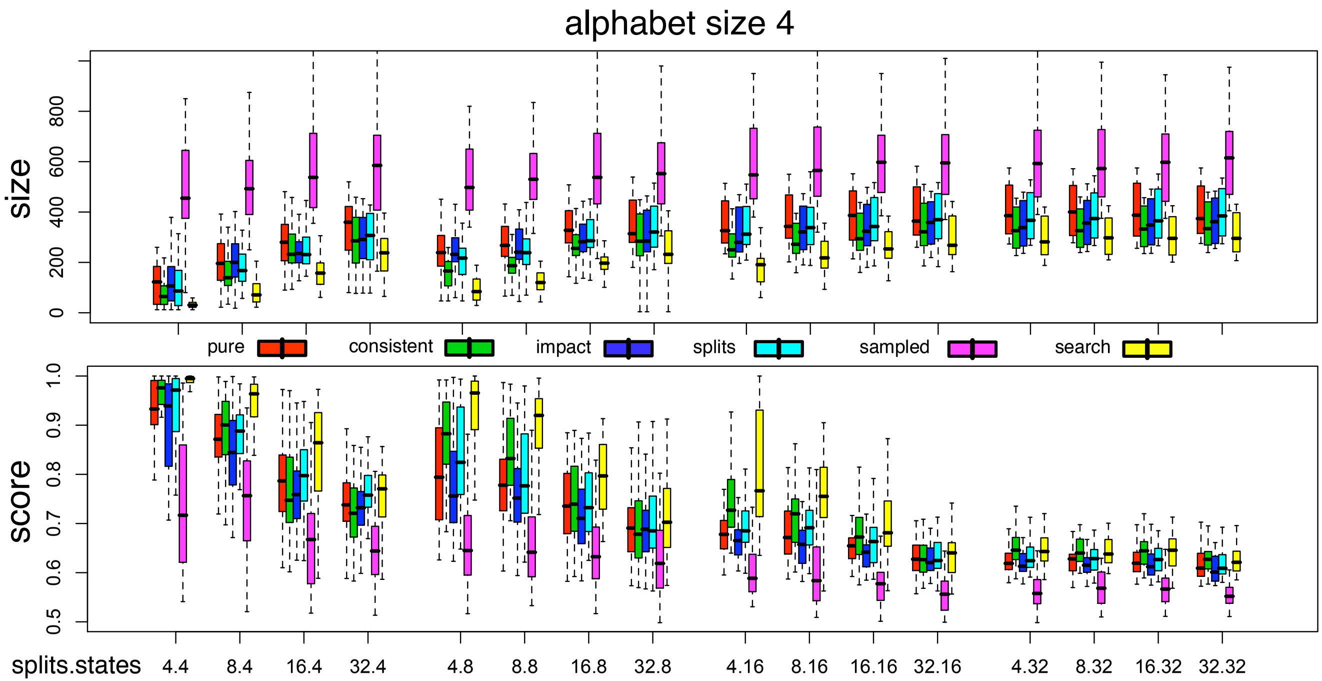


Figure 4.17: Boxplots of the sizes and scores of every method on the data-sets with an alphabet size of 4 and when varying the amount of states and splits of the original DRTA.

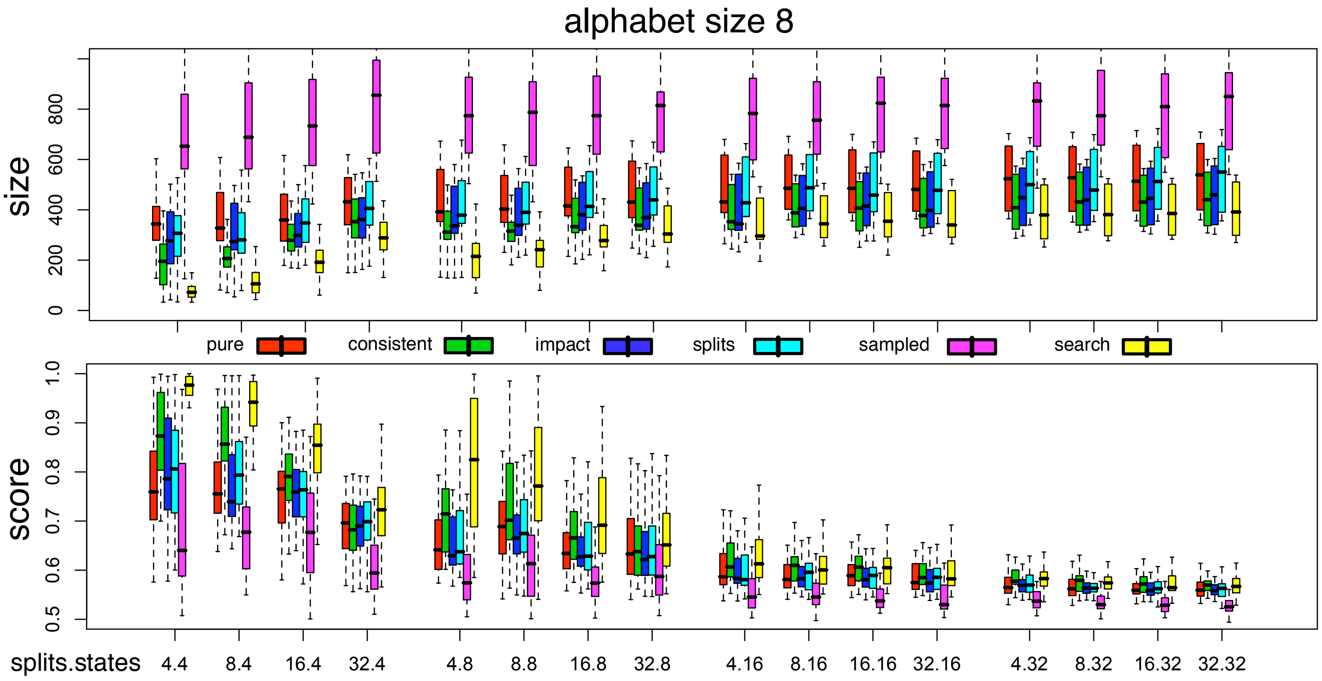


Figure 4.18: Boxplots of the sizes and scores of every method on the data-sets with an alphabet size of 8 and when varying the amount of states and splits of the original DRTA.

- The search procedure scores sufficiently good for DRTAs with up to 4 states, 8 splits, and alphabet of size 8.

- For the larger DRTAs, the search procedure does not significantly improve the performance of RTI.

These conclusions show that RTI performs sufficiently good for real-world problems when either the alphabet, or the number of states is small. In addition, the second conclusion shows that identifying the timed properties of a DRTA is indeed a difficult problem.

Timing We did not perform any time measurements of our experiments. However, the consistent value can be computed very quickly. Running RTI with the consistent value over all of the 1920 problems took a few hours. Running RTI with the splits value took twice as long. The sampling approach took even longer. It took about 5 days in order to compute the results. Naturally, the search procedure took the longest, which required about 3 weeks. However, searching the smaller DRTAs (alphabet 4 and 8 states) usually finished in one or two minutes. We performed the experiments on a 1.8 GHz PowerPC G5 computer.

4.6 Discussion

We have constructed a novel identification algorithm for deterministic real-time automata (DRTAs). These automata can be used to model systems for which the time between consecutive events is important for the system's behavior. We adapted the state-merging algorithm for the identification of deterministic finite automata to the setting of real-time automata. To the best of our knowledge, ours is the first algorithm that can identify a timed automaton model from a timed input sample. Our results show that learning a DRTA returns a model of higher accuracy than a straightforward adaptation of a state-of-the-art state-merging method by using sampling.

We believe that the main reason why RTI outperforms the sampling method is that RTI uses a heuristic to determine the values of the delay guards. In the sampling case, these values are fixed and lead to an exponential blowup. Intuitively, the use of a heuristic to avoid this blowup seems a good idea.

For many applications, a big problem of RTI is that it requires *labeled* data. In many settings, it is very difficult to obtain labeled data: we can use sensors to observe systems, but not to label their behavior. Because we also want to apply RTI to such a setting, we will extend the RTI algorithm to the setting of unlabeled (only positive) data in the next chapter.

CHAPTER 5

Identifying deterministic probabilistic real-time automata

This chapter is based on work published in Proceedings of Induction of Process Models at ECML PKDD (Verwer, de Weerdt and Witteveen 2008a).

5.1 Introduction

In Chapter 4, we described the RTI algorithm (Algorithm 4.5) for identifying deterministic real-time automata (DRTAs) from labeled data, i.e., from an input sample $S = (S_+, S_-)$. Although this algorithm is efficient in both run-time and convergence (see Section 4.3.4), in practice it can sometimes be difficult to apply this algorithm, the reason being that data can often only be obtained from observations of the process to be modeled. In these cases, we have access to a system that continuously produces event-time value pairs. We can observe such a system, for example using sensors. From such observations we only obtain timed strings that have actually been generated by the system. In other words, we only have access to the positive data S_+, also known as *unlabeled data*.

In this chapter, we adapt the RTI algorithm to this setting. A straightforward way to do this is to make the model probabilistic, and to check for consistency using statistics. This has been done many times, and in different ways, for the problem of identifying DFAs (see Section 2.4.3). As far as we know, this is the first time such an approach is applied to the problem of identifying DRTAs.

We start this chapter by defining a *probabilistic DRTA* (PDRTA, Section 5.2). In addition to a DRTA *structure*, a PDRTA contains *parameters* that model the probabilities of events in the DRTA structure. In order to identify a PDRTA, we thus need to solve two different identification problems: the first problem is to identify the correct DRTA structure, and the second is to set the probabilistic parameters of this model correctly. This type of identification is also known as *model selection*, see, e.g., (Grünwald 2007). However, because a PDRTA is a deterministic model, we can simply set these parameters to the normalized frequency counts of events in the input sample S_+.[1] This is very easy to compute and it is the unique correct setting of the parameters. We therefore focus on identifying the DRTA structure of a PDRTA.

We introduce several statistical tests that can be used to solve this identification problem (Section 5.3). These tests are based on the commonly used χ^2, likelihood-ratio, and Kolmogorov-Smirnov tests (Sections 5.3.1, 5.3.2, 5.3.3). Intuitively, the tests we introduce test the null-hypothesis that the suffixes of strings that can occur after two different states have been reached follow the same PDRTA distribution, i.e., whether these two states can be modeled using a single state in a PDRTA. If this null-hypothesis is rejected with sufficient confidence, then this is considered to be evidence that these two states should not be merged. Equivalently, if these two states result from a split of a transition, then this is evidence that this transition should be split. In this way, the *statistical evidence* resulting from these tests replace the evidence value in the original RTI algorithm. In addition, we require methods in order to deal with small frequencies in the data (Section 5.3.4). The result is the RTI+ algorithm (Section 5.3.5), which stands for *real-time identification from positive data*. The RTI+ algorithm is a polynomial time algorithm that converges efficiently to the correct PDRTA in the limit with probability 1.

The statistical tests used by the RTI+ algorithm are designed specifically for the purpose of identifying a PDRTA from unlabeled data. Although many algorithms like RTI+ exist for the problem of identifying probabilistic DFAs (PDFAs, see Section 2.4.3), none of these algorithms uses the non-timed version of a test that we introduced for RTI+. These tests can however be modified in order to identify PDFAs. Hence, they also contribute to the current state-of-the-art in PDFA identification. As such, this chapter is somewhat different from the previous chapter: instead of applying and extending results and algorithms for the problem of DFA identification to the problem of DRTA identification, we develop new techniques that can also be used in DFA identification.

We evaluate the performance of the RTI+ algorithm with different statistical tests experimentally on artificially generated data (Section 5.4). For these tests, we use the same experimental setup as the previous chapter (see Section 4.5.2), the only difference is that we only generate positive data (Section 5.4.1). Unfortunately, unlike models that are identified from labeled data, it is difficult to measure the quality of models that are identified from unlabeled data. A commonly-used quality measure for natural language identification tasks is *test set perplexity* (see, e.g., (Jurafsky and Martin 2000)). This measures the predictive quality of identified models on unseen data. Although test set perplexity does reflect how good the identified models represent the actual distribution of the data, we argue that it is unsuitable for our model selection problem (Section 5.4.2). The reason is that we

[1] In the case of a non-deterministic model, setting the model parameters is a lot harder. In fact, it can be as difficult as identifying the model itself.

are only interested in identifying the correct DRTA model of a PDRTA, and not in setting the parameters of this model correctly. Another performance measure we could use is the Akaike Information Criterium (AIC) (Section 5.4.3). The AIC is a well-known model selection criterion from statistics, see, e.g., (Grünwald 2007). Unfortunately, the AIC does not measure how well the identified models perform on unseen data, i.e., how good the identified models represent the actual distribution of the data.

Since we are only interested in identifying the correct PDRTA model, but not its parameters, we therefore propose the following approach for testing the performance of the RTI+ algorithm: first identify a PDRTA model from a data set, then set the parameters of the model using the test set, and finally compute the resulting perplexity and AIC using the test set (Section 5.4.4). We call this the *test-set-tuned* perplexity and AIC, respectively. Both measures give sensible results. Therefore, we use both of them to test the performance of the RTI+ algorithm (Section 5.4.5). An interesting conclusion of these experiments and the differences from the previously discussed measures is that in terms of data-requirements it is often easier to identify a model than to set its parameters correctly. In addition, the results show that the RTI+ algorithm is capable of identifying sufficiently large PDRTAs in order to be used in practice.

We end this chapter with some conclusions and pointers for future work regarding the algorithm, statistics, and the new performance measures (Section 5.5).

5.2 Probabilistic DRTAs

In order to identify a DRTA from positive data S_+, we need to model a probability distribution for timed strings using a DRTA structure. This is done by adding probability distributions for the symbols and time values of timed events to every state of a DRTA. Learning a DRTA then consists of fitting these distributions and the model structure to the data available in S_+. We want to adapt the RTI of the previous chapter to identify these ***probabilistic DRTAs*** (PDRTAs). Since they have the same structure as DRTAs, we only need to decide how to represent the probability of observing a certain timed event (a, t) given the current state q of the PDRTA, i.e., $Pr(O = (a, t) \mid q)$. In order to determine the probability distribution of this random variable O, we require two distributions for every state q of the DRTA: one for the possible symbols $Pr(S = a \mid q)$, and one for the possible time values $Pr(T = t \mid q)$. The probability of the next state $Pr(X = q' \mid q)$ is determined by these two distributions because the PDRTA model is deterministic.

The distribution over events $Pr(S = a \mid q)$ that we use is the standard generalization of the Bernoulli distribution, i.e., every symbol a has some probability $Pr(S = a \mid q)$ given the current state q, and it holds that $\sum_{a \in \Sigma} Pr(S = a \mid q) = 1$ (also known as the categorical distribution). This is the most straightforward choice for a distribution function and it is used in many probabilistic models, such as Markov chains, hidden Markov models, and probabilistic automata (see Section 2.3.3).[2] A consequence of modeling the event distributions in this way is that

[2] In probabilistic automata there sometimes exists for every state q an additional probability $Pr(end \mid q)$ that the process ends in q, given that the current state is q. We choose not to model

the resulting PDRTA model is *Markovian* with respect to the events:

Definition 5.1. *(Markovian events) A probability distribution $Pr(S = a)$ of the next symbol a in an automaton model is called* Markovian *if it depends only on the current state q_i of the PDRTA and not on the past states $q_1, \ldots, q_{i-1}$ or observations $o_1, \ldots, o_{i-1}$, i.e., if $Pr(S = a \mid q_i) = Pr(S = a \mid q_1, \ldots, q_i, o_1, \ldots, o_{i-1})$.*

The choice of which distribution over time $Pr(T = t \mid q)$ to use is more difficult. A commonly used distribution is the exponential distribution. This distribution is used for example in continuous time Markov chains (see Section 2.3.3). A nice property of this distribution is that it is Markovian: the amount of time spent in q is independent of the amount of time already spent in q. A problem, however, of using such a distribution is that the time distribution is fixed to be of a specific form. Hence, this poses strict requirements on the type of model that our algorithm can identify. More specifically, if the actual time distribution does not follow this form, then a model that uses the exponential distribution will not be able to model the process very well. We like our algorithm to be able to identify a wider range of models.

The most straightforward way to approximate an unknown distribution is to use a histogram. A histogram divides the domain of the distribution (in our case time) into a fixed number of bins H. Every bin $[v, v'] \in H$ is an interval in $\mathbb{N}$. The distributions inside the bins are modeled uniformly, i.e., for all $[v, v'] \in H$ and all $t, t' \in [v, v']$, $Pr(T = t \mid q) = Pr(T = t' \mid q)$. Naturally, it has to hold that all these probabilities sum to one: $\sum_{t \in \mathbb{N}} Pr(T = t \mid q) = 1$. Usually, we use the probability of a bin $Pr([v, v'] \mid q) = \sum_{t \in [v,v']} Pr(T = t \mid q)$ directly instead of the probability of a time value. Using a histogram to model the time distribution might look simple, but it is very effective. In fact, it is a common way to model time in hidden semi-Markov models, see, e.g., (Guédon 2003). As a consequence of using a histogram, the time distributions in the PDRTA model are *semi-Markovian*:

Definition 5.2. *(semi-Markovian time values) A probability distribution $Pr(T = t)$ of the next time values t in an automaton is called* semi-Markovian *if $Pr(T = t)$ is Markovian (with respect to past states and observations), but depends on the time already spend in the current state, i.e., if $\exists t, t' : Pr(T = t \mid q) \neq Pr(T = t + t' \mid T > t', q)$.*

The price of using a histogram to model time is that we need to specify the amount, and the sizes (division points) $[v, v']$ of the histogram bins.[3] Choosing these values boils down to making a tradeoff between the model complexity and the amount of data required to identify the model. More bins lead to a more complex model that is capable of modeling the time distribution more accurately, but it requires more data in order to do so. It seems to make sense to let the amount and the sizes of the bins depend on the amount (and perhaps the shape) of the

this probability because we want to model a continuous (never ending) process. This choice does not influence the algorithm significantly. With some minor modifications it can also be applied when these ending probabilities are required.

[3] Note that the RTI+ algorithm can still split transitions on any possible time-value. The histogram bins only serve as a model for the time distribution in a PDRTA, they do not restrict the possible PDRTA structures in any way.

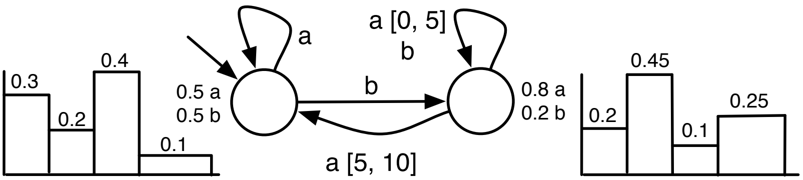


Figure 5.1: A probabilistic DRTA. Every state is associated with a probability distribution over events and over time. The distribution over time is modeled using histograms. The bin sizes of the histograms are predetermined but left out for clarity.

data that is available. To simplify matters, however, we assume that every time distribution has the same division points. More specifically, we do not assume that every state has the same time distribution, but that the PDRTA models uses the same set of bins to approximate all of these distributions. These bins are specified beforehand, for example by a domain expert, or by performing data analysis.

In addition to choosing how to model the time and symbol distributions, we need to decide whether to make these two distributions dependent or independent. It is common practice to make these distributions independent, see, e.g., (Guédon 2003). In this case, the time distribution represents a distribution over the waiting (or sojourn) time of every state. In some cases, however, it makes sense to let the time spent in a state depend on the generated symbol. By modeling this dependence, the model can deal with cases where some events are generated more quickly than others. Unfortunately, this dependence comes with a cost: the size of the model is increased by a polynomial factor (the product of the sizes of the distributions). Due to this blowup, we require a lot more data in order to identify a similar sized DRTA. This is our main reason for modeling these two distributions independently. Furthermore, we also do not have sufficient reasons to divert from common practice.[4]

This results in the following PDRTA model:

Definition 5.3. *(PDRTA) A* probabilistic DRTA *(PDRTA) $\mathcal{A}$ is a quadruple $\langle \mathcal{A}', H, \mathcal{S}, \mathcal{T} \rangle$, where $\mathcal{A}' = \langle Q, \Sigma, \Delta, q_0 \rangle$ is a DRTA without final states, H is a finite set of bins (time intervals) $[v, v']$, $v, v' \in \mathbb{N}$, known as the histogram, $\mathcal{S} = \{p_{a,q} \mid a \in \Sigma, q \in Q\}$ is a finite set of symbol probabilities $p_{a,q} = Pr(S = a \mid q)$, and $\mathcal{T} = \{p_{h,q} \mid h \in H, q \in Q\}$ is a finite set of time-bin probabilities $p_{h,q} = Pr(T \in h \mid q)$. For every state $q \in Q$ it holds that $\sum_{a \in \Sigma} p_{a,q} = 1$ and $\sum_{h \in H} p_{h,q} = 1$.*

The DRTA without final states specifies the *structure* of the PDRTA. The symbol- and time-probabilities $\mathcal{S}$ and $\mathcal{T}$ specify the probabilistic properties of a PDRTA. The probabilities in these sets are called the *parameters* of $\mathcal{A}$. However, in every set, the value of one of these parameters follows from the others because

[4] In future work it might be interesting to identify models with dependent time and symbol distributions using our algorithm. It is also possible to test for this dependence beforehand for every state, before deciding how to model it.

their values have to sum to 1. Hence, there are $(|\mathcal{S}|-1)+(|\mathcal{T}|-1)$ parameters per state of our PDRTA model.

The probability $p_{t,q}$ that the next time value equals t given that the current state is q is defined as $p_{t,q} = Pr(T = t \mid q) = \frac{p_{h,q}}{v'-v+1}$, where $h = [v, v'] \in H$ is such that $t \in [v, v']$. Thus, in every time-bin the probabilities of the individual time points are modeled uniformly. The probability of an observation (a, t) given that the current state is q is defined as

$$Pr(O = (a, t) \mid q) = p_{a,q} \times p_{t,q}$$

This implies that the distributions over events and time are independent. The probability of the next state q' given that the current state is q is defined as

$$Pr(X = q' \mid q) = \sum_{\langle q,q',a,t\rangle \in \Delta} p_{a,q} \times p_{t,q}$$

Thus, the model is deterministic. The computation of a PDRTA is defined analogously to the computation of a DRTA. A PDRTA models a distribution over timed strings $Pr(O^* = \tau)$. Let

$$q_0 \xrightarrow{(a_1,t_1)} q_1 \dots q_{n-1} \xrightarrow{(a_n,t_n)} q_n$$

be the computation of a PDRTA $\mathcal{A}$ over the timed string $\tau = (a_1, t_1) \dots (a_n, t_n)$. The probability of τ given $\mathcal{A}$ is:

$$Pr(O^* = \tau \mid \mathcal{A}) = \prod_{1 \le i \le n} Pr(O = (a_i, t_i) \mid q_{i-1})$$

A PDRTA can be used as a *predictor* because it defines distributions over next timed events (a, t) given a finite history τ:

$$Pr(O = (a, t) \mid \tau, \mathcal{A}) = \frac{Pr(\tau(a, t) \mid \mathcal{A})}{Pr(\tau \mid \mathcal{A})}$$

Example 5.1. Figure 5.1 shows a PDRTA $\mathcal{A}$. Let $H = \{[0, 2]; [3, 4]; [5, 6]; [7, 10]\}$ be the histogram. In every bin the distribution over time values is uniform. We can use $\mathcal{A}$ as a predictor of timed events. For example, the probability of $(a, 3)(b, 1)(a, 9)(b, 5)$ is $Pr((a, 3)(b, 1)(a, 9)(b, 5)) = 0.5 \times \frac{0.2}{2} \times 0.5 \times \frac{0.3}{3} \times 0.8 \times \frac{0.25}{4} \times 0.5 \times \frac{0.4}{2} = 1.25 \times 10^{-5}$. The probability of observing $(b, 5)$ after seeing $(a, 3)(b, 1)(a, 9)$ (used for prediction) is $Pr((b, 5) \mid (a, 3)(b, 1)(a, 9)) = \frac{1.25 \times 10^{-5}}{1.25 \times 10^{-4}} = 0.1 = 0.5 \times \frac{0.4}{2}$.

A consequence of computing the probability of a timed string τ in this way is that the PDRTA model is *output independent*:

Definition 5.4. *(output independence) An automaton is called* output independent *if the probability $Pr(O = o_i \mid q_{i-1})$ of an observation o_i (timed event) given the current state q_{i-1} is independent of the previous observations $o_1, \dots, o_{i-1}$, i.e. $P(O = o_i \mid q_{i-1}) = P(O = o_i \mid q_{i-1}, o_1, \dots, o_{i-1})$.*

In HMMs, output independence also holds and it has been said to be very restrictive and hence limit the possible applications of HMMs, see, e.g., (Rabiner 1989). In our case, however, this is not a big problem because we try to identify the *structure* of a PDRTA in addition to its parameters. Therefore, in the case that output independence does not hold for some state q of the actual system we are trying to identify, our algorithm will identify multiple states that represent q, each of which is output independent. In other words, when trying to identify an output dependent system, our algorithm will identify a (sometimes much larger) output independent system that is language equivalent to the actual output dependent system. Hence, this does not influence the applicability of our algorithm.

A possible issue is that identifying multiple states that represent a single state in the actual system can lead to some blow-up in the size of the PDRTA model. If the actual system is highly output dependent (or non-stationary), such as a multi-clock DTA (see Section 3.4), this blow-up can be exponential. In such cases, it will be difficult to apply our algorithm because it will require huge amounts of data in order to identify the correct PDRTA. However, this is not a big issue since it is very likely impossible to identify such highly dependent systems efficiently, as is the case for multi-clock DTAs (see Section 3.2).

A PDRTA essentially models a certain type of distribution over timed strings. All of the mentioned assumptions are valid when the actual model can be modeled using this type of distribution (perhaps at the cost of some blow-up). An input sample S_+ can be seen as a sample drawn from such a distribution. The problem of identifying a PDRTA then consists of finding the distribution that generated this sample. We now describe how we adapt our DRTA identification algorithm in order to identify a PDRTA from such a sample.

5.3 Identifying PDRTAs from positive data

In this section, we adapt the RTI algorithm for the identification of DRTAs from labeled data of the previous chapter to the setting of unlabeled data. The result is the RTI+ algorithm, which stand for *real-time identification from positive data*. Given a set of observed timed strings S_+, the goal of RTI+ is to find a DRTA that describes (the behavior of) the real-time process that generated S_+. Note that, because RTI+ uses statistics to find this DRTA, and hence needs to count the occurrences of timed symbols, S_+ is a multi-set, i.e., S_+ can contain the same timed string multiple times.

Like before (see Section 4.3.2), RTI+ starts with an augmented prefix tree acceptor (APTA). However, since we only have positive data available, the APTA will not contain rejecting states. Moreover, since the points in time where the observations are stopped are arbitrary, it also does not contain accepting states. Thus, the initial DRTA simply is the *prefix tree* of S_+. An example prefix tree is shown in Figure 5.2.

Starting from a prefix tree, our original algorithm tries to merge states and split transitions using a red-blue framework (see Section 2.4.1). RTI+ uses exactly the same operations and framework. The only difference is the evidence value we use. Originally, the evidence was based on the number of positive and negative examples that end in the same state. For RTI+, we require an evidence value that

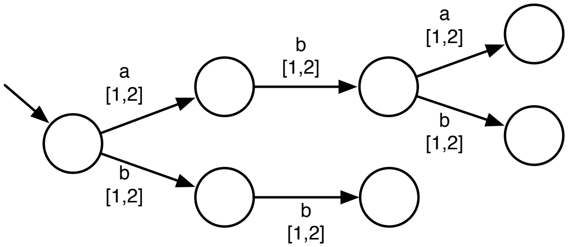


Figure 5.2: A prefix tree. It is identical to an augmented prefix tree acceptor, but without accepting and rejecting states.

uses only positive examples, and that disregards which states these examples end in. The most straightforward evidence values for this type of setting are statistics that count the occurrences of timed events. Statistical tests on these counts can be used to measure the similarity between two states. This similarity measure can be used both as statistical evidence values and as a consistency check. Two states are inconsistent if their similarity goes below a prespecified bound.

We now show the different statistics we use (Sections 5.3.1 to 5.3.4) and how we fit them into RTI+ (Section 5.3.5). Each of these statistics are alternative implementations of the evidence value in RTI+. There exist of course many other possible statistics that can be used as evidence values. We chose the ones described here because they are the most straightforward implementations of the ideas in this chapter. We test every implementation in the results section (Section 5.4).

5.3.1 A likelihood ratio test for state-merging

The likelihood ratio test (see, e.g., (Hays 1994)) is a common way to test nested hypotheses. A hypothesis H is called *nested* within another hypothesis H' if the possible distributions under H form a strict subset of the possible distributions under H'. Less formally, this means that H can be created by constraining H'. Thus, by definition H' has more unconstrained parameters (or degrees of freedom) than H. Given two hypotheses H and H' such that H is nested in H', and a data set S_+, the likelihood ratio test statistic is computed by

$$\mathsf{LR} = \frac{\mathsf{likelihood}(S_+, H)}{\mathsf{likelihood}(S_+, H')}$$

where likelihood is a function that returns the *maximized likelihood* of a data set under a hypothesis, i.e., $\mathsf{likelihood}(S_+, H)$ is the maximum probability (with optimized parameter settings) of observing S_+ under the assumption that H was used to generate the data.

Let H and H' have n and n' parameters respectively. Since H is nested in H', the maximized likelihood of S_+ under H' is always greater than the maximized likelihood under H. Hence, the likelihood ratio LR is a value between 0 and 1. When the difference between n and n' grows, the likelihood under H' can be optimized more and hence LR will be closer to 0. Thus, we can increase the likelihood of the data S_+ by using a different model (hypothesis) H', but at the cost of using more parameters $n' - n$. The likelihood ratio test can be used to test whether this increase in likelihood is statistically significant. The test compares the value $-2\mathsf{ln}(\mathsf{LR})$ to a χ^2 distribution with $n' - n$ degrees of freedom. The result of this comparison is what is called a p-value.

A *p-value* is the probability that a value (in our case a χ^2 distribution with $n' - n$ degrees of freedom) would assume a value greater than or equal to the observed value (in our case $-2\mathsf{ln}(\mathsf{LR})$) strictly by chance. Thus, in our case, a high p-value indicates that H is a better model since the probability that $n' - n$ extra parameters results in the observed increase in likelihood is high. Consequently, a low p-value indicates that H' is a better model. When the p-value is less than 0.05, we are 95% certain that the observed increase in likelihood cannot have occurred strictly by chance.

Applying the likelihood ratio test to state-merging and transition-splitting is remarkably straightforward. Suppose that we want to test whether we should perform a merge of two states. Thus, we have to make a choice between two PDRTAs (models):

- the PDRTA $\mathcal{A}$ resulting from the merge of these states;
- the PDRTA $\mathcal{A}'$ before merging these states.

Clearly, $\mathcal{A}$ is nested in $\mathcal{A}'$. Thus all we need to do is compute the maximized likelihood of S_+ under $\mathcal{A}$ and $\mathcal{A}'$, and apply the likelihood ratio test. Since PDRTAs are deterministic, the maximized likelihood can be computed simply by setting all the probabilities in the PDRTAs to their normalized counts of occurrence in S_+. We now show how to use this test in order to determine whether to perform a merge using an example.

Example 5.2. In this example, we disregard the time values of timed strings and the timed properties of PDRTAs. This greatly simplifies matters and the procedure can easily be extended to make use of the time values and timed properties. Suppose we want to test whether to merge the two root states of the prefix trees of Figure 5.3. These two prefix trees are parts of the PDRTA we are currently trying to identify. Hence only some strings from S_+ reach the top tree, and some reach the bottom tree.

Let $S = \{10 \times a, 10 \times aa, 20 \times ab, 10 \times b\}$ and $S' = \{20 \times aa, 20 \times bb\}$ be the suffixes of these strings starting from the point where they reach the root state of the top and bottom tree respectively, where $n \times \tau$ means that the (timed) string τ occurs n times. We first set all the parameters of the top tree in such a way that the likelihood of S is maximized: $p_{a,q_0} = \frac{4}{5}, p_{b,q_0} = \frac{1}{5}, p_{a,q_1} = \frac{1}{3}, p_{b,q_1} = \frac{2}{3}$ (this is easy because the model is deterministic). We do the same for the bottom tree and S': $p'_{a,q_0} = \frac{1}{2}, p'_{b,q_0} = \frac{1}{2}, p'_{a,q_1} = 1, p_{b,q_2} = 1$.

We can now compute the maximized likelihood[5] of the relevant data set (of the parts that reach the states that we want to merge) by first computing the

[5] This procedure can be simplified by computing the logarithms of these probabilities directly. This avoids the problem that the computed value becomes very small very quickly.

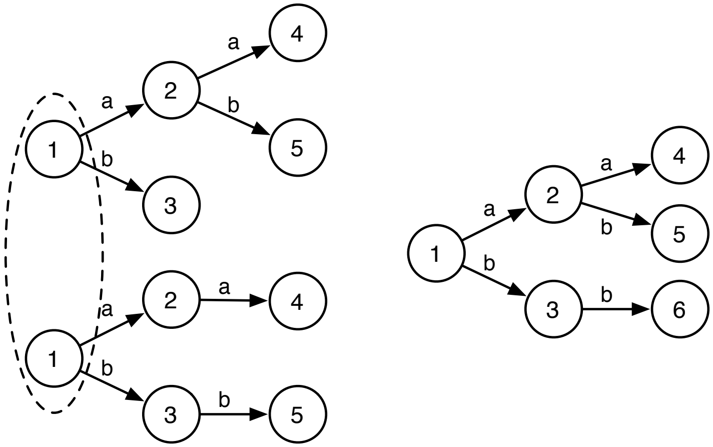


Figure 5.3: The likelihood ratio test. We test whether using the left model (two prefix trees) instead of the right model (a single prefix tree) results in a significant increase in the likelihood of the data with respect to the number of additional parameters (used to model the state distributions).

probability of S under the top tree:

$$p_1 = \left(\frac{4}{5}\right)^{40} \times \left(\frac{1}{5}\right)^{10} \times \left(\frac{1}{3}\right)^{10} \times \left(\frac{2}{3}\right)^{20} \approx 6.932 \times 10^{-20}$$

then the probability of S' under the bottom tree:

$$p_2 = \left(\frac{1}{2}\right)^{20} \times \left(\frac{1}{2}\right)^{20} \approx 9.095 \times 10^{-13}$$

Next, we set the parameter of the right tree to maximize the likelihood of $S \cup S'$: $p_{a,q_0} = \frac{2}{3}, p_{b,q_0} = \frac{1}{3}, p_{a,q_1} = \frac{3}{5}, p_{b,q_1} = \frac{2}{5}, p_{b,q_2} = 1$, and compute the likelihood of the data under the right (merged) tree:

$$p_3 = \left(\frac{2}{3}\right)^{60} \times \left(\frac{1}{3}\right)^{30} \times \left(\frac{3}{5}\right)^{30} \times \left(\frac{2}{5}\right)^{20} \approx 3.211 \times 10^{-40}$$

We multiply the top and bottom tree probabilities in order to get the likelihood of the data under the left (un-merged) tree, and use this to compute the likelihood ratio:

$$LR = \frac{p_3}{p_1 \times p_2} \approx 5.093 \times 10^{-9}$$

The χ^2 value that we need to compare to a χ^2 distribution then becomes $\chi^2 \approx 38.19$. Per state $|\Sigma| - 1$ untimed parameters are used ($|\Sigma| \times |H| - 2$ in the timed case). We subtract 1 because the value of the last parameter follows from the others since their sum has to be equal to 1. In the un-merged model, the number of (untimed) parameters is 5. In the merged model it is 3. The un-merged model in general uses roughly twice as many parameters as the single tree model. A likelihood-ratio test using these values results in a p-value of 5.093×10^{-9}. Since this is a lot less than 0.05, we conclude that the merge results in a significantly worse model.

Testing whether to perform a split of a transition can be done in a similar way. When we want to decide whether to perform a split, we also have to make a choice between two PDRTAs: the PDRTA before splitting $\mathcal{A}$, and the PDRTA after splitting $\mathcal{A}'$. $\mathcal{A}$ is again nested in $\mathcal{A}'$, and hence we can perform the likelihood ratio test in the same way.

5.3.2 Fisher's method of combining p-values

In the previous section, we described a maximum-likelihood method for obtaining a confidence value for whether two states represent the same distribution. The method is straightforward but it is very dependent on the number of parameters of the models we test. This causes some problems, which we discuss in Section 5.3.4. There are of course many alternatives to the likelihood ratio test. However, all of the commonly used tests from statistics operate on either binned or ordinal data. It is not possible to directly apply them to a prefix tree. We need some way to transform the data that is available in the prefix trees to the data these tests use. We do so using Fisher's method (Fisher 1948).

Fisher's method for combining p-values can be used to combine several p-values that come from independent tests of a single hypothesis into a single overall test for that hypothesis. The main idea of the method is that the p-values coming from these tests should follow some known distribution under the null-hypothesis. Given n independent tests, let v_i denote the p-value of the ith test. Fisher's method relies on the fact that under the null-hypothesis

$$F = -2 \times \sum_{1 \leq i \leq n} \ln(v_i)$$

approximates a χ^2 distribution with $2n$ degrees of freedom. Thus, we can compare F to the critical values of this distribution in order to obtain a single overall p-value.

The main use of Fisher's method is in cases where there are several tests that do not allow us to reject the null-hypothesis, for example because each of these tests is performed on an insufficient number of data samples. In such a scenario, Fisher's method can be used to combine the results of these tests. Although each of the individual tests is based on an insufficient amount of data, the combination of these tests can lead to a rejection of the null-hypothesis.

In our case, we use Fisher's method as an alternative to the likelihood ratio test. The idea is that we can use simple and well-known tests in order to test whether two states are the same locally, i.e., with respect to the binned distributions associated with those states. The overall test can then be computed by combining the p-values from several of these simple tests. In the case of a merge, we perform the simple tests for every pair of states that is merged by the determinization process, see Figure 5.4. In the case of a split, we perform these tests for every pair of states that would be merged if we were to undo the split.

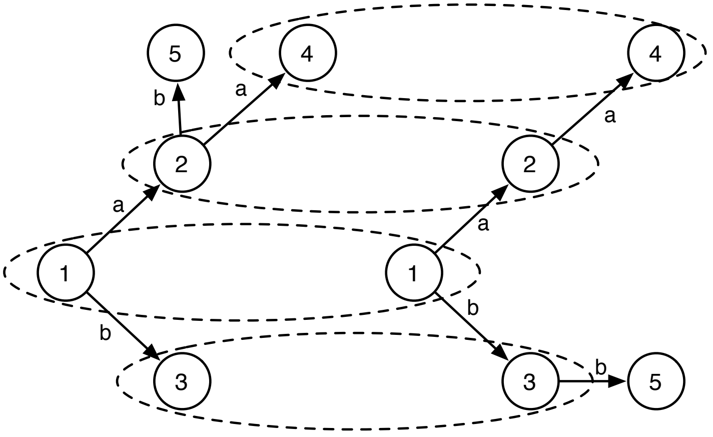


Figure 5.4: Fisher's method for prefix trees. We test whether every pair of states that are surrounded by a dashed ellipsoid are the same. Since every state models two distributions (one over time, and one over symbols), this results in a number of p-values equal to two times the number of such pairs. These p-values are combined using Fisher's method into one single overall p-value.

The simple test we use is the χ^2 test for independence (see, e.g., (Hays 1994)). In order to perform this test, we need to compute the value

$$\chi^2 = \sum_{1 \leq i \leq n} \frac{(O_i - E_i)^2}{E_i}$$

where for all n events, O_i is the observed frequency of event i, and E_i is the expected frequency of event i. The observed frequencies can easily be determined from data. The expected frequency is determined by the null-hypothesis. In our case, we want to test whether two states q and q' are the same. Thus, the expected frequency $E_{i,q}$ of event i in state q is the frequency we expect if q and q' are merged. This is equal to the number of occurrences O_q of state q times the total fraction of i events:

$$E_{i,q} = O_q \times \frac{O_i}{\sum_{1 \leq j \leq n} O_j}$$

In order to test whether q and q' are the same we compute:

$$\chi^2 = \sum_{1 \leq i \leq n} \frac{(O_{i,q} - E_{i,q})^2}{E_{i,q}} + \frac{(O_{i,q'} - E_{i,q'})^2}{E_{i,q'}}$$

where $O_{i,q}$ is the observed frequency of event i in state q. The value χ^2 can be compared with a χ^2 distribution with $n-1$ degrees of freedom. The result is a p-value that gives the significance of the difference between the observed and expected frequencies.

We perform an identical test for the bins of the time distribution of the two states. Thus we have two p-values for every pair of states we test.[6] In theory, these tests that compute these p-values are all independent from each other. This follows from the assumptions in Section 5.2: the probability distribution over the next observation (both on its symbol and its time-value) depends only on the current state, not on any of the other distributions. Hence, the tests over these distributions are also independent from each other. Unfortunately, it is not clear whether this also holds in practice. It could very well be the case that the result of one of these tests influences the results of the other tests. In the case that these results are dependent, it is unclear how to combine them into a single overall value. Our solution is to assume that this independence holds and combine all of the p-values into a single p-value using Fisher's method.

Assumption 5.1. A statistical test that tests whether a pair of states of a PDRTA have similar event distributions is independent of the result from the same test applied to another pair of states in the same PDRTA.

We now use the same example we used in the likelihood ratio test to show how this test is used to determine whether to merge two states.

Example 5.3. As before, we disregard the time values of timed strings and the timed properties of PDRTAs. We want to test whether to merge the two root states of the prefix trees of Figure 5.4. These two prefix trees are parts of the PDRTA we are currently trying to identify. Hence only some strings from S_+ reach the left tree $S = \{10 \times a, 10 \times aa, 20 \times ab, 10 \times b\}$, and some reach the right tree $S' = \{20 \times aa, 20 \times bb\}$. We compute the χ^2 test for the two pairs of states that would be merged if we merge the roots of the prefix trees (the pair of states 1 and 2 in Figure 5.4). For state q_0: $E_{a,q_0} = 50 \times \frac{60}{90} = \frac{100}{3}$, $E_{b,q_0} = 50 \times \frac{30}{90} = \frac{50}{3}$, $E_{a,q_0'} = 40 \times \frac{60}{90} = \frac{80}{3}$, and $E_{b,q_0'} = 40 \times \frac{30}{90} = \frac{40}{3}$. Thus,

$$\chi^2_{q_0} = \frac{(40 - \frac{100}{3})^2}{\frac{100}{3}} + \frac{(20 - \frac{80}{3})^2}{\frac{80}{3}} + \frac{(10 - \frac{50}{3})^2}{\frac{50}{3}} + \frac{(20 - \frac{40}{3})^2}{\frac{40}{3}} = 9$$

For state q_1: $E_{a,q_1} = 30 \times \frac{30}{50} = 16$, $E_{b,q_1} = 30 \times \frac{20}{50} = 12$, $E_{a,q_1'} = 20 \times \frac{30}{50} = 12$, and $E_{b,q_1'} = 20 \times \frac{20}{50} = 8$, thus:

$$\chi^2_{q_1} = \frac{(10-16)^2}{16} + \frac{(20-12)^2}{12} + \frac{(20-12)^2}{12} + \frac{(0-8)^2}{8} = 22\frac{2}{9}$$

We compare these values to a χ^2 distribution with 1 degrees of freedom. This results in the following p-values: 2.70×10^{-3} and 2.43×10^{-6}. We apply Fisher's method to these p-values:

$$F \approx -2 \times (\ln(2.70 \times 10^{-3}) + \ln(2.43 \times 10^{-6})) \approx 37.68$$

We compare this to a χ^2 distribution with $2n = 4$ degrees of freedom, resulting in the following overall p-value: 1.304×10^{-7}. Because this is a lot less than 0.05, we conclude that the two data sets are significantly different and hence should not be merged.

[6] Note that all of these tests are also performed on different pieces of the data set S_+, removing the need for correction due to multiple hypothesis testing.

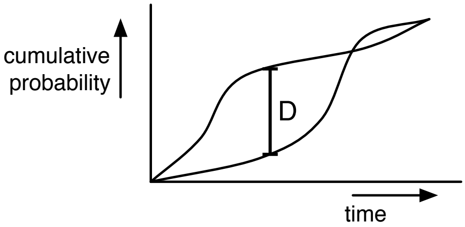


Figure 5.5: The Kolmogorov-Smirnov test tests whether the distance D between two observed cumulative distribution functions is significantly greater than one would normally expect if the two distributions are the same.

5.3.3 A Kolmogorov-Smirnov test for time values

As said before, we do not want to fix the shape of the time distribution. Because of this, we use histograms to model the distribution of time values in states. A problem one may have with these histograms is that they are just a different way of fixing the time distributions. Like the exponential distribution, we only need to set a few parameters in order to determine the entire distribution function. These tests are therefore known as parametric tests. There also exist non-parametric tests. These tests use only the data and no model in order to determine whether the two distributions are similar. We also implemented a non-parametric test for RTI+. The test we use is the well-known Kolmogorov-Smirnov (KS) test.

The KS test operates on ordinal data. Given two sets of time values, of size n and n' respectively, the KS test first computes the cumulative distribution functions F and F' of the two sets. It then finds the maximum distance between these functions, see Figure 5.5:

$$D = \sup_{x \in \mathbb{N}} |F(x) - F'(x)|$$

The value $K = \sqrt{\frac{n \times n'}{n+n'}}D$ converges to the Kolmogorov distribution under the null-hypothesis. Thus, we can use the probability of obtaining an observed K or something less likely from the Kolmogorov distribution as a p-value for a test whether two time distributions are similar. The KS test can be performed on the time distributions from every pair of states that is merged during the determinization procedure (or merged when a split is undone). The resulting p-values can then be combined with the p-values that result from the χ^2 tests of the symbol distributions using Fisher's method. The result is again a single overall p-value.

5.3.4 Dealing with small frequencies

It is well-known that all of the statistical methods we just described do not perform well when few data is available. In the likelihood ratio test, this leads to problems because many parameters that are available in the model will not be used (especially in the leafs of the prefix tree). This test tests whether an increase in the number of parameters leads to a significantly higher likelihood. Thus, if there are many unused parameters, this increase will usually not be significant. Hence, there will be a tendency to accept null-hypotheses, i.e., to merge states.

In the χ^2 test, it also leads to problems because the quality of the approximation of the χ^2 distribution is bad when some of the possible outcomes occur less than five times. In our case, this leads to many problems because we are testing whether two states and their suffix trees (using the determinization process) are similar. In such a tree, the majority of the states (close to the leaf states) will contain very few data.

In the KS test, smaller frequencies also lead to worse approximations. Another problem with the χ^2 tests is that it cannot handle 0 frequencies: we need to divide by $E_{i,q}$ to obtain the value of χ^2. Thus, every expected frequency should be greater than 0.

We deal with the issue of small frequencies in all of the following ways:

- We do not compute anything if the total number of occurrences in both states is less than 10.
- We pool bins of the histogram and symbol distributions if the frequency of these bins in both states is less than 10.
- We apply Yates correction for continuity (Yates 1934) for the χ^2 test if an expected frequency is less than 10.

In addition, as an alternative to Fisher's method we implemented a test that weights the individual tests by the amount of data used to perform these tests, known as the weighted Z-transform, see, e.g., (Whitlock 2005). We explain each of these in more detail.

Stopping computation Stopping the computation when the data is less than 10 in both states effectively stops the recursion of the determinization procedure before the point where it reaches the leaf states of the suffix trees. Hence, every test will be performed on sufficient data, but the overall test will be performed on less data. We do not compute the probabilities/frequencies of the parts of timed strings that reach states with less than 10 occurrences in total. However, the probabilities of the prefixes of such timed strings that reach more frequently occurring states are computed.

Pooling data Pooling is the process of combining the frequencies of two bins into a single bin. In other words, we treat two bins as though it were a single one. For example, suppose we have three bins, and their frequencies are 7, 14, and 5, respectively. Then we treat it as being two bins with frequencies 12 and 14.[7] We perform pooling in both the χ^2 and the likelihood ratio test. In the likelihood ratio test, this effectively reduces the amount of parameters of the tested models, see Figure 5.6. Theoretically, it can be objected that this changes the model using the data.[8] However, if we do not pool data, we will obtain too many parameters for

[7] When the data is ordinal (such a time values), it is also possible to only pool consecutive bins. This maintains the order in the data, but sometimes merges low frequency bins with high frequency ones. It is unclear whether this is beneficial.

[8] Eric Cator, personal communication.

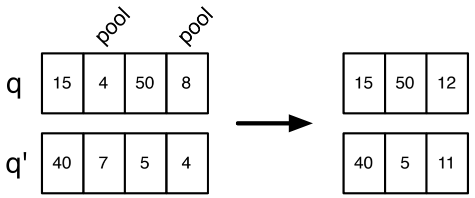


Figure 5.6: Pooling of frequencies from two distributions over bins. We combine two bins if the observed frequencies are less than 10 in both of the two observed distributions. Pooling reduces the number of parameters of the models.

the states in which some bin occurrences are very unlikely. For instance, suppose we have a state in which 1000 symbols could occur, but in fact only 10 of them actually occur. Then according to theory, we should count this state as having 999 parameters. We count it as having only 9.

Yates correction for continuity Yates correction subtracts 0.5 from the difference between the expected and observed frequency before computing the χ^2 value (Yates 1934). As a result, the χ^2 value is reduced and the resulting p-value is increased. Hence, it reduces the probability of rejecting the null-hypothesis due to a bad approximation of the χ^2 distribution in the case of small frequencies.

Z-transform As an alternative to Fisher's method, we implemented a weighted Z-transform, see, e.g., (Whitlock 2005). This transform is similar to Fisher's method but it gives more weight (power) to tests that are based on more data. It computes the following statistic from a set of p-values:

$$Z = \frac{\sum_{1\leq i\leq n} w_i z_i}{\sqrt{\sum_{1\leq i\leq n} w_i}}$$

where z_i is the z value from the standard normal distribution corresponding to the ith p-value and w_i is the weight of the ith test. As a weight we use the amount of examples used to compute the p-value. More specifically, since we always compare two states, and since the certainty of this comparison only increases if the number of examples in both states increases, we use the minimum amount of examples in either of these states as the weight of the test. The Z-statistic is compared to a normal distribution to obtain an overall p-value.

5.3.5 The algorithm

We have just described the tests we use to determine whether two states are similar. The null-hypothesis of all of these tests is that the two states are the same. When we obtain a p-value less than 0.05, we can reject this hypothesis with 95% certainty. When we obtain a p-value greater than 0.05, we cannot reject the possibility that the two states are the same. However, we do not want to test whether two states are the same; we want to test whether to perform a merge or a split, and if so, which one. When we test a merge, a high p-value indicates that the merge is good. When we test a split, a low p-value indicates that the split is good. We implemented this statistical evidence in RTI+ in a very straightforward way:

- If there is a split that results in a p-value less than 0.05, perform the split with the lowest p-value.
- If there is a merge that results in a p-value greater than 0.05, perform the merge with the highest p-value.
- Otherwise, perform a color operation.

Thus, we merge two states unless we are very certain that the two states are different. In addition, we always perform the merge or split that leads to the most certain conclusions.

In every iteration, RTI+ selects the most visited transition from a red state to a blue state and determines whether to merge the blue state, split the transition, or color the blue state red. The main reason for trying out only the most visited transition is that it reduces the complexity of the algorithm. Trying every possible merge and split would take much longer. Additionally, the tests performed using the most visited transition will be based on the largest amount of data. Hence, we are more confident that these conclusions are correct. In state-merging it is very important to make the correct decisions, especially in the first few iterations. A complete overview of the RTI+ for identifying DRTAs from a single very long sequence of observations is shown in Algorithm 5.1.

We claim that RTI+ returns a PDRTA that is equal to the correct PDRTA $\mathcal{A}_t$ in the limit. With equal we mean that these PDRTAs model the exact same probability distributions over timed strings. Because the data has been generated by $\mathcal{A}_t$, and since a PDRTA can be used as a predictor, this means that RTI+ finds the optimal predictor in the limit.

Proposition 5.5. *The result $\mathcal{A}$ of RTI+ converges in the limit to the correct PDRTA $\mathcal{A}'$ (with probability 1).*

Proof. Besides the evidence measure, the algorithm is the same as Algorithm 4.5. Hence, by Proposition 4.5, the algorithm is complete, i.e., capable of returning any PDRTA $\mathcal{A}$. Thus, we only have to show that with sufficient data all correct merges and splits are performed.

With increasing amounts of data, the p-value resulting from any of the tests we use converges to 0 if the two states are different. However, when the two states are the same, there is always a probability of 0.05 that the p-value is less than 0.05. Thus in the limit, RTI+ will perform all the necessary splits, and perhaps some more, and it will never perform an incorrect merge, but at times it will not perform a merge when it should. Not performing a merge or performing an extra split does not influence the language of the DRTA, or the distribution of the PDRTA. It only adds additional (unnecessary) states to the resulting PDRTA $\mathcal{A}$. Thus, in the limit, the algorithm returns a PDRTA $\mathcal{A}$ that is equivalent to the target PDRTA $\mathcal{A}'$. □

Algorithm 5.1 Real-time identification from positive data: RTI+

```
Require: A multi-set of timed strings $S_+$ generated by a PDRTA $\mathcal{A}_t$
Ensure: The result is a small DRTA $\mathcal{A}$, in the limit $\mathcal{A} = \mathcal{A}_t$
  Construct a timed prefix $\mathcal{A}$ tree from $S_+$
  Color the start state $q_0$ of $\mathcal{A}$ red
  while $\mathcal{A}$ contains non-red states do
    Color blue all non-red target states of transitions with red source states
    Let $\delta = \langle q_r, q_b, a, g \rangle$ be a transition from a red to a blue state that is visited by the
    largest number of timed strings from $S_+$, i.e., $\delta = \arg\max_{\langle q,q',a,g \rangle \in \Delta} |\{\tau\tau' \in S_+ \mid$
    $\tau$ ends in $q'$ and $q'$ is blue $\}|$
    Evaluate all possible merges of $q_b$ with red states
    Evaluate all possible splits of $\delta$
    if the lowest p-value of a split is less than 0.05 then
      preform this split
    else
      if the highest p-value of a merge is greater than 0.05 then
        perform this merge
      else
        color $q_b$ red
      end if
    end if
  end while
  Return The constructed DRTA $\mathcal{A}$
```

In addition, RTI+ returns the optimal predictor in polynomial time:

Proposition 5.6. *RTI+ is a polynomial-time algorithm.*

Proof. This follows from Proposition 4.3 and the fact that every statistic can be computed (up to sufficient accuracy) in polynomial time for every state. Since, at any time during a run of the algorithm, the number of states does not exceed the size of the input, the proposition follows. □

Thus, RTI+ identifies PDRTAs in polynomial time in the limit.

Remark We would also like to show that it only requires a polynomial amount of data in order to do so. This should be possible, because none of the statistics we use requires a large amount of data. Moreover, the fact that there exist polynomial characteristics sets for DRTAs (part of Theorem 4.6) should somehow extend to PDRTAs. Then some bounds from probability theory can be used to bound the number of examples that are required to form an unlabeled characteristic set. Constructing such a proof, however, is not easy and will take quite some time. It could be easier to prove efficient convergence in a framework different from identification in the limit, for instance PAC identification. Recently, a PAC identification algorithm has been shown to efficiently identify probabilistic DFAs using few examples (Castro and Gavaldà 2008). We believe that similar algorithms and proofs can be written for the class of PDRTAs since, like DFAs, DRTAs are efficiently identifiable in the limit (see Section 4.3.4).

5.4 Tests on artificial data

In order to discover which of the statistics discussed in the previous section work best in RTI+, we implement RTI+ with all of these different statistics and test it on artificially generated data. The data (and test) sets are generated using the same method we used in the results section of the previous chapter (see Section 4.5.2). The only difference is that we now only generate positive data. Because we can only use positive data, both in the data and test sets, we require a new measure to evaluate the results of these experiments. The accuracy (percentage of correctly classified test examples) is unknown and meaningless because we do not identify a classifier. The size of the solution is not an appropriate measure because this size can be arbitrarily large or small; the identification problem has no consistency constraints that restrict the size of the resulting PDRTA.[9]

Instead, the goal of RTI+ is to find a PDRTA that is consistent with the data in the sense that it correctly represents the distribution of the events in the data. Such a PDRTA can be used to predict future events. Instead of measuring the size or the accuracy of this PDRTA, we can thus measure the predictive quality of this PDRTA. Intuitively, the predictive quality of a model measures how good the model is at predicting unseen (new) data: the more likely the new data is given the distribution of the model, the better the predictive quality of the model. In addition, because PDRTA identification is a model selection problem, another option is to measure the performance of the different statistics in RTI+ using a model selection criterion.

We start this section by describing our experimental setup (Section 5.4.1). We then provide the standard measure for measuring the predictive quality of models: perplexity (Section 5.4.2). We argue and show using our experimental results, that the perplexity measure is unsuitable for the identification of PDRTAs or automata in general. The reason is that we are interested in identifying the structure of a PDRTA, and not in setting the parameters optimally. The test set perplexity measure is influenced greatly by the parameter settings: a correct model will obtain a worse perplexity score than an incorrect model with better parameter settings. Unfortunately, setting the parameters of an identified model correctly can require a lot of data: in our experiments we observed that the actual DRTA that we used to generate the data almost always obtained a worse test set perplexity than a single state DRTA. This is clearly undesirable. We also discuss a standard model selection criterion from statistics: the Akaike information criterion (AIC) (Section 5.4.3). Although the AIC shows more sensible results in our experiments, it does not make use of the unseen data, and hence does not measure the predictive quality of a model. Unfortunately, if we were to calculate the AIC using the likelihood of the unseen data instead of the data sample, we get into the same problems as with the perplexity measure. We therefore propose an easy fix for both these measures: use the unseen data to optimize the parameter settings before computing them (Section 5.4.4). We call the resulting measure the *test-set-tuned* perplexity and AIC, respectively. Because these two measures

[9] There are some possible consistency notions. For example, we can define a model to be consistent if it is Markovian (see Section 2.3.3) with respect to S_+, i.e., if there exists no pair prefixes in S_+ that end in the same state, and that have different suffix distributions in S_+. Unfortunately, it is not clear how to implement this notion of consistency.

display sensible results in our experiments, we use them to evaluate the PDRTAs identified by RTI+ in our experiments (Section 5.4.5).

5.4.1 Experimental setup

The goal our experiments is to discover which statistics work best in RTI+. In addition, we would like to know the size of the PDRTAs that RTI+ can identify in practice. Thus, we need to test RTI+ on many problem instances of different sizes. These problem instances are generated using the same algorithm we used for the identification of DRTAs (see Section 4.5.2). The only difference is that we now use a PDRTA model and that we generate the events using the distributions associated with this model. The original PDRTA models have varying numbers of states and splits. These are set to 4, 8, 16, or 32. The number of possible time values for strings is fixed at 100. The number of histogram bins used in the PDRTA is set to 10. Thus, there are individual probabilities for $[0, 9]$, $[10, 19]$, etc. The probabilities of these bins and the symbol bins are generated by first assigning to each bin a value between 0 and 1, drawn from a uniform distribution. These values are then normalized such that both the histogram values and the symbol values summed to 1. The alphabet size is set to 2, 4, or 8. The number of examples in S_+ is set to 1000 or 2000. The timed strings are generated using an exponential length distribution with an average length of 10. For each combination of these settings we generate 10 different data sets and test sets. The tests sets contained 10000 examples that are generated in addition to the 1000 or 2000 data set examples. In total, we generate $4 \times 4 \times 3 \times 2 \times 10 = 960$ problem instances.

5.4.2 Perplexity

In the grammatical inference and machine learning literature, several quality measures for predictive models exist. For language models, the most common predictive quality measure is the *test set perplexity*. Since RTI+ identifies (timed) language models, it therefore seems sensible to use this measure to evaluate the results of RTI+. The test set perplexity of a model is a value for how surprised (or perplexed) the model is on new data. Intuitively, it measures how sure the model is that the new data would have occurred. We give a small example.

Example 5.4. Suppose there are three possible events a, b, and c, and suppose the model $\mathcal{A}$ gives equal probability of $\frac{1}{3}$ to each of these events. If in the new data an event a occurs, then the perplexity of $\mathcal{A}$ on this data is equal to 3 because the model is as surprised that a occurred as arbitrarily choosing 1 out of 3 possible events. Now, suppose that the probabilities that $\mathcal{A}$ gives to these events are as follows: $Pr(a) = \frac{1}{2}$ and $Pr(b) = Pr(c) = \frac{1}{4}$. In this case, the perplexity of $\mathcal{A}$ on the new event a is 2 because the model is as surprised that a occurred as arbitrarily choosing 1 out of 2 possible events.

Perplexity is commonly used in natural language identification tasks, see, e.g., (Jurafsky and Martin 2000). In natural language identification, the data consists of a database of strings, also known as a corpus. The usual approach is to use only a part of this corpus as a data set for an identification algorithm. The algorithm will construct a model from this data and the remaining part of the corpus is then used to measure the quality of this model. Basically, the better the model is at predicting the observations in this remaining *test set*, the higher the quality score of this model is. The *perplexity* of a model $\mathcal{A}$ on a set of new observations $S'_+ = \{o_1, \ldots, o_n\}$ is computed as follows:

$$perplexity = 2^{-\frac{1}{n}\sum_{1 \leq i \leq n} \log_2(Pr(o_i|\mathcal{A}))}$$

The probability of the ith observation $Pr(o_i \mid \mathcal{A})$ is computed using the model $\mathcal{A}$. Part of the above equation $\sum_{1 \leq i \leq n} \log_2(Pr(o_i \mid \mathcal{A}))$ is the *log-likelihood* of the test set S'_+ under the model $\mathcal{A}$. If the likelihood of the test set increases, then perplexity of the test set decreases. Thus, as a quality measure, a smaller perplexity implies a better predictive model.

A simple upper bound for the perplexity measure can be determined using the uniform distribution, i.e., a distribution that assigns equal probability to every possible observation. It is easy to see that such a distribution always achieves a perplexity equal to the number of possible observations, no matter the actual observations. This distribution can be seen as a form of random-guessing. Every identified model should achieve at least this bound.

For the problem of identifying PDRTAs, every occurrence of a timed symbol is a new observation. Timed strings are sequences of observations. We compute the probability of a timed symbol given $\mathcal{A}$ (and the previous occurrences) as discussed in Section 5.2. Because the perplexity measure is computed for every observation, and not for every (timed) string, this is sometimes called the per (timed) symbol perplexity.

Problems of perplexity

Perplexity measures how surprised an identified model is when observing unseen data. A model that is more surprised makes more mistakes when predicting new data. Hence, it seems a very good measure for the predictive quality of a model. Unfortunately, the perplexity measure has some problems when one wants to identify models instead of its parameters.

In our case, the RTI+ algorithm tries to identify the structure of the correct PDRTA from a data sample S_+. The parameters of an identified PDRTA are set in such a way that they maximize the likelihood of S_+. We then use the likelihood of the test set S'_+ in order to compute the perplexity measure.[10] We compute the perplexity not per timed symbol, but per time histogram bin and symbol. For example, in the case that the alphabet size is 4, there are 40 possible time bin-symbol combinations since we used 10 histogram bins in the experiments. In this case, the uniform distribution over timed symbols (a very simple model) achieves a perplexity of 40. If we compute the perplexity per timed symbol instead, the perplexity of the uniform distribution will be 400 since we use 100 possible time values in the experiments. Our reason for computing the perplexity per time

[10] In order to avoid a likelihood of 0, and hence a perplexity of ∞, we slightly modified the PDRTAs identified by RTI+: for every state-symbol pair for which there exists no outgoing transition, we created a transition with small probability to a garbage state. This can be seen as a very simple smoothing method.

histogram bin instead of per timed symbol is that the data is generated using these bins. As such, this perplexity measure generalizes over time in the way that the timed strings are generated, and hence it gives a better overview of the performance of the identified models.

The perplexity results are shown in Figure 5.7. The figure shows box-plots of the test set perplexity of the different variations of RTI+. In addition, it shows the test set perplexity of the original PDRTA that we used to generate the data, and of a trivial PDRTA consisting of a single state (which is also the target of all transitions). The figure contains three plots: one with alphabet size 2, one with alphabet size 4, and one with alphabet size 8. We depict the results for different sized alphabets because the perplexity is highly dependent on the size of the alphabet.

From the perplexity results we can draw an interesting conclusion: test set perplexity does not (always) correctly test whether an identified model is similar to the model that generated the data. This can be seen by only looking at the first two boxes. The first box shows the results of the original PDRTAs, and the second shows the results of a trivial single state PDRTA. Especially in the case of large alphabets, the single state PDRTA significantly outperforms the actual PDRTAs (the smaller the perplexity the better). This seems counterintuitive, but it actually has a simple explanation. The reason is that we use the data set in order to tune the parameters of the PDRTAs. Because we want to compare the results of RTI+ in a fair way with the original PDRTAs, we also did this for the original PDRTAs. Especially with large alphabets, these PDRTAs have many parameters to tune. Consequently, this requires a lot of data. For instance, tuning a single parameter that determines the probability of heads when flipping a coin with two digits accuracy already requires more than 2000 coin flip examples. In our case, we try to tune many parameters (12, 14, or 18 per state), with many dependencies. However, the data sets we use only contain 1000 or 2000 examples. Since the average length of these examples is 10, we can hope to tune about 10 to 20 of these parameters accurately.

When the alphabet is small, and the number of states is small, this tuning should result in approximately the correct parameter settings. This can also be observed from the plots in Figure 5.7: with a size 2 alphabet, the original PDRTAs do sometimes (in about 65% of all cases) outperform the trivial PDRTA in terms of perplexity. In all other cases, however, it does not even come close. It outperforms the trivial method in only 6% of all cases with an alphabet of size 4, and never with an alphabet of size 8. In fact, the original PDRTA sometimes even performs worse than random guessing. Thus, a correct PDRTA, but with incorrect parameter settings, can result in a high perplexity. This is a serious problem of the perplexity measure and our main reason to look for other quality measures.

5.4.3 Akaike information criterion

Although perplexity does measure the predictive quality of models, it does not take the complexity of these models into account. Intuitively, a large model with many parameters should be able to achieve better predictions than a small model. However, if it uses a lot more parameters in order to achieve a small increase in

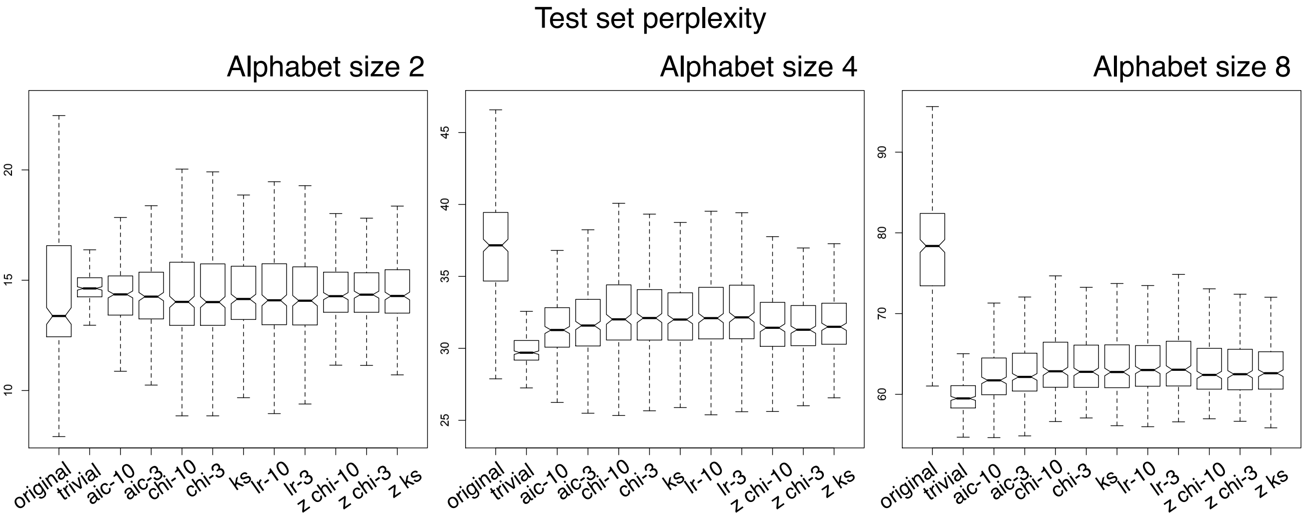


Figure 5.7: Boxplots of the test set perplexity of every implementation of RTI+ on all instances, for varying sized alphabets. From left to right, each plot contains the following boxes: the original PDRTA, the trivial one state PDRTA, the AIC method with 10 and 3 time bins, Fisher's method with a χ^2 test with 10 and 3 time bins, and a KS test, the likelihood ratio test with 10 and 3 time bins, and the weighted Z-transform with a χ^2 test with 10 and 3 time bins, and a KS test.

predictive performance, then the smaller model is to be preferred. This is a simple implementation of Occam's Razor (see Section 2.2). In statistics, model selection criteria are used to make a tradeoff between the complexity of the model and the quality of its predictions. A well-known model selection criterion is the Akaike information criterion (AIC), see, e.g., (Grünwald 2007). The AIC is of a model $\mathcal{A}$ on a set of (new) observations $S'_+ = \{o_1, \ldots, o_n\}$ is computed as follows:

$$AIC = 2k - 2\ln(\max_{par} \prod_{1 \leq i \leq n} (Pr(o_i \mid \mathcal{A}_{par})))$$

where k is the number of parameters of $\mathcal{A}$, par denotes a parameter setting, and $\mathcal{A}_{par}$ is model $\mathcal{A}$ with parameter setting par. The term $\max_{par} \prod_{1 \leq i \leq n}(Pr(o_i \mid \mathcal{A}_{par}))$ is the maximized value of the likelihood of S'_+ under $\mathcal{A}$. We use the AIC to measure the quality of a PDRTA identified by RTI+.

Given a set of models, the AIC can be used in order to determine which one is to be preferred. Hence, it can also be used as a heuristic in the RTI+ algorithm: simply compute the AIC for every alternative and then choose the one that minimizes this criterion. We test this implementation of RTI+ in addition to the different statistics.

Problems of the AIC

The AIC is a model selection criterion. As such, it can be used to decide, given the data set S_+ and two or more models, which model is the most preferred one. In contrast to perplexity, this decision can be made without the use of a separate test set S'_+. One may think that this can lead to the problem that a model that simply reproduces the data set results in a high quality score. This is not true, however, because such a model will require many parameters. In the AIC, the discount factor for parameters will then weigh more heavily than the good predictive quality on the data set.

We computed the AIC for the PDRTAs identified by RTI+. The results are shown in Figure 5.8. The type of plots is similar to those in Figure 5.7. A big difference is that we only used the data sets to determine the AIC values. Consequently, we plot them for the different data set sizes because the AIC is highly dependent on the amount of data that is used to compute it. In contrast to the perplexity measure, the AIC does give the best score to the original PDRTA, and the worst score is given to the trivial PDRTA. In addition, the plots clearly show the difference in AIC performance of each of the individual methods. Therefore, this seems to be a good performance measure.

There is, however, a source of critique for the AIC difference measure: it does not make use of the test set S'_+. In other words, it does not test how good the identified PDRTAs are at predicting *new* data. In trying to resolve this issue, we also tried to compute the AIC values using the test set, instead of the data set. More specifically, we identified the PDRTAs, including all the parameters, from the data set, and then tested the quality of these PDRTAs by calculating the AIC based on the likelihood of the test set. Unfortunately, this showed the same behavior as the perplexity measure.

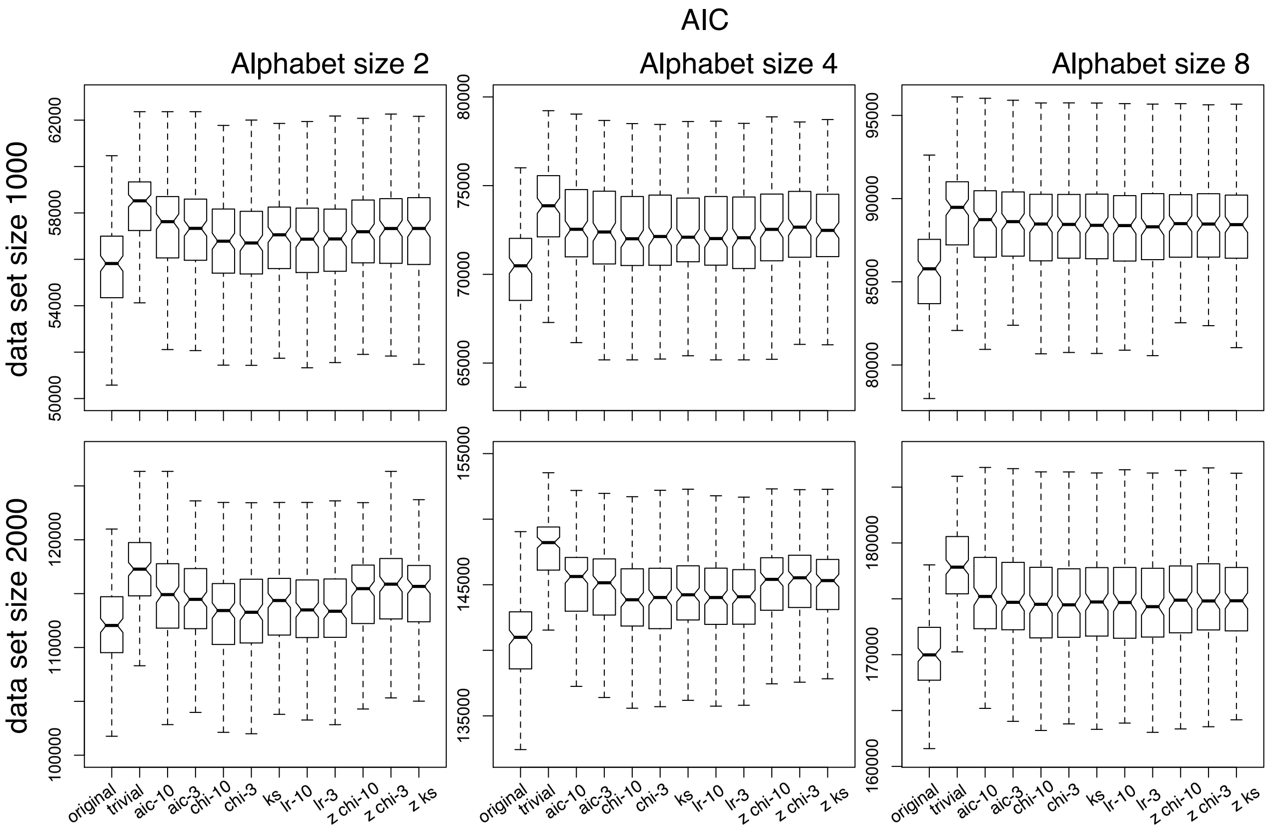


Figure 5.8: Boxplots of the AIC of every implementation of RTI+ on all instances, for varying sized alphabets, and varying data set sizes. See Figure 5.7 for a description of the individual boxes.

5.4.4 Test-set tuned performance measures

The problem of the test set perplexity and test set AIC measures is that there is simply not sufficient data to tune all the parameters of the identified PDRTAs correctly. As such, they over-evaluate smaller models. This can clearly be seen in our results because the single state PDRTA outperforms the actual PDRTA that we used to generate the data in these measures. Since we are interested in identifying the correct PDRTA structure, and not its parameters, this is a serious problem.

We propose a simple fix in order to resolve this: use the test set S'_+, instead of the data set S_+, to tune the parameters. More specifically, we identify the PDRTA models from the data set S_+, use the test set S'_+ to tune its parameters, and then calculated the perplexity and AIC of the identified PDRTA using the test set S'_+. We call the resulting performance measures the *test-set-tuned* perplexity and AIC. These measures effectively separate the problem of tuning the parameters from the problem of identifying the structure: the parameters are set in such a way that maximizes the performance given the identified PDRTA structure. Figure 5.9 shows the test-set-tuned AIC scores of our results.[11] The test-set-tuned perplexity scores are shown in Figure 5.10. Both of these figures show results that are similar to the original AIC scores (Figure 5.8). Hence, they seem to be sensible performance indicators.

This is surprising because we do not know of any work that uses such a measure to evaluate the results of an identification algorithm. Tuning the parameters of an identified model using the test set almost seems like cheating because this maximizes the likelihood of the test set. In fact, a very large tree-like model with many states and splits will perform very good if the perplexity is computed in this way. This is another form of trivial model, and it should not perform well. The main reason why this perplexity measure does work for RTI+ is that we try to minimize the size of the resulting model. In other words, the resulting PDRTAs are as small as possible given the data set. They are local minima in the search space of all possible PDRTAs and the test-set-tuned perplexity measure can be used to compare these local minima. We do believe, however, that the test-set-tuned AIC is a better quality measure because it includes a discount for larger models. The downside of the AIC is that the resulting performance values are difficult to interpret. The perplexity has some meaning in terms of the predictive quality of the model.

A small final issue we have with the test-set-tuned AIC and perplexity measures is that they are highly dependent on the size of the alphabet. A larger alphabet increases the number of possibilities and hence influences the likelihood of the data. Since the original PDRTA always achieves the smallest test-set-tuned value, we can eliminate (most of) this dependence by subtracting the original values from the values of all other approaches. In other words, we pair every result with the optimal result and then calculate the difference. Figure 5.11 shows the results when using such a measure. These plots give a nice overall picture of the performance of each of the individual methods. In the following, we use these measures to

[11] Since we now use the equally sized test sets to compute the AIC, we no longer need to provide different plots for different data sizes.

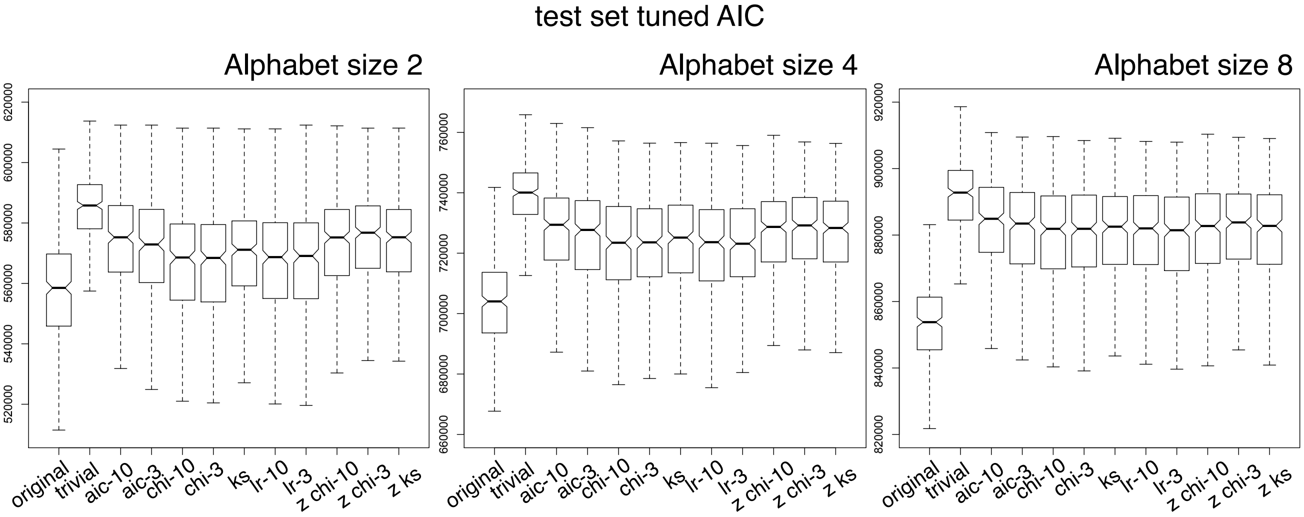


Figure 5.9: Boxplots of the test-set-tuned AIC of every implementation of RTI+ on all instances, for different sized alphabets. See Figure 5.7 for a description of the individual boxes.

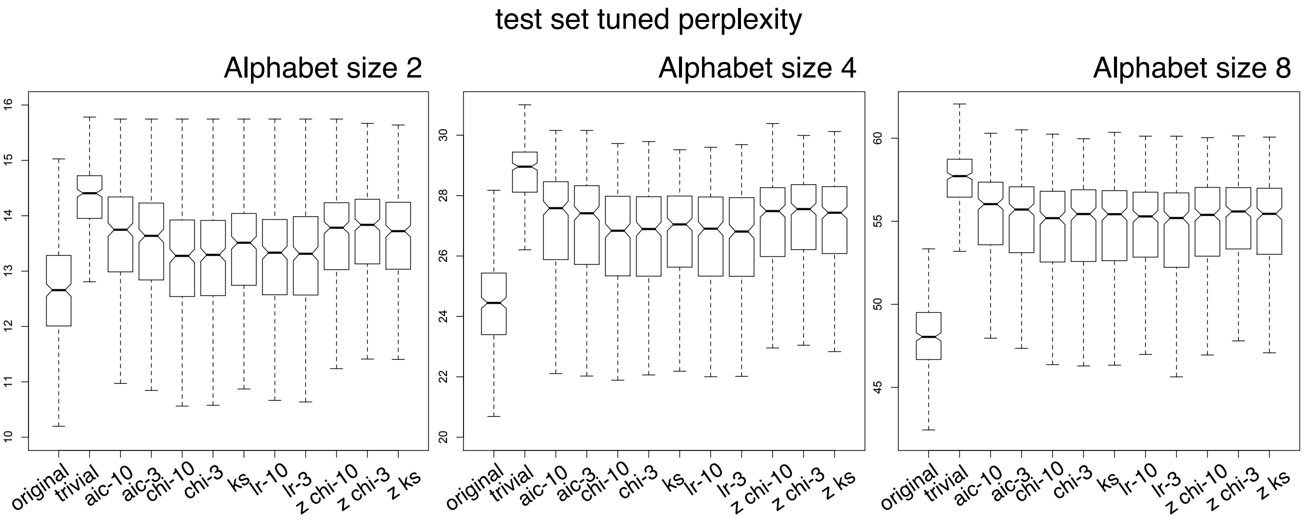


Figure 5.10: Boxplots of the test-set-tuned perplexity of every implementation of RTI+ on all instances, for different sized alphabets. See Figure 5.7 for a description of the individual boxes.

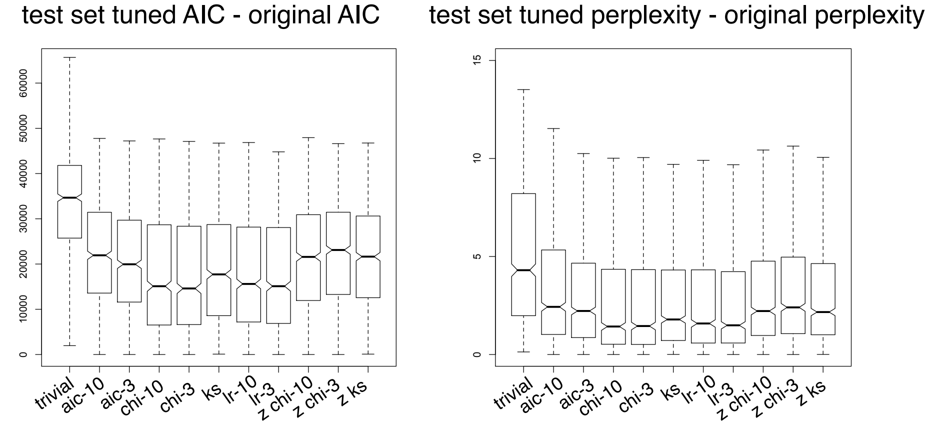


Figure 5.11: Boxplots of the difference of the test-set-tuned AIC and perplexity AIC of every implementation of RTI+ on all instances with the (optimal) original PDRTA. See Figure 5.7 for a description of the individual boxes.

evaluate our results.

5.4.5 Results

Like the experiments for our algorithm for identifying DRTAs from positive and negative data, we performed experiments on a large set of instances where we modified the size settings of the automaton generating the data and test sets. Here, we discuss the results obtained with different implementations of RTI+ from this set. We first discuss the overall results, and then zoom in on each of the individual settings in order to discover the size of PDRTAs that RTI+ can identify.

Overall The overall results are depicted in Figure 5.11. From the plots in Figure 5.11, we can conclude that the χ^2 and likelihood ratio implementations significantly outperform every other method. This can be seen in the figures by looking at the notches (triangular cuts) in the box-plots. A notch depicts (roughly) the 95% confidence interval of the median. Thus, if the median of another boxplot is outside of this range, then the median of these two plots differ significantly. The other implementations include the AIC method, the χ^2 combined with Kolmogorov-Smirnov tests, and the weighted Z-transform consensus test. Also, all implementations perform significantly better than the trivial PDRTA that consists of a single state.

The differences between the χ^2 and likelihood ratio implementations are not large, but still significant: a paired t-test for independence applied to the results of χ^2 and likelihood implementations with 10 bins results in a p-value of about 0.058 for test-set-tuned perplexity, and about 0.0036 for the test-set-tuned AIC. In both cases, the χ^2 implementation performs a little bit better than the likelihood ratio implementation. The mean differences are about 0.032 for the test-set-tuned perplexity and 312 for the tes-set-tuned AIC. The same test applied to the results of either the χ^2 or the likelihood implementations with any of the other implementation result in very small (typically less than 1×10^{-10}) p-values.

Interestingly, when we perform the same test to the results of χ^2 and likelihood implementations with 3 bins, we obtain a different picture. The resulting p-values are 0.0011 for the test-set-tuned perplexity and 0.36 (not significant) for the test-set-tuned AIC. In this case, the likelihood ratio implementation performs slightly better with mean differences of 0.05 for the test-set-tuned perplexity and 91 for the test-set-tuned AIC. This seems to indicate that using a smaller number of histogram bins is beneficial for the likelihood ratio implementation. Indeed, when we perform a paired t-test applied to the results the likelihood ratio implementation using 3 and 10 bins we obtain p-values of about 0.000038 for test-set-tuned perplexity, and about 0.00068 for the test-set-tuned AIC. Hence these are significantly different. The same test applied to the χ^2 implementation results in p-values of about 0.14 and 0.27, respectively. It is a little surprising that using 3 histogram bins yields better results for the likelihood ratio implementation since the data is generated using 10 bins. A possible reason for this is that the 10 bins implementation results in more pooling of the different bins, and hence there is some loss of information that is available in the 3 bin method. The differences however are again very small: the resulting mean differences are 302 for the test-set-tuned AIC, and 0.06 for the test-set-tuned perplexity.

Different sized alphabets Figure 5.12 shows the plots of the test-set-tuned AIC and perplexity for different sized alphabets. The main observation we make from these plots is that they show the same performance results as the overall plots for the smaller alphabet sizes 2 and 4. However, for an alphabet of size 8 the difference in performance between the different implementations of RTI+ is very small. Fortunately, they all still perform better than the trivial PDRTA. Another observation is that the variance of the performance measures increases with increasing alphabet size. This can easily be explained by observing that the range of these values also increases with increasing alphabet size.

Different amounts of data It is interesting to see the influence of more data on the performance of RTI+. Plots of different-sized data sets (1000 and 2000) are shown in Figure 5.13. From the plots, a clear performance increase can be seen when going from 1000 examples to 2000 examples. Of course, this is logical because the certainty that a merge or split is correct increases with more data.

Different number of states The results for the different settings for the number of states of the original PDRTA are shown in Figure 5.14. These plots show a clear difference between the results obtained with 4 and 8 states and those obtained with 16 and 32 states. For the larger number of states, there is a large variance and the difference between the different methods is very small. Also, the difference with the trivial method gets smaller as the size of the original PDRTA increases. For the results obtained with 4 states in the original PDRTA, this difference is quite large.

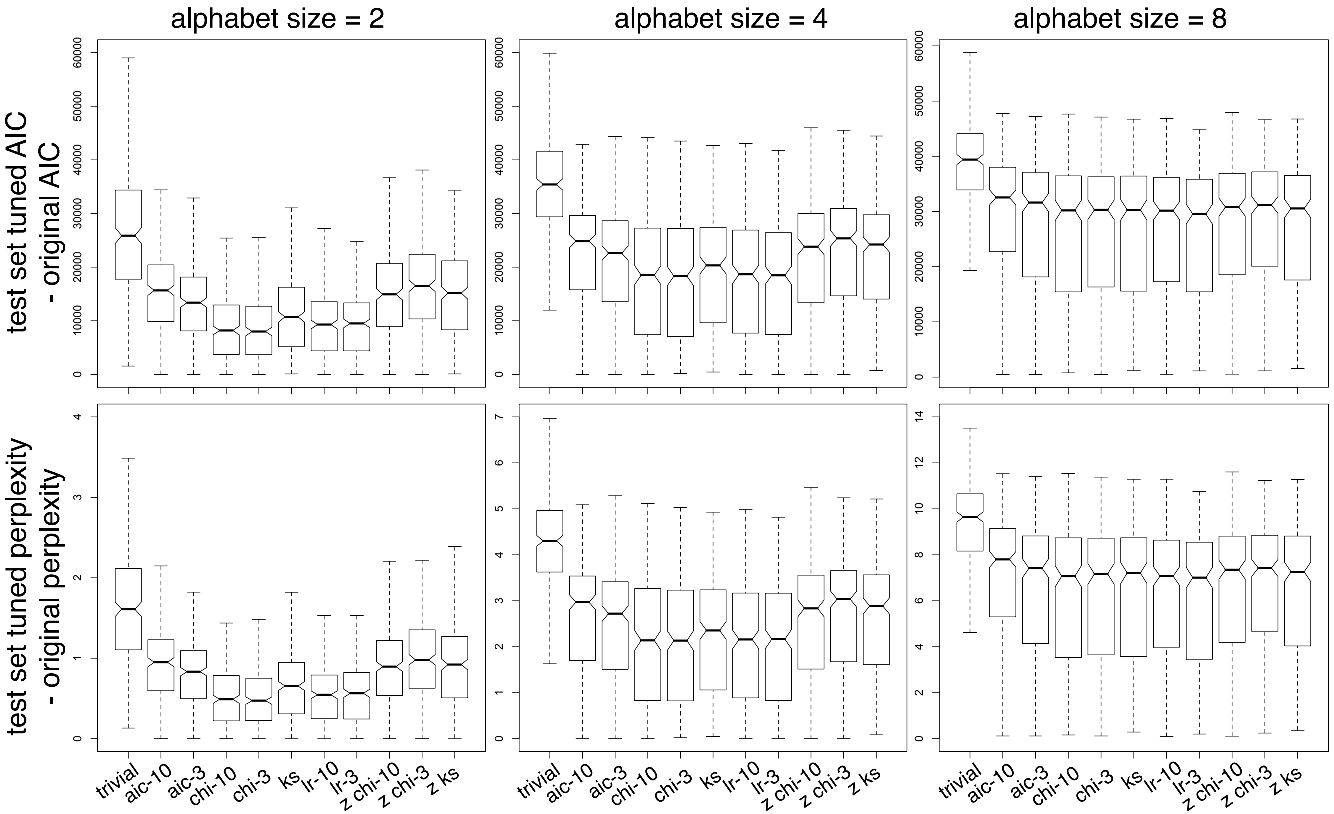


Figure 5.12: Boxplots of the difference of the test-set-tuned AIC and perplexity AIC of every implementation of RTI+ on all instances with the (optimal) original PDRTA, for varying sized alphabets. See Figure 5.7 for a description of the individual boxes.

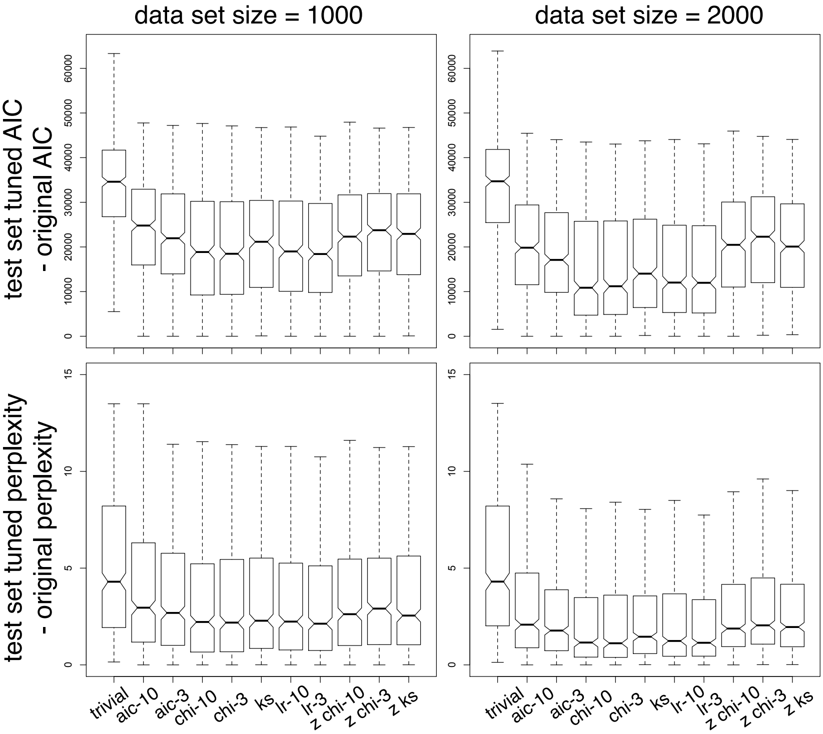


Figure 5.13: Boxplots of the difference of the test-set-tuned AIC and perplexity AIC of every implementation of RTI+ on all instances with the (optimal) original PDRTA, for varying sized data sets. See Figure 5.7 for a description of the individual boxes.

Unfortunately, it is difficult to use these measures to determine how large the PDRTAs are that RTI+ can reliably identify from 1000 or 2000 example timed strings. The measures can only compare the different implementations with each other. However, the performance on a PDRTA of size 4 seems to be very good, especially for the χ^2 and likelihood ratio implementations. We visually compared some of the results obtained on the smaller sizes with the original PDRTA. For the size 4 PDRTAs, in almost all cases, the only difference between these models is that some of the identified clock guards are slightly off. For the size 8 PDRTAs, this is still the case, but it occurs more often that incorrect transitions and additional or less states are identified. Figure 5.16 shows an example of an original and an identified PDRTA of size 8, with a size 4 alphabet. It is clear that in this example the most common mistake is the incorrect identification (or absence) of a clock guard. From such observations we can conclude that our method works well on these smaller DRTAs, and that correctly identifying the clock guards is difficult and requires a lot of data. In addition, we inspected some results on size 32 PDRTAs, and on these instances, almost always too few states are identified. From this we conclude that 2000 examples of average length 10 is too few data in order to identify a 32 state PDRTA correctly.

Different number of splits The other size value for PDRTAs is the number of splits we used in its construction. The results for varying numbers of splits are shown in Figure 5.15. The number of splits has less influence on the performance measure than the number of states. This makes sense since each additional state creates a new probability distribution over a timed symbol, but each additional split (transition) does not. In the plots of Figure 5.15, still the same performance results can be seen as the overall picture. The identification problem does become more difficult with increasing number of splits, but how much more difficult is hard to say. Even in the case of many splits, there is still a significant difference between the different implementations of RTI+.

Summary We presented the results of RTI+ using new performance measures. From the different plots in this section, and the additional paired t-tests we performed, we can conclude that the χ^2 and the likelihood ratio implementations consistently (independent of the size of the problem) perform best and produce comparable results. The χ^2 implementation uses a χ^2 test on both the symbols and the time bins, and uses Fisher's method as a consensus test. Using either of these implementations, it should be possible to identify a PDRTA with 8 states and an alphabet of size 4. We show the results in this setting for 2000 example timed strings in Figure 5.17.

The plots in the figure show that the original PDRTA that we used to generate the data (the optimal PDRTA) achieves a median perplexity of about 25. The results of the RTI+ algorithm using the χ^2 and the likelihood ratio statistics achieve a median perplexity a little under 26. This means that the optimal distribution and the results of RTI+ predict events as good as guessing 1 out of 25 and 26 events, respectively. Thus, our results only requires 1 additional event to guess from. In contrast the uniform distribution over timed events guesses 1 out of 40 events, and the trivial single state model guesses 1 out of 29 events.

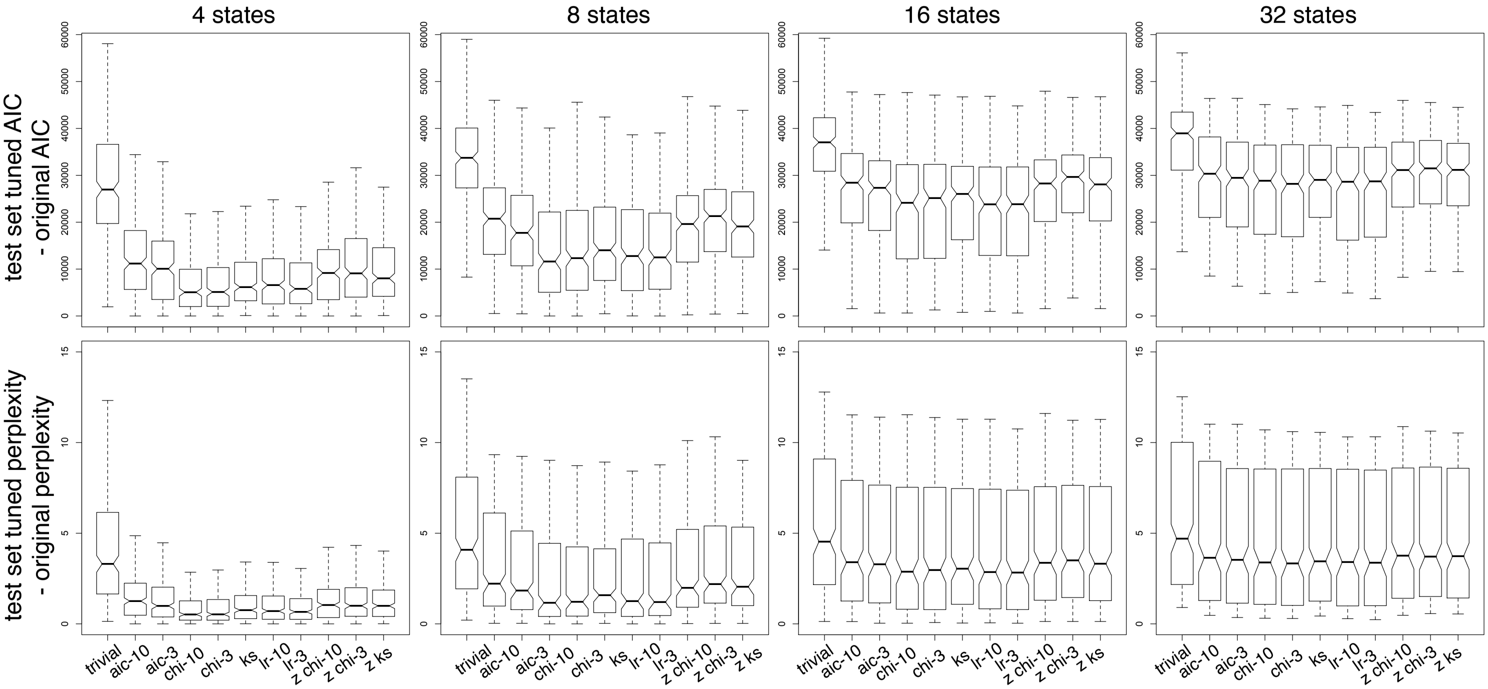


Figure 5.14: Boxplots of the difference of the test-set-tuned AIC and perplexity AIC of every implementation of RTI+ on all instances with the (optimal) original PDRTA, for varying amounts of states in the original PDRTA. See Figure 5.7 for a description of the individual boxes.

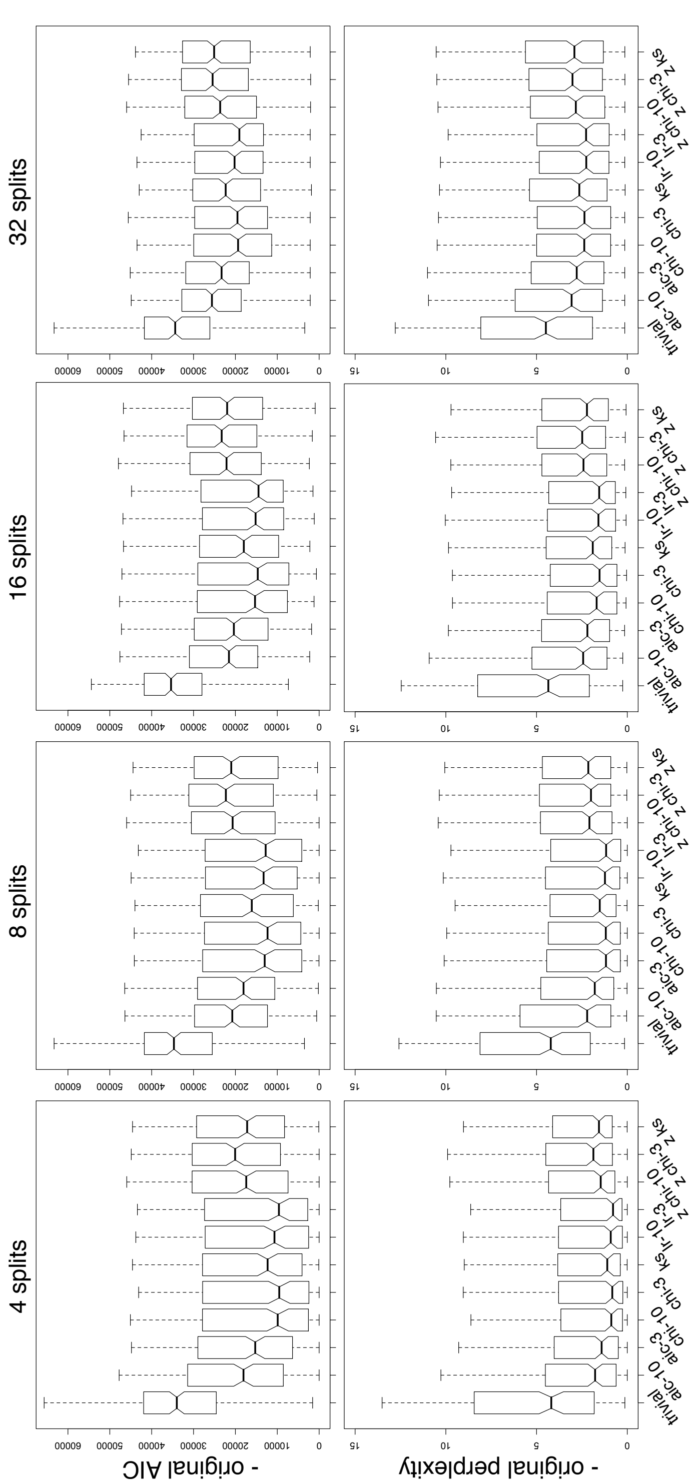
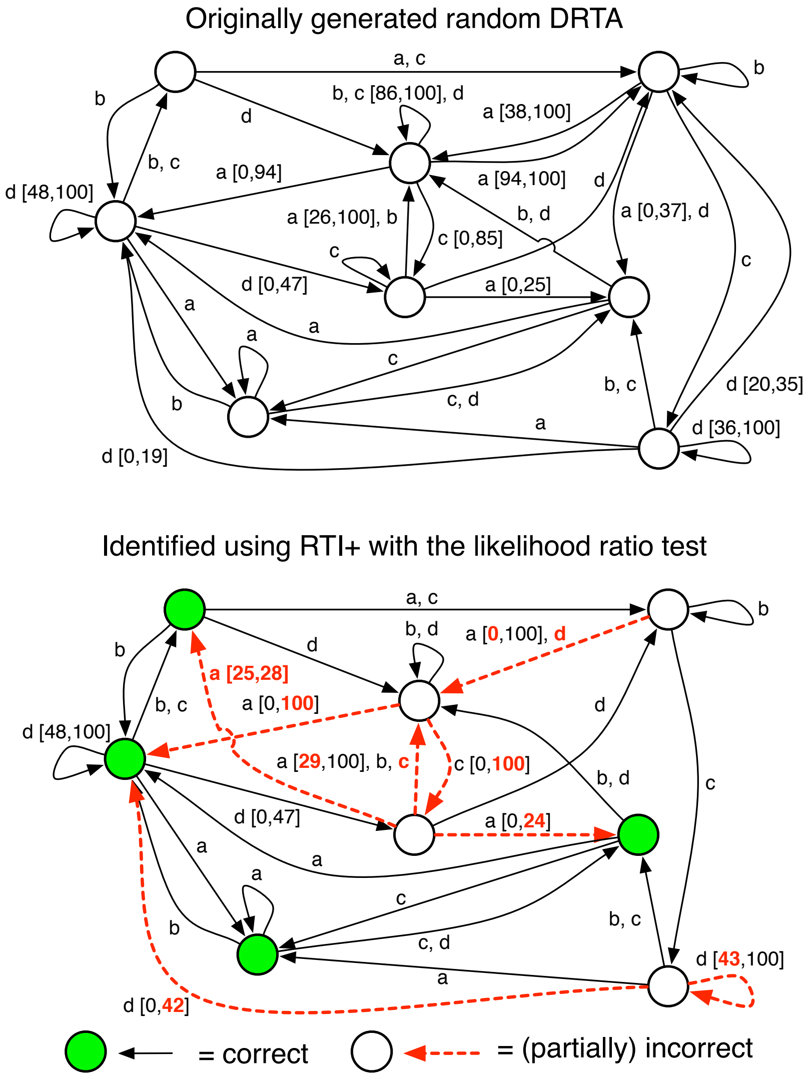


Figure 5.15: Boxplots of the difference of the test-set-tuned AIC and perplexity AIC of every implementation of RTI+ on all instances with the (optimal) original PDRTA, for varying amounts of splits in the original PDRTA. See Figure 5.7 for a description of the individual boxes.


Originally generated random DRTA

Identified using RTI+ with the likelihood ratio test

Figure 5.16: An example of an original DRTA (top) and the DRTA identified by our algorithm using the likelihood ratio statistic (bottom). The dashed lines are (partially) incorrectly identified transitions. The solid states are correctly identified, including all outgoing transitions. Note that the incorrectly identified transitions often still lead to the correct behavior because a large part (or all) of the timed strings that fire such a transition reach the correct target state.

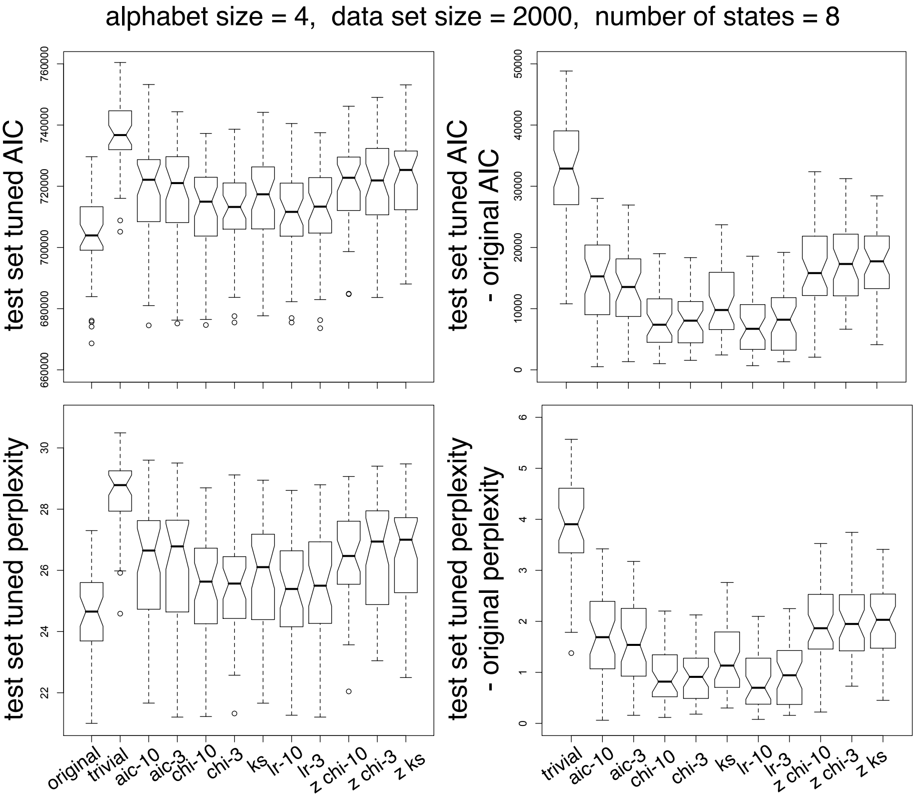


Figure 5.17: Boxplots of the test-set-tuned AIC and perplexity AIC of every implementation of RTI+ on all instances with an alphabet of size 4, an original PDRTA with 8 states, and a data set of size 2000. In addition, the difference with the original PDRTA of the same values and instances. See Figure 5.7 for a description of the individual boxes.

However, it is difficult to draw precise conclusions from these results because the performance measures are relative measures. In other words, they can be used to compare the performance of different methods, but they do not assign an overall quality value to the results. In fact, we do not know of any overall quality value for PDRTAs or probabilistic automata in general. We believe RTI+ can infer these PDRTAs because RTI+ performs well on those data sets, and the resulting PDRTAs are visually similar (see for example Figure 5.16).

5.5 Discussion

In the previous chapter, we described the RTI algorithm (Algorithm 4.5) for identifying deterministic real-time automata (DRTAs) from labeled data. In this chapter, we showed how to adapt it to the setting of unlabeled data. The result is the RTI+ algorithm (Algorithm 5.1). The RTI+ algorithm is a polynomial time algorithm that converges efficiently to the correct PDRTA in the limit with probability 1. It uses statistical tests in order to determine which states to merge and which transitions to split. We introduced a couple of statistical tests for this purpose. Although these tests are designed specifically for the purpose of identifying a PDRTA from unlabeled data, they can be modified in order to identify probabilistic DFAs. Hence, they also contribute to the current state-of-the-art in probabilistic DFA identification.

We tested the performance of the RTI+ algorithm with different statistical tests on artificially generated data. For these tests, we proposed a different way to compute the perplexity and AIC quality measures: use the test set to tune the parameters of the identified models. We call the resulting measures *test-set-tuned perplexity* and *test-set-tuned AIC.* These measures effectively separate the problem of tuning the parameters from the problem of identifying the structure: the parameters are set in such a way that maximizes the performance given the identified PDRTA structure. Since we are only interested in identifying the correct PDRTA model, and not its parameters, we use these test-set-tuned measures in our experiments.

An interesting conclusion of these experiments is that in terms of data requirements it is often easier to identify a model than to set its parameters correctly. Because of this, the traditional test set perplexity measure over-evaluates smaller models. In our experiments, small trivial models outperform the actual model that generated the data when perplexity is computed in this way. We believe this to be the reason why a state-merging algorithm for the identification of probabilistic DFAs (state-merging) is often reported to be outperformed by simple N-gram models, see, e.g., (Kermorvant and Dupont 2002). It would be interesting to investigate what happens with this performance if our test-set-tuned perplexity or test-set-tuned AIC measure is used as a performance measure.

Our results also show which of the introduced statistics achieves the best performance. Both the χ^2 and the likelihood ratio statistics perform significantly better than any of the other statistics. Between themselves they produce comparable results. The achieved performance using either of these two statistics is sufficient in order to apply RTI+ in practice. This application is the topic of the next chapter.

CHAPTER 6

Inference of a real-time process

6.1 Introduction

The techniques we developed in the previous chapters have many potential practical applications. In this chapter, we explore one such application in order to give a proof of concept for their use: we apply it to the problem of identifying truck driving behavior monitoring system, i.e., the motivating example mentioned in the introduction (Chapter 1).

The data we have at our disposal to identify this process from consists of only positive data; assigning the correct labels to this data unfortunately is very difficult and time consuming. From this data, we want to identify a DRTA model that we can use to monitor the driving behavior in new data, i.e., to use it as a classifier. The data consists of unlabeled (positive) time-stamped event sequences (examples). Our approach is to first identify a PDRTA model using RTI+ (Algorithm 5.1) from this data, and then to use a small amount of labeled (positive and negative) examples to label some of the states of the identified model. The labels of these states can be used to classify the new data. This is a general method that can be used to learn from many unlabeled examples, and just a few labeled examples. The setting of many labeled and some unlabeled examples is known as *semi-supervised learning*, see, e.g., (Bishop 2006). Such a setting occurs in many practical applications because it is often too difficult and time consuming to assign labels to the observed examples.

In addition to there being only unlabeled data, there is an additional restrictive property in our application domain: the process under observation is *continuous*, i.e., it never stops producing events. We can only observe a finite part of the infinite sequence produced by this process. In other words, we need to identify a PDRTA from a single unlabeled time-stamped event sequence. This setting coincides with the theory from *computational mechanics* (see Section 2.4.4). In (Shalizi and Shalizi 2004), this theory has been used to identify an HMM from such data. In a similar way, we show how this theory allows us to identify a PDRTA model for such a process using the RTI+ algorithm.

This chapter is structured as follows. We first describe the background of our application, its goals, and why we choose our approach in order to achieve these goals in Section 6.2. Afterwards, we explain the data that we collect in this application, and how we transform this (using computational mechanics) into a form suitable for this approach in Section 6.3. Due to the nature of this data, we make some small changes to the DRTA models that we try to identify. These changes are described in Section 6.4. In Section 6.5, we explain the main topic of this chapter, that is how to identify a model using only positive data, and still use it as a classifier for new data. We discuss some results obtained using a classifier we identified and implemented for the application in Section 6.6. We end this chapter with a summary and some pointers for future research in Section 6.7.

6.2 The Van der Luyt case

The Van der Luyt Transport company is a transport company located in Oegstgeest, the Netherlands. The company has considerable interest in technologies that lower the cost of transport by reducing the amount of fuel used by its trucks. The fuel usage has a close relation to the driving behavior of the driver, being the reason for Van der Luyt Transport to install sensors on some of its trucks. These sensors, the technology, and expertise required to receive and record accurate data are provided by the Squarell Technology and CD Systems companies, both located in Lisse, the Netherlands.

The sensors measure performance values of the truck, such as speed, engine temperature, break application, fuel level, etc. Figure 6.1 shows an example of the measured sensor values. The measured values make it possible to detect patterns in the driving behavior of the driver. This behavior has a direct influence on these values. Van der Luyt Transport wants to be able to detect patterns in real-time from these values in order to give feedback to the driver. The driver can then be notified when he or she is conducting behavior which is considered not appropriate.

An example of such non-appropriate behavior is the so-called *harmonica behavior*. In Chapter 4, we already showed that a DRTA model can give a concise and intuitive representation of this behavior, see Example 4.1. Moreover, because it is easy to compute the runs of a DRTA, such a model can be used to detect this behavior efficiently in real-time from the sensor values. We are interested in modeling driving behavior as DRTAs, detecting them in real-time, and giving feedback to the truck driver.

In order to model behavior as DRTAs, we first need to *discretize* the sensor values into basic events. These events include: driving on the highway, speeding up, slowing down, breaking, signaling, using cruise control, etc. We then need to construct models that use these events as symbols. However, due to the lack of sufficient expert knowledge required to construct these models by hand, we will try to identify these models automatically from data collected by the different sensors. In doing so we effectively identify a model that describes the language of a good

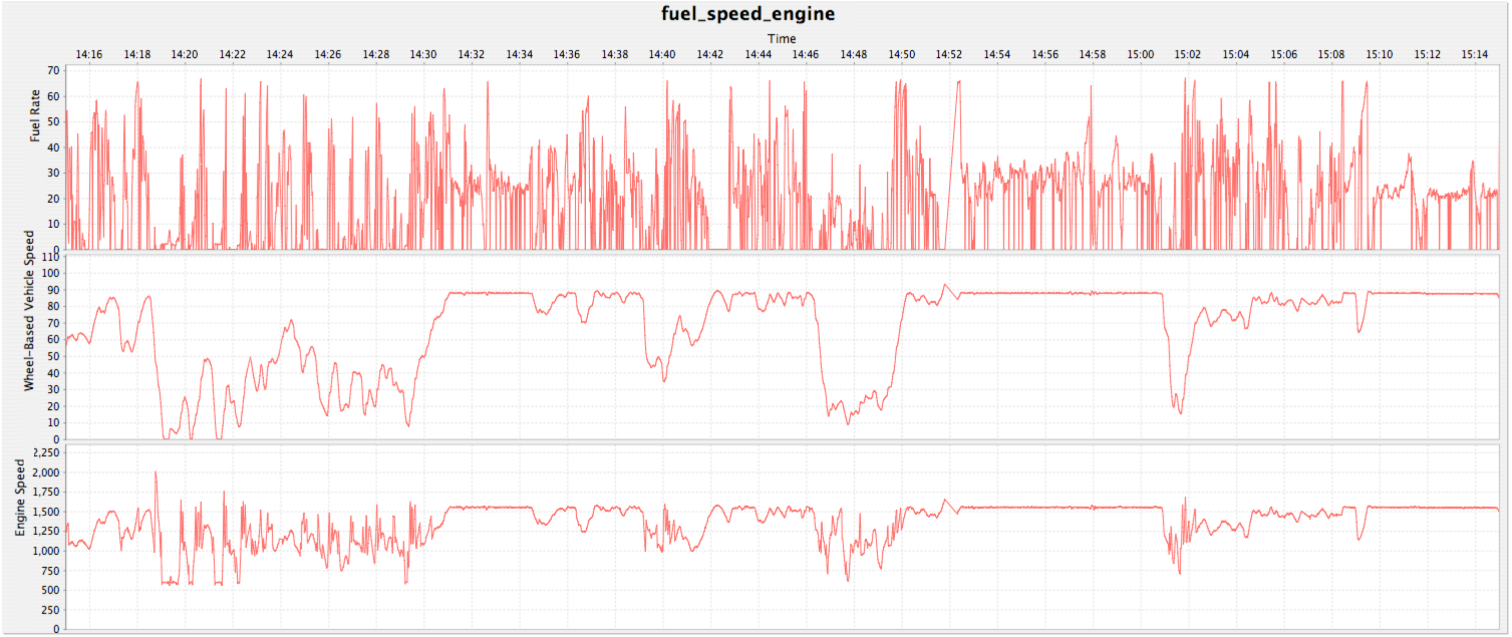


Figure 6.1: The values of three sensors installed on trucks at Van der Luyt. The first graph shows the fuel rate in liters per hour. The second shows the vehicle speed in kilometers per hour. The third shows the engine speed in rotations per minute.

or bad driver. This approach of identifying a language model in order to describe patterns in data is known as *syntactic pattern recognition* (Fu 1974). The main reason for using such a method is that we can use the identified models not only to find out whether a driver is driving well or badly, but also to give insight into the exact road behavior of good and bad drivers, i.e., to learn what feedback to give to drivers in order to reduce their fuel usage.

The reasons why we use a timed automaton instead of a regular automaton are twofold. First, the time between vehicle speedups and slowdowns is significant for classifying truck behavior. For instance, a sequence of fast changes from slowing down to speeding up and vice versa indicates driving in a city, while a sequence of slow changes indicates driving on a freeway. Second, there is timed information available and there is no reason not to use it directly. In the previous chapters we have shown that not using this data directly can result in more difficult (or less efficient) identification problems.

We now describe the data we obtained from the trucks at Van der Luyt, and how we transformed it into a form that can be used to identify real-time automata.

6.3 Transforming the sensor data

The data we obtained from the trucks were 15 complete round trips from the Netherlands to Switzerland or England and back. One round trip typically takes two full days. These trips were driven on different dates, with different weather conditions, and with different drivers. During these trips, a large amount of sensors measured many (around 50) different values. These measurements were recorded at a rate of one per second.

Identifying a model for all of these sensors is possible but unnecessarily complicated for our purpose. We only want to give a proof of concept for the techniques we developed in the previous chapters. Therefore, we focus our attention on three of the sensor values: *speed*, *engine rotations per minute* (RPM), and *fuel usage*. In this section, we describe the preprocessing transformations we applied to each of these three values.

6.3.1 Discretizing timed events

The data obtained from the trucks are recorded in numeric values. Since our method requires a finite alphabet, preferably of a small size, we have to encode these values using such an alphabet. This process is called *discretization*. Ideally, we would like the symbols of our alphabet to represent the actions of the system. In our case, we would like to have symbols that represent: speed-ups, slow-downs, a turn left/right, a bump in the road, etc. Unfortunately, the problem of finding patterns in the sensor values that are indicative of these kinds of events is not an easy one and solving it will require additional research. In (Roddick and Spiliopoulou 2002), different methods that could be used to learn and detect this kind of patterns are surveyed.

The method we use to discretize the data is much simpler: we divide the range of a sensor value into different regions, and whenever the value enters a region, we treat it as the occurrence of an event associated with that region. The time value

symbol	*a*	*b*	*c*	*d*	*e*	*f*
speed (km/hour)	5	30	50	80	90	∞
engine (RPM)	1200	1400	1550	∞		
fuel (liters/hour)	3	10	20	50	80	∞

Figure 6.2: The regions of the discretization routine per sensor. The numbers are upper bounds of the respective symbols. The lower bounds are equal to the upper bound of the previous symbol plus one, except for the first symbol, in that case it is zero.

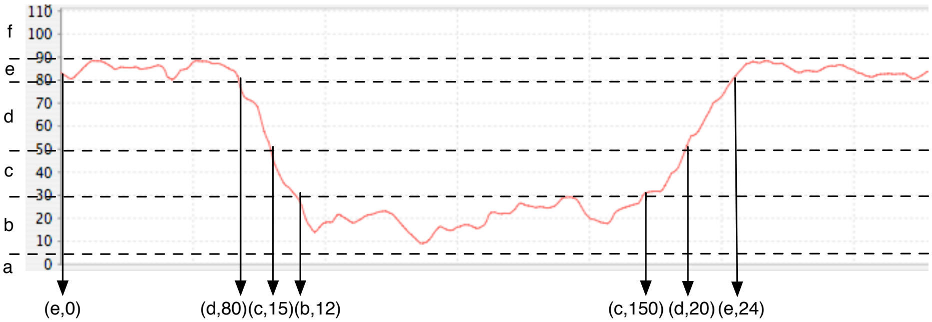


Figure 6.3: The discretization routine used by our algorithm. Whenever the sensor value exceeds one of the bounds of Figure 6.2, plus or minus a small additional region (in this case an additional 2 km/h), an event is generated corresponding to the region the value has entered. The time value of the event is the time that has elapsed since the previous event.

of the occurrence is the time that has elapsed since the previous event occurrence. We use expert knowledge from Squarell Technology to create an intuitive division of the sensor values into regions. Figure 6.2 shows these regions. The main benefit of using an intuitive division is that it makes the models that our method identifies easier to interpret. For instance, whenever a high-speed event (speed *e*) occurs, we know that the truck is driving on a highway. Thus, if we are interested in discovering the behavior of drivers on the highway, we only have to zoom in on the parts of the model where this event occurs.

In our discretization method, there is one additional modification we apply in order to ensure that the event occurrence does not change constantly when the sensor value is near a region boundary. For instance, when the truck is driving with an approximately constant speed of 30 km/h, it will sometimes drive 31 km/h, and sometimes 29 km/h. We do not want this to generate an alternating sequence of low-speed (speed *b*) and average-speed (speed *c*) events. Instead, we either generate a single low-speed or average-speed event, depending on the previous event. We achieve this by adding additional regions that surround the region boundaries. As long as the sensor value is within such an additional region, we do not treat it as an event occurrence, even if it crosses a boundary. The sensor value has to cross this additional region before we treat it as the occurrence of an event. The whole discretization process is depicted in Figure 6.3.

6.3.2 Using computational mechanics

Using the discretization procedure, we obtain from every round-trip of a truck, and for every sensor, a very long sequence of timed symbols. We can still apply the RTI+ algorithm to this data by making use of the theory from computational mechanics (see Section 2.4.4). Computational mechanics models a process $O = \ldots O_{-2}O_{-1}O_0O_1O_2\ldots$ as a bi-infinite sequence of random variables (observations). These variables can take any value from a *countable* set, which in our case is the set of timed events. As before, we use $\mathbb{N}$ to represent the time values of event occurrences. As a consequence, the set of possible observations is still countable. Thus, adding time values to the events does not influence the model of a process as used in computational mechanics.[1]

The goal of computational mechanics is to predict the future of a process based only on an observed finite history $\overleftarrow{\tau} = (a_1, t_1)\ldots(a_n, t_n)$. We would like to identify a PDRTA using the RTI+ algorithm described in the previous chapter. This adapted algorithm requires a set of positive example strings S_+. We can create this set S_+ from the observed history $\overleftarrow{\tau}$ by selecting (random or all) subsequences of $\overleftarrow{\tau}$. A problem of these subsequences is that they are allowed to start and stop at arbitrary points in time. In order to generalize (identify/learn) over such arbitrary subsequences, we have to make the assumption that the process is stationary:

Definition 6.1. *(stationary process) A process* $O = \ldots O_{-2}O_{-1}O_0O_1O_2\ldots$ *is* stationary *if it is time-translation invariant, i.e., if* $Pr(O_i \ldots O_{n+i} = (a_1, t_1)\ldots(a_n, t_{n+1})) = Pr(O_j \ldots O_{n+j} = (a_1, t_1)\ldots(a_n, t_{n+1}))$ *for all* $i, j \in \mathbb{Z}$, $a_i \in \Sigma$, *and* $t_i, n \in \mathbb{N}$.

The assumption of stationarity seems restrictive. However, without this assumption it is very difficult to identify a model from some observed history. When a process is non-stationary, the exact (non-relative) time of occurrence of each event can matter to the future behavior of the process. The type of models we try to identify use only the relative times of occurrence as information. Of course, it is also possible to define a model that does use the exact times of occurrence of events as input. However, since there are many more possible exact occurrence times, identifying such a model from data is a much more difficult problem.

The timed strings in S_+ can be used to identify the so-called *causal states* of a stationary process. Essentially, the causal states form a DRTA $\mathcal{A}'$ that represents behavior (a probability distribution over timed strings) identical to a non-deterministic version of the original DRTA $\mathcal{A}$. This non-deterministic version has an identical structure as the original DRTA, but has the complete set of states as possible start states. We illustrate this concept for the problem of identifying a DRTA using an example.

Example 6.1. In Figure 6.4, a DRTA is depicted along with its causal states. The causal states also form a DRTA. The start state of this DRTA corresponds to

[1] We are not sure what the consequences would be of using the real numbers (a non-countable set) to represent time.

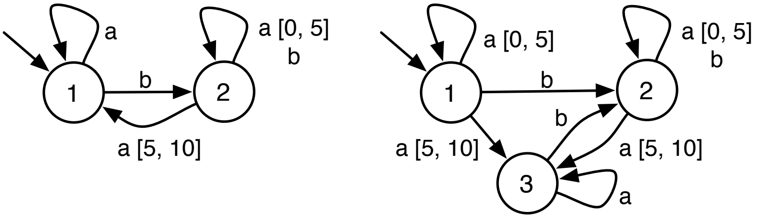


Figure 6.4: A deterministic real-time automaton $\mathcal{A}$ (left) and its causal states $\mathcal{A}'$ (right).

all (or any) of the states of the original DRTA. In other words, we do not know whether the original DRTA is in state 1 (left) or state 2 (right). The occurrences of events can be used to determine the state of the original DRTA. For example, the occurrence of a b event ensures that the next state is state 2, since all transitions with b as label have state 2 as their target. Similarly, the occurrence of an a event with a time value greater than 5 ensures that the next state is state 1. When an a event occurs with a time value less than 4, we do not know whether 1 or 2 is the next state. Thus, in this case, the target state in the causal DRTA is the state that corresponds to both state 1 and 2, i.e., the start state.

The causal states are the deterministic equivalent of a non-deterministic automaton. Therefore, it is possible that the blow-up in the number of states is exponential. This is the price we have to pay for identifying the causal states instead of the original automaton. We have no choice, however, because identifying the original DRTA is only possible if the timed strings are guaranteed to start in the start state. In our application setting, we cannot obtain sufficient data of this form in order to identify a DRTA model. Together with the transitions between them, these causal states form the smallest optimal predictor for future events.

Combining all the data we have available, we obtain for the speed sensor about 50, 000 symbol occurrences in total. For the engine and fuel sensor we obtain about twice this amount. From this data, we took 10, 000 random subsequences of length 20. The reason for using only parts of the data is that more data makes the algorithm run slower. Since the algorithm is still in an experimental phase, we wanted to be able to run it multiple times and try out different settings. For instance, we first tried using a moving average as an alternative to the discretization routine described in the previous section. However, this turned out to give too much of a delay in the time values of events.

6.4 The modified PDRTA model

From the data we have to identify a model. Since we only have positive data available, this model is the PDRTA described in Section 5.2. However, before the RTI+ algorithm can be used to identify a PDRTA, we have to specify the bins of the histograms that model the time distributions in the PDRTAs. Because the timing of the events is different in the different sensors, we have to specify these for every individual sensor. In addition, after initial trial runs of our algorithm, it turned out that a PDRTA is too powerful for the kind of data we are dealing with in our application; the algorithm identified too many clock guards, resulting in an unintuitive model. In the following we first discuss the choices of the histogram bins, and then the changes we made to the PDRTA model in order to make it less powerful.

6.4.1 Setting the histogram bins

In order to determine good histogram bins for the time distributions of the different sensors, we first plotted the density of the time values for every sensor using a histogram. These histograms are shown in Figure 6.5.

The speed histogram shows that most events are concentrated around 6 and 10 time ticks. We need to make histogram bins that can occur sufficiently often to detect (dis)similarities in the time distributions of states. For example, in the case of 4 bins, using the interquartile ranges (bounds at 25%, 50%, and 75% of the data) seems to make sense. In addition, however, we should use our knowledge of these events to determine suitable values. For instance, a speed change within 5 seconds is extremely quick, and should be treated differently from the more common (and normal) speed change within 6 to 20 seconds. Combining these two sources of information, we came up with the following values for the histogram bins: $[0, 5], [6, 20], [20, 50], [51, \infty)$.

For both the RPM and the fuel sensor, the histograms show a little over one third of all events occur within one time tick. The reason for this is that the data was recorded using a sampling rate of one second. Both the RPM and fuel values change a lot within one second. In other words, both these signals are under-sampled. Therefore, we use small values for the histogram bin ranges: $[0, 1], [2, 4], [5, 9], [10, \infty)$. Although learning a PDRTA from this data is questionable because the time values are almost always less than 4, our algorithm can still be used to identify PDRTAs that are capable of classifying different types of behavior.

6.4.2 Bounding the amount of splits

In initial experiments, we discovered that the algorithm identified a PDRTA with too many clock guards. In fact, in some states, it identified a new clock guard, with a new transition to another state, for every time value. Upon closer inspection of the target states of these transitions, we noticed that the distributions generated by these states were all different. So, our algorithm was doing as it is supposed to: it splits transitions when the future distribution of the target states of the resulting transitions are different. However, the result was not as we had hoped. Intuitively, it should not matter whether an average-speed event occurs after 10, 11, 12, or 13 seconds. Why these future distributions are different need to be further investigated. Perhaps it is due to the simple discretization method we use. Or it could be that on some stretches of road it always takes exactly 12 seconds to speed-up, while on others it takes exactly 13. If this is the case, the future distribution will probably also be different.

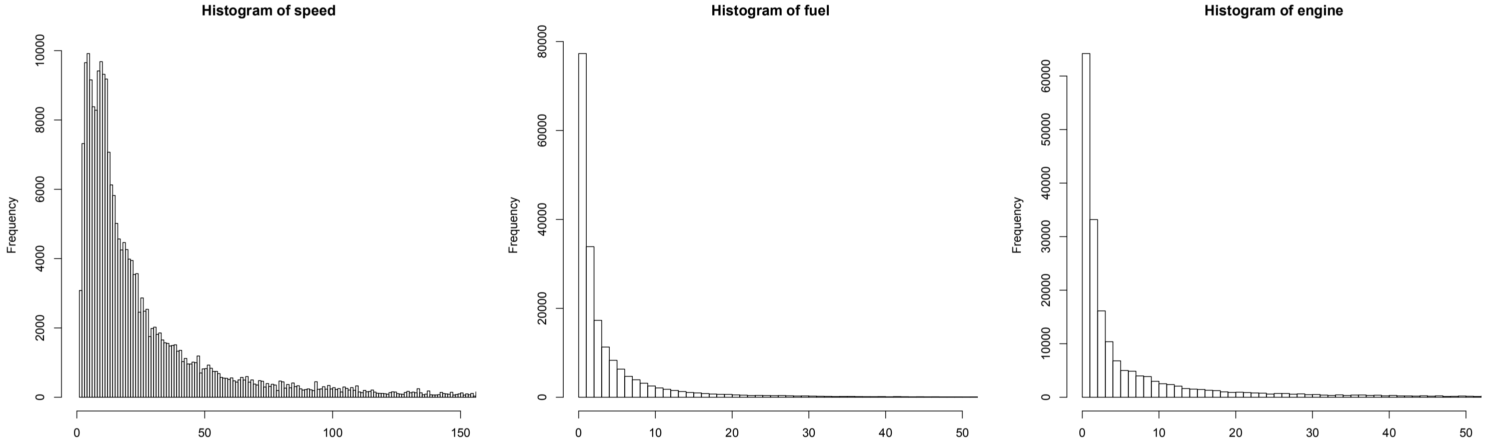


Figure 6.5: Histograms (frequencies) of all the time values of events in the three sensors. There are $10,000 \times 19 = 190,000$ time values in total (19 because the first symbol always has time value 0). Thus, for both the fuel and engine sensors, a fraction of about $\frac{70,000}{190,000} \approx 0.37$ of all events occurred within one second.

In any case, the model we obtain is not suitable for our purposes. We do not want to distinguish between events that occur within 10, or 11, or 12, etc. seconds. We want there to be one, or maybe two bounds that represent deadlines for events. We believe that just two of such deadlines can be used to distinguish good driving behavior from bad driving behavior. For example, in some specific state an average-speed event within 10 seconds could be indicative of pulling up too quickly. After 11 seconds, it could indicate pulling up with a good speed. In addition, we could want another bound at 20 seconds. When the average-speed event exceeds this bound the pull-up behavior is slow.

In conclusion, we want to identify PDRTAs with at most three clock guards (at most two deadlines) per symbol per state. We make our algorithm return only such PDRTAs by simply disallowing it to split transitions when there already exist two other transitions from its source state with the same symbol.

6.4.3 Modifying the initial symbol and state

Because we use arbitrary subsequences as input for our algorithm, the initial symbol of a timed string should be treated differently from the others. More specifically, the time values of initial symbols have little meaning. In timed strings, these values represent the time that has elapsed since the previous symbol. For the initial symbol, there is no previous symbol. Therefore, all of the initial symbols of the subsequences have a time value of 0.

The consequence for our model is that the initial state has a special purpose, namely to distinguish subsequences based on their initial symbol, not on their time values. All the other states of our model distinguish timed strings based on their symbols and time values. Therefore, we disallow any of these states to be merged with the initial state. Usually, this results in their actually being $|\Sigma|$ (the size of the alphabet) different initial states, one for every initial symbol. However, when the future distributions are similar for some of these symbols, then these can be merged into a single state.

6.5 PDRTA classifiers

We can use the transformed data and modified models as described in the previous sections in our algorithm. The statistical test we use is the likelihood ratio test as described in Section 5.3.1. We use this test because it is intuitive and because it did not perform significantly worse than any of the other statistics on artificial data (see Section 5.4.5). The result of our algorithm is a PDRTA for every (in our case three) sensor value. In this section, we explain how we use all of these PDRTAs in order to classify the behavior displayed in new data. We first describe how to construct a classifier from a PDRTA using just a few labeled examples, then we show how we combine the classifications of the different sensors into a single overall classification.

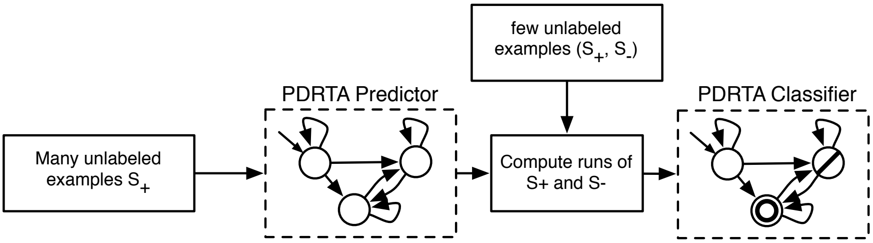

few unlabeled examples (S_+, S_-)
PDRTA Predictor
PDRTA Classifier
Many unlabeled examples S_+
Compute runs of S+ and S-

Figure 6.6: Constructing a classifier based on a PDRTA. We use the runs of a few labeled examples to label (make final) the states of the PDRTA. The result is a classifier.

6.5.1 Classifying using few labeled examples

The result of running our algorithm on an unlabeled data set is a PDRTA. This automaton does not contain any final states and hence can initially not be used to classify new data. However, the PDRTA model does describe the behavior that is displayed in the unlabeled data set. When identified correctly, two timed strings only reach the same states if their possible futures are the same. Intuitively, these two timed strings display the same behavior, and hence their labels should also be the same. Consequently, we can use the label of a single labeled string to label (make final) a state of the PDRTA. This process is depicted in Figure 6.6.

Essentially, this is the method we use to make a classifier out of a PDRTA. There is only a small issue because of the fact that we use arbitrary subsequences as input for our identification algorithm. Therefore, our algorithm identifies the causal states instead of the actual states of the PDRTA (see Section 6.3.2). We label the states of these causal states using a small sample of labeled timed strings in the following way.

Suppose we want to construct a classifier for two different types of behavior b and b', for instance good and bad driving. Let Q_b and $Q_{b'}$ denote the sets of states that are reached by the timed strings that display behavior b and b' respectively. The sets of states Q_b and $Q_{b'}$ can overlap, but they should differ in some states. We use these states to make a classifier from a PDRTA:

- assign the label b to the states in $Q_b \setminus Q_{b'}$, i.e., the states reached by b but not by b';
- assign the label b' to the states in $Q_{b'} \setminus Q_b$, i.e., the states reached by b' but not by b.

When a new timed string reaches a state labeled with b, we conclude that it is displaying behavior b, and not b'. In addition to classifying new timed strings, the labels assigned to states can also be used to analyze the old unlabeled data set, for instance, in order to determine how often good or bad driving occurred.

In this way, our algorithm identifies one classifier for every sensor. These classifiers can assign a label to any new timed string. However, since our identification method is based on the theory from computational mechanics, the identified PDRTAs can be used to determine the current state for observed histories of any length. Thus, at any point in time, our classifiers can give many different classifications. Since we want to give a single classification, we require some method that combines these classifications into a single overall classification. There are many possible ways to do this. For instance returning a classification only if all classifiers agree on the label, or using a majority vote to determine the label. In the next section, we discuss some issues regarding this combination and give our solution to this problem.

6.5.2 Combining causal classifications

The fact that we model the causal states of the process makes it difficult to determine the precise classification label. The problem is that it is unclear what part of a new timed string to use in order to determine the classification. We could use the whole timed string, i.e., simply compute the complete run and determine the label at every index. But this only works if the identified model is completely correct. When we make some small identification error, then the run of the new timed string can diverge from its actual run and from that point on many classifications will be incorrect. We identified a model using timed strings of length 20 (see Section 6.3.2). Similarly, we could use length 20 subsequences to determine the classification labels. This can be done by computing for every index, the run of the length 20 subsequence up to that index and using the label of the state reached by this subsequence.

Using the subsequences of length 20 is possible, but it makes more sense to use shorter timed strings. For these shorter timed strings we have used future distributions in order to determine the states they reach in our identification algorithm. Unfortunately, it is unclear which length to use exactly. The run of a shorter sequence will be more correct since it is based on more data (larger future distributions) during identification. The run of a longer sequence will be less correct but better at distinguishing the different types of behavior. We decided to use all sequences of length 1 to 10 and to combine them in the same way we combine the classifications of the different sensors.

It is currently not yet clear how to combine the different labels assigned to these 10 timed strings. Therefore, we have not yet implemented any combination method. We simply return all labels, giving by all classifiers, and all lengths from 1 to 10. A human inspector then has to decide the label based on this information.

6.6 Results

We implemented the procedure for obtaining a classifier as discussed in the previous sections and applied it to the Van der Luyt data. From this, we obtained three PDRTA classifiers. In this section, we first discuss the quality of these models and their ability to classify new data in Section 6.6.1.

We implemented a simple detection mechanism for the identified classifiers and tested it first in a real-time test in Section 6.6.3. Unfortunately, it turned out to be difficult to establish the quality of our classifiers from this test. Therefore,

	number of transitions	AIC	AIC single state model
speed	2,305	871,550	1,472,840
fuel	960	842,536	1,102,270
engine	1,009	697,190	1,076,750

Figure 6.7: The sizes and AIC scores of the PDRTAs identified by our algorithm for every sensor.

we also tested it on historical data in Section 6.6.4. In this test, it turned out to be surprisingly accurate. This result is a proof of concept for the use of our techniques.

6.6.1 Identifying PDRTA models

We applied the RTI+ algorithm to the discretized van der Luyt data (see Section 6.2). After some initial tests of our algorithm on this data, we modified the PDRTA model as described in Section 6.4. We then tried to identify one such model for each of the three sensors. The performance of each of these models is measured using their sizes and their AIC values (see Section 5.4.3). These AIC values were computed without a test set. We simply tuned the parameters of both the identified and the single state models using the data set and then used this to compute the AIC. In addition, we used the same histogram bins as described in Section 6.4.1 for the computing the AIC. The sizes and AIC scores of each of these models and a trivial single state model are displayed in Figure 6.7.

From the AIC results, we can conclude that the models perform a lot better than the trivial single state model. Unfortunately, the AIC scores by themselves do not say much about the quality of the identified PDRTA models. However, the sizes of the identified models allow us to draw some conclusions regarding this quality.

The identified models are quite large. From the results of the previous two chapters (Section 4.5 and Section 5.4.5), we know that our algorithm is capable of correctly inferring a PDRTA with about 8 states, 8 splits, and an alphabet of size 4, from a data set of size 2,000. The number of transitions of such a PDRTA is 40. From the Van der Luyt data, we obtained PDRTAs that are much bigger. Hence, it is very unlikely that the identified PDRTAs are completely correct. Of course, we did identify these PDRTAs using 10,000 instead of 2,000 examples. Thus, we should be able to identify larger PDRTAs, but this increase in data does not explain the large size of the identified models.

However, the first few states of the PDRTAs (in distance from the start state) are identified using a lot of data. Because of cycles in the transitions of the PDRTAs, most of these states are reached by around 5,000 timed strings. Since 2,000 timed strings is sufficient to correctly identify the start state of a size 40 PDRTA, these first few states are probably identified correctly.

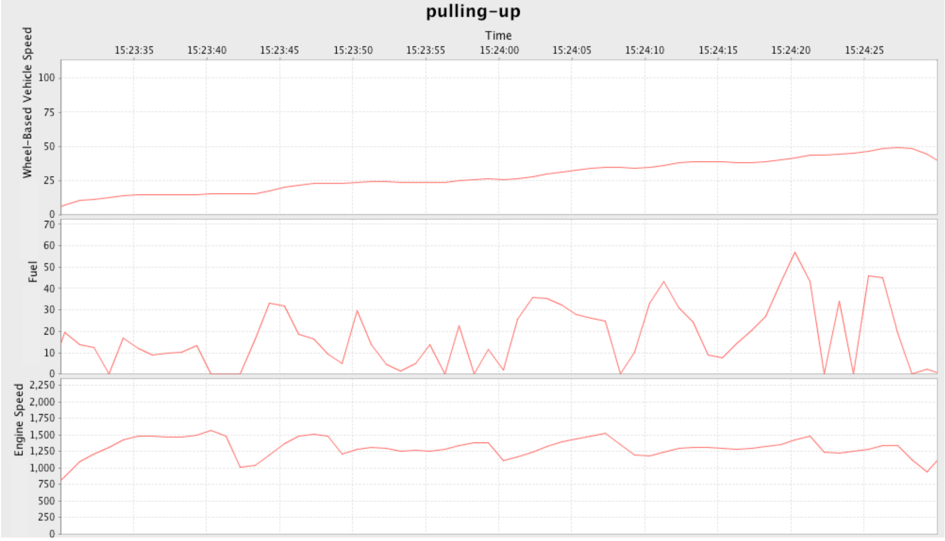


Figure 6.8: A typical example of the sensor values when a truck pulls-up (normally) from 0 to 50 km/h.

6.6.2 Labeling states for classification

Although one can question the quality of the identified PDRTAs due to their sizes, it is still possible to use them as classifiers. The idea is that, although they can sometimes be wrong, the PDRTA models do combine the paths of timed strings only if their possible futures follow the same probability distribution. The future distributions of different classes of behavior are also different. Hence, we should be able to use the identified PDRTAs as classifiers.

Collecting a few examples

For the Van der Luyt case, we tried to classify a type of behavior that results in unnecessary fuel usage: pulling up too quickly from a traffic light, see Figure 6.8. Pulling up too fast requires a lot more fuel than slowly letting the truck gain speed, i.e., pulling up normally. In order to classify this behavior, we required a few labeled examples. These examples can then be used to create a classifier out of the identified PDRTAs as explained in Section 6.5. The labeled examples were obtained by driving a short lap around Oegstgeest, with many traffic light stops. After these stops, the truck driver pulled up too fast about half of the time, and the other half he pulled up normally. We labeled the timed strings that occurred while pulling up. The results for the speed sensor are shown in Figure 6.9.

All but one of the timed strings in Figure 6.9 start with an a symbol. This means that the truck came to a full stop (or at least a speed less than 5 km/hour). In one timed string the initial symbol is a c symbol. In this case, the driver actually did not slow down completely, but pulled up too fast from driving around

pulling up too fast	pulling up normally
$(a,42)(b,80)(c,12)(d,10)(c,7)$	$(a,10)(b,14)(c,21)(b,42)$
$(a,13)(b,65)(c,6)(d,8)$	$(a,11)(b,34)(c,13)(b,18)$
$(a,6)(b,39)(c,10)(d,10)$	$(a,8)(b,10)(c,14)(b,45)$
$(a,7)(b,15)(c,8)(d,7)$	$(a,25)(b,5)(c,18)(b,27)$
$(a,7)(b,2)(c,7)(d,7)$	$(a,11)(b,6)(c,20)(b,14)$
$(c,22)(b,4)(c,12)(d,7)(c,8)$	$(a,12)(b,18)(c,15)(b,20)$
$(a,7)(b,11)(c,6)(d,7)(c,16)$	$(a,8)(b,35)(c,24)(b,20)$
$(a,8)(b,29)(c,7)(d,8)(c,8)$	$(a,12)(b,32)(c,18)(d,13)$
$(a,11)(b,28)(c,7)(b,12)$	$(a,7)(b,62)(c,21)(b,28)$
$(a,10)(b,320)(c,12)(d,7)(c,34)$	$(a,12)(b,8)(c,17)$
$(a,10)(b,8)(c,8)(d,8)(c,31)$	$(a,10)(b,27)(c,19)(d,16)$
$(a,8)(b,24)(c,9)(d,8)(c,12)$	

Figure 6.9: The events obtained from the speed sensor during a test-lap in which the driver pulled-up too quickly (left) and normally (right).

20 km/hour (the next event is a b event). In every timed string, the second symbol is a b symbol, which indicates a speed-up of the truck. The time until this symbol occurs is the number of seconds the truck stood still at the traffic light. The third symbol is a c event, indicating a further speed-up. It should be possible to use the time value of this event in order to classify the two different pull-up behaviors: a small time value indicates a fast pull-up, a large value indicates a slow pull-up. This can be clearly seen in the examples. The next event of the examples is either a b or a d event, depending on whether the speed of the truck exceeds 50 km/hour or not. A small number combined with a d event indicates that the driver is pulling-up to quickly. A b event indicates that the driver is slowing down again, for instance in order to stop for the next traffic light. For some examples, there is a fifth event. This usually is a slow-down event after driving fast. We included these events if it occurred within the time span we used for labeling the pull-up behaviors.

We used the same time span in order to obtain examples for the fuel and engine sensors. However, because a lot more events occur within this time span in these sensors (around 15), we only show the first few (7) of these events, see Figures 6.10 and 6.11. The timed strings from the fuel and engine sensors are less intuitive than those from the speed sensor. This is mostly due to the fact that they are under-sampled (see Section 6.4.1), but also partially due to the fact that understanding them requires some knowledge of engines and the values that these sensors record. At the start of this section, we remarked that pulling up too fast is associated with more fuel usage. In the timed strings, this can be seen for example by comparing the number of d events (a lot of fuel usage) of the too fast and normal puling-up behaviors. From Figure 6.10, it seems clear that more d events occur in the pulling-up too fast case (19 against 5).

Interestingly, there are no occurrences of e and f events (very high fuel usage) in the fuel sensor. Probably this is due to the fact that the truck was empty at the time of the test. In the case of pulling-up too fast, a pattern that can be observed in the fuel sensor events is a repetition of consecutive a and d events, with little

pulling up too fast	pulling up normally
$(a,21)(b,89)(c,0)(d,1)(c,2)(a,1)(c,2)$	$(a,1)(b,16)(c,3)(a,3)(b,2)(c,0)(b,5)$
$(a,1)(b,76)(c,0)(d,2)(a,1)(d,0)(a,1)$	$(a,1)(b,38)(c,2)(d,1)(c,1)(a,1)(b,2)$
$(a,1)(d,40)(c,3)(a,0)(c,1)(d,1)(a,1)$	$(a,0)(b,10)(c,3)(a,3)(b,2)(c,0)(d,1)$
$(a,0)(b,17)(d,2)(a,1)(d,1)(a,0)(b,1)$	$(a,22)(b,9)(c,2)(a,4)(b,1)(c,1)(d,0)$
$(a,0)(b,5)(c,1)(d,1)(a,1)(d,0)(a,1)$	$(a,1)(b,11)(c,3)(a,3)(b,1)(c,1)(b,3)$
$(a,1)(b,3)(a,1)(c,4)(a,1)(c,1)(b,0)$	$(a,1)(b,22)(d,2)(c,2)(a,0)(b,2)(c,1)$
$(a,1)(b,15)(d,1)(a,1)(d,1)(a,1)(d,1)$	$(a,1)(b,40)(c,2)(b,3)(a,0)(b,1)(c,1)$
$(a,1)(b,32)(c,1)(d,1)(a,1)(d,0)(a,1)$	$(a,1)(b,33)(c,3)(b,3)(a,0)(b,2)(c,0)$
$(a,0)(b,30)(d,3)(a,1)(d,1)(a,0)(b,1)$	$(b,66)(c,4)(b,3)(a,1)(b,1)(c,1)(b,4)$
$(a,1)(b,324)(a,0)(b,3)(c,3)(b,1)(c,2)$	$(a,1)(b,13)(c,3)(a,3)(b,2)(c,0)(b,2)$
$(a,1)(b,11)(d,1)(a,1)(c,1)(a,1)(c,1)$	$(a,1)(b,28)(c,4)(a,3)(b,1)(c,1)(d,0)$
$(a,1)(b,11)(d,20)(a,2)(d,1)(c,0)(a,1)$	

Figure 6.10: The first 7 events obtained from the fuel sensor during a test-lap in which the driver pulled-up too quickly (left) and normally (right).

pulling up too fast	pulling up normally
$(a,19)(b,87)(d,0)(b,3)(a,1)(c,2)(d,1)$	$(a,3)(b,25)(c,1)(d,1)(c,0)(a,1)(b,2)$
$(a,2)(c,76)(d,1)(c,1)(a,0)(b,1)(c,1)$	$(a,2)(b,46)(c,0)(d,1)(c,1)(a,1)(b,2)$
$(a,1)(b,43)(d,2)(b,2)(d,1)(b,3)(c,2)$	$(a,2)(b,17)(d,1)(b,1)(a,1)(b,3)(a,2)$
$(a,2)(c,23)(d,0)(b,2)(c,2)(d,0)(c,2)$	$(a,2)(b,11)(d,1)(b,1)(a,1)(b,2)(c,2)$
$(a,2)(b,11)(c,0)(d,1)(b,1)(d,2)(c,2)$	$(a,2)(b,35)(d,1)(b,1)(c,4)(a,2)(b,3)$
$(a,3)(b,2)(a,2)(b,2)(a,1)(c,5)(d,1)$	$(a,3)(b,29)(d,1)(b,2)(a,1)(b,2)(a,4)$
$(a,1)(b,17)(d,0)(b,2)(c,2)(d,0)(a,2)$	$(a,2)(b,42)(c,1)(d,0)(b,1)(a,1)(b,2)$
$(a,3)(c,38)(d,1)(b,1)(c,2)(d,1)(a,2)$	$(a,3)(b,43)(d,1)(c,1)(b,1)(a,0)(b,2)$
$(a,3)(b,38)(d,1)(a,2)(b,0)(c,1)(d,1)$	$(a,1)(b,70)(c,1)(d,1)(c,1)(b,0)(a,1)$
$(a,2)(b,329)(c,2)(d,0)(b,2)(c,2)(d,0)$	$(a,1)(b,22)(c,1)(d,1)(b,1)(d,3)(b,1)$
$(a,2)(c,17)(d,0)(c,3)(d,2)(a,3)(b,1)$	$(a,12)(b,40)(d,1)(b,1)(a,1)(b,2)(c,1)$
$(a,1)(b,30)(d,2)(b,2)(d,2)(c,2)(d,3)$	

Figure 6.11: The first 7 events obtained from the engine sensor during a test-lap in which the driver pulled-up too quickly (left) and normally (right).

time between them. Such a sequence means that the truck first uses little fuel, then a lot of fuel, then little again, etc, all within a few seconds. This has to do with the shifting of gears. When a truck shifts gear, it uses little fuel, subsequently starting again in a higher gear requires a lot of fuel. This pattern is different if the truck pulls up normally. In this case, the truck also shifts its gear, but not with the same rapid succession. In these timed strings one thus sees more b or c events in between.

It is more difficult to draw such conclusions from the engine sensor. There should be different patterns for the different pull-up behaviors. But from the labeled timed strings it is not very clear what these patterns are. We only observe that more d events occur when the truck is pulling up too fast.

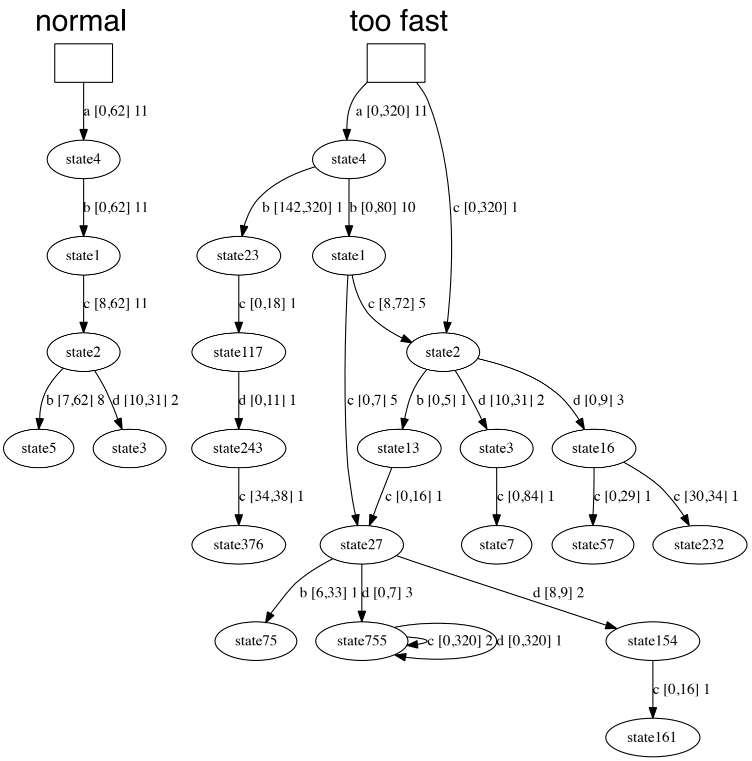


Figure 6.12: Parts of the identified PDRTA for the speed sensor that are reached by the timed strings from Figure 6.9. Left is the part reached by the normal pull-up behavior, right the too-fast pull-up behavior. States are labeled with their number. Transitions are labeled with their label, delay guard, and the number of examples strings that fire the transition.

Constructing PDRTA classifiers

Having collected these labeled examples, all we need to do is to run the identified PDRTAs on these examples, and discover which states are reached when pulling-up too quickly, and when pulling up normally. Figure 6.12 shows these states, and the traversed transitions, for the speed sensor examples. It seems to be a coincidence that the part of the identified PDRTA that is traversed by the too fast examples is larger than the one traversed by the normal examples. This does not occur in the other sensor PDRTAs.

We used the PDRTA parts of Figure 6.12 to identify some states that are only reached when the driver pulls up too quickly and some when he pulls up normally. In Figure 6.12, state 5 is reached by 8 of the 11 normal examples, and none of the 12 fast examples. An example of a state that is only reached by the fast examples

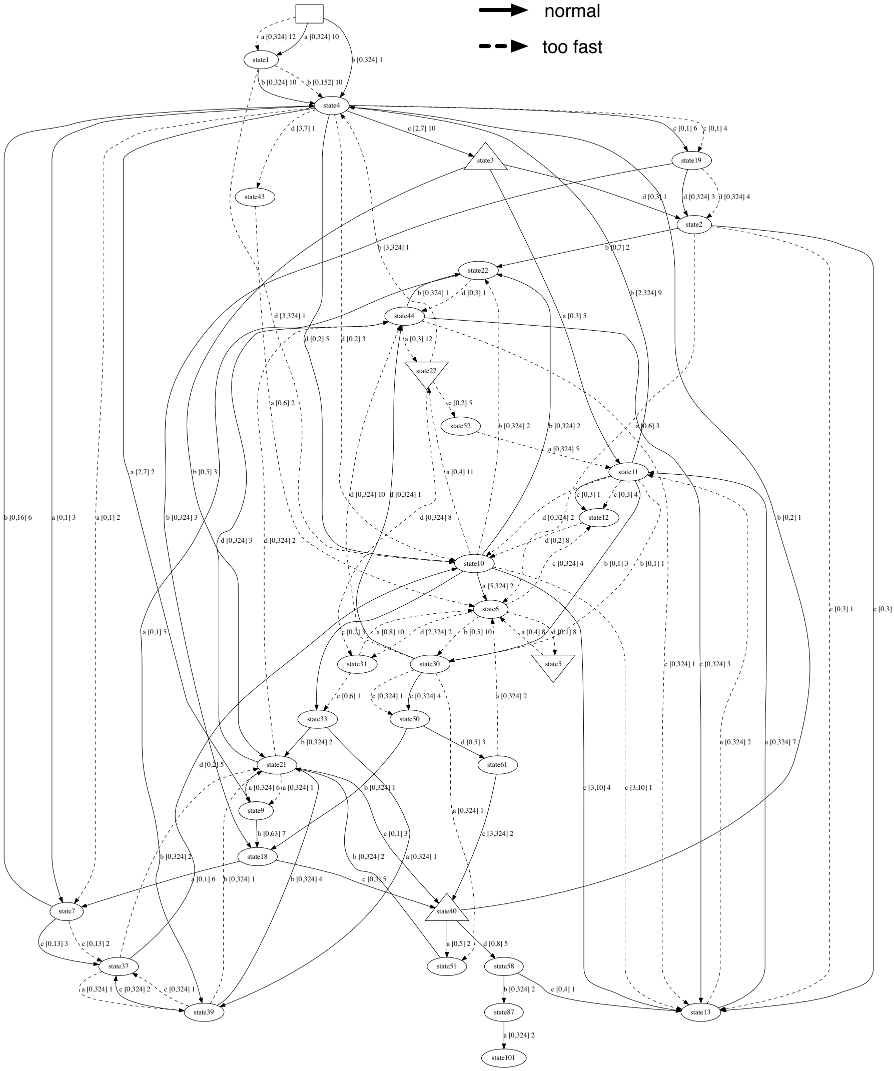


Figure 6.13: Parts of the identified PDRTA for the fuel sensor that are reached by the timed strings from Figure 6.10. States with a △ shape are labeled normal, states with a ▽ shape are labeled too fast.

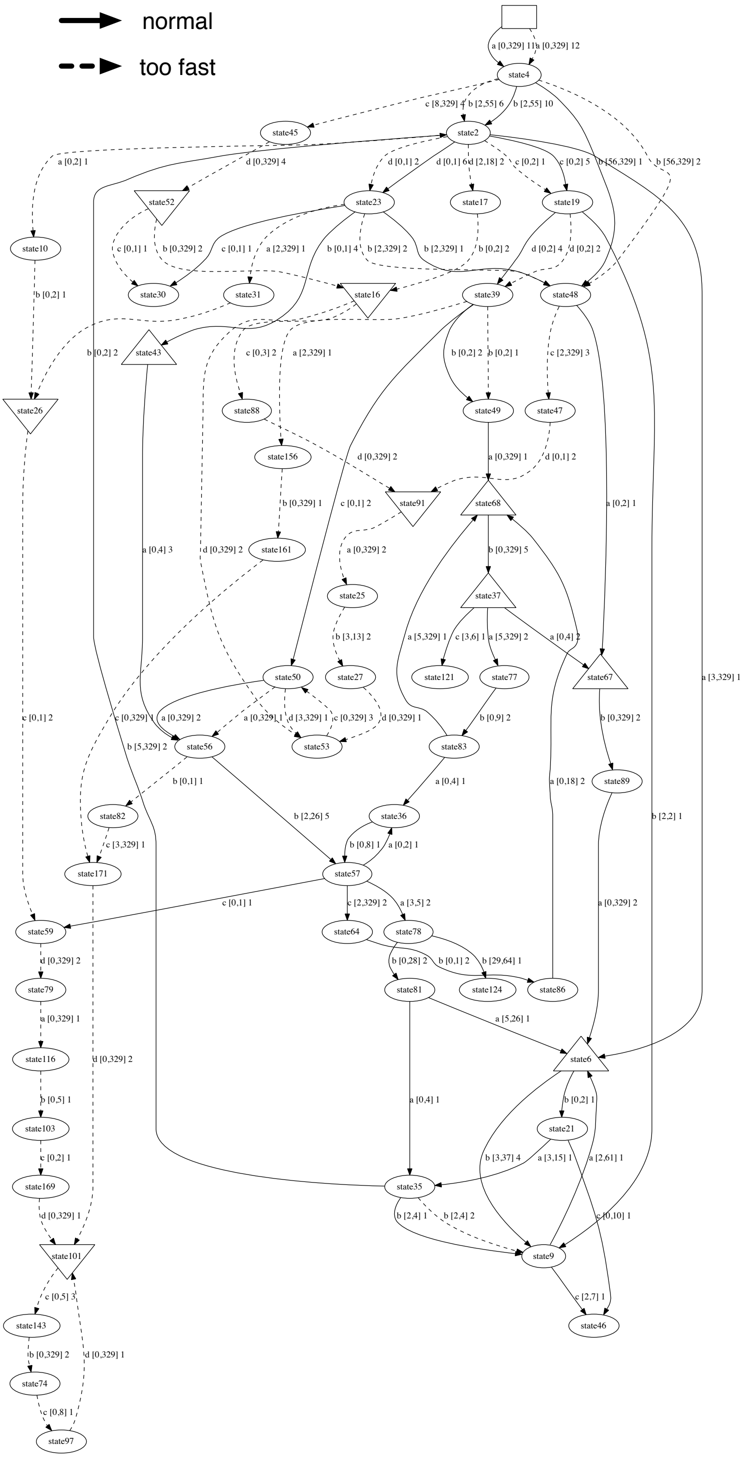


Figure 6.14: Parts of the identified PDRTA for the engine sensor that are reached by the timed strings from Figure 6.11. States with a △ shape are labeled normal, states with a ▽ shape are labeled too fast.

is state 27; 6 of the fast examples reach this state. Note that the transition to this state has c as symbol and $[0, 7]$ as clock guard. Thus, it is only reached by timed strings that contain a c event within 7 seconds after the b event of the pull-up behavior. A c event with more than 7 seconds goes to state 2, which is also the state that is reached by all the normal pull-up examples. Thus, the identified PDRTA model for the speed sensor does distinguish between the two behaviors of pulling up too fast and pulling up normally, exactly in the way we expected, based on the timed strings of Figure 6.9.

This shows that our PDRTA identification algorithm can be used to distinguish different types of behavior, even if we beforehand do not know what behavior to distinguish (the PDRTAs are identified from unlabeled data). In this way, our algorithm can be used to give more insight into the different kinds of behavior of a system/process.

For both the normal PDRTA and the too-fast PDRTA, we identified and labeled all states that were reached by at least 4 example strings in one case, and in none the other. Thus, state 27 and 16 are labeled in the too-fast speed PDRTA, and state 5 is labeled in the normal speed PDRTA. We did the same for the fuel sensor and engine sensor PDRTAs. The result of this labeling is depicted in Figures 6.13 and 6.14. These figures only show the parts of the PDRTAs that are reached by at least 2 example strings. Although we labeled only a few states in the large PDRTAs for each of these sensors, these labels result in three classifiers that can be used to detect whether a driver is pulling up too fast or not.

6.6.3 A real-time test

We first implemented a simple real-time detection mechanism for our classifiers. This mechanism was tested by checking whether it was capable of correctly classifying the pull-up behavior of a driver in real-time. The detection mechanism reads the sensor data directly from the USB port. This port was connected to a central bus of the truck, known as a CAN-bus using technology from Squarell Technology. This enabled us to detect the behavior of a truck driver in real-time while he was actually driving a truck.

We performed this test using a different driver, a different truck, and under different weather conditions. In spite of these differences, our models were often able to detect when the driver was pulling up too fast. Unfortunately, it is difficult to precisely determine the quality of our classifier in this setting. Partially, this is due to the way we combine the information from the three classifiers. A computer printed the labels of reached states in all three of the models in real-time. We had to decide ourselves how many labels we needed before making a classification. For instance, it sometimes occurred that one normal and many too-fast labels were printed. In such a case we concluded that the driver was pulling-up too fast and indicated this to the driver. Figure 6.15 shows a typical example of the output during the first 10 seconds of a too-fast pull-up. Another part is due to the problem that we cannot simulate the driving behavior in a clean environment. We have to test it on the road. During the test drive we came across some obstacles (for instance vehicles blocking the road, or changes in weather) that influence the quality of the test itself.

```
speed = 3.625
read fuel: D, 2
44 10 10 10 10 10 10 10 2
speed = 9.375
read speed: B, 10
72 15 15 15 15 134 15 1 1
read engine: C, 16
108 108 108 108 108 108 108 45 1
speed = 13.875
read fuel: A, 2
27 too fast 27 too fast 27 too fast 27 too fast 27 too fast 27...
read engine: D, 1
122 122 122 122 122 122 52 too fast 3 3
speed = 15
read engine: C, 1
128 128 128 128 128 30 7 7 1
speed = 14.9375
read engine: A, 1
137 137 137 137 31 25 25 25 4
speed = 15.625
read fuel: D, 3
31 31 31 31 31 31 31 10 2
speed = 18.125
read engine: B, 2
138 138 138 26 too fast 26 too fast 26 too fast 26 too fast 2 2
speed = 22.0625
read fuel: A, 2
6 6 6 6 6 6 27 too fast 6 1
read engine: D, 1
80 80 40 40 40 40 23 23 3
speed = 25.5
read fuel: C, 1
12 12 12 12 12 52 12 12 3 normal
speed = 26.5
read fuel: A, 1
28 28 28 28 11 28 28 11 1
read engine: B, 2
130 130 130 130 130 48 48 16 too fast 2
```

Figure 6.15: The first 10 seconds of output of a typical example of a too-fast pull-up. Some lines show the current speed of the truck. Others show the discretized symbols for the three different sensors. Whenever a symbol is read for a sensor, the line directly under it shows a list of the state numbers the timed strings of length 1 to 10 end in the classifier for that sensor. If such a state is labeled, this label is printed directly afterwards. Interestingly, the different length timed strings often end in the same states.

Unfortunately, we did not record a significant decrease in fuel usage during the test. The main reason is that we notified the driver when it in fact was already too late. It takes our models at least 3 seconds before drawing any conclusion about the pull-up behavior. We believe this to be fast, because it then only requires 2 or 3 events to draw a conclusion. Usually, such a conclusion was drawn using the fuel sensor. The engine sensor sometimes also indicated when the driver was pulling up too quickly, but only after 5 seconds.

6.6.4 Tests on historical data

Because the real-time test does not provide us with sufficient information to make an informed decision regarding the quality of our classifiers, we also tested them on the historical data that we have available. This is the same data that we initially identified the PDRTAs from, see Section 6.3. From this data, we extracted all pull-up instances and used this data to first determine the correlation of our classifiers with the truck speedup. The truck speedup is the speed increase between two consecutive measurements of the speed sensor.

In our tests, the correlation between these speedups and our classifiers turns out to be high only for the fuel sensor. Consequently, when determining the quality of the classifiers, we focus on the fuel classifier. Determining the quality values for this classifier is still difficult however, since we do not exactly know which of the extracted pull-up behaviors are too fast, and which are not. Instead of determining the exact values, we therefor use a simple method that is currently used by Squarell Technology to determine a driver's profile to approximate these values: count the number of speedups that are normal (less than $0.8{}^{m}\!/\!{}_{s^2}$) and those that are too fast (greater than $1.2{}^{m}\!/\!{}_{s^2}$). These two numbers give a good overview of a driver's profile. We show that the labels given by the fuel classifier are highly correlated with these two numbers. We use this correlation to determine a simple detection rule for too-fast pull-up behavior.

In the following, we first describe the data we used to test the quality of our classifiers, then we give our correlation results, and finally we give the quality results of the fuel classifier.

Data from truck pull-ups

We extracted pull-up instances from all the data we have available, i.e., the data described in Section 6.3. We extracted these instances by copying pieces consisting of 30 consecutive measurements (seconds) that satisfy the following rules:

- The initial speed measurement is less than $5{}^{km}\!/\!{}_{h}$.
- The second speed measurement is greater than $5{}^{km}\!/\!{}_{h}$.
- During the 30 measurements, the speedup does not drop below $-0.1{}^{m}\!/\!{}_{s^2}$.

In total we obtained 1532 pull-up instances using these rules. The resulting data set is summarized in Figure 6.16. From this figure, it is clear that there are many different pull-up behaviors: some reach a speed of $80{}^{km}\!/\!{}_{h}$ (a highway pull-up), while others do not go faster than $20{}^{km}\!/\!{}_{h}$.

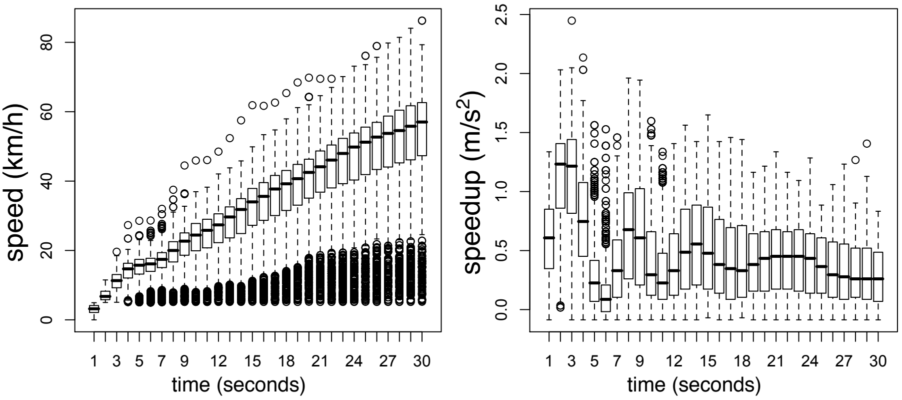


Figure 6.16: A visualization of the 1532 pull-up instances using box plots. The left plot shows the speed of the instances for every measurement. The right plot shows the speedup.

Interestingly, there are two clear drops in the vehicle speedups: one after 6 measurements (seconds), and one after 11 measurements. These are likely the points in time where the truck shifts into a higher gear. A truck has more than three gears, but somehow these two gear shift points are more or less independent of the pull-up behavior.

The largest speedups clearly occur in the first few seconds of a pull-up. This can be problematic for our classifiers since they will often not be able to make good use of these initial measurements. These measurements will often result in just one or two timed symbols, and this is insufficient information to determine whether the pull-up is too fast or not.

Correlation with speedups

The first property of our classifiers we have to determine is their correlation with the truck speedups. Such correlation is important since we intend to use these classifiers to determine the type (normal or too fast) of the truck speedup. In the data, there is a single truck speedup value for every measurement. In order to determine the correlation with the labels given by our classifiers, we require a method that determines a classifier value for every such measurement. We test two straightforward methods to compute such a value:

- one using a *single* run of the classifier PDRTAs;
- one using a *combination* of runs as described in Section 6.5.2.

The first method is possible because a pull-up contains a clear starting point: the first time the truck obtains a speed greater than $5^{km}/_{h}$. We simply start the

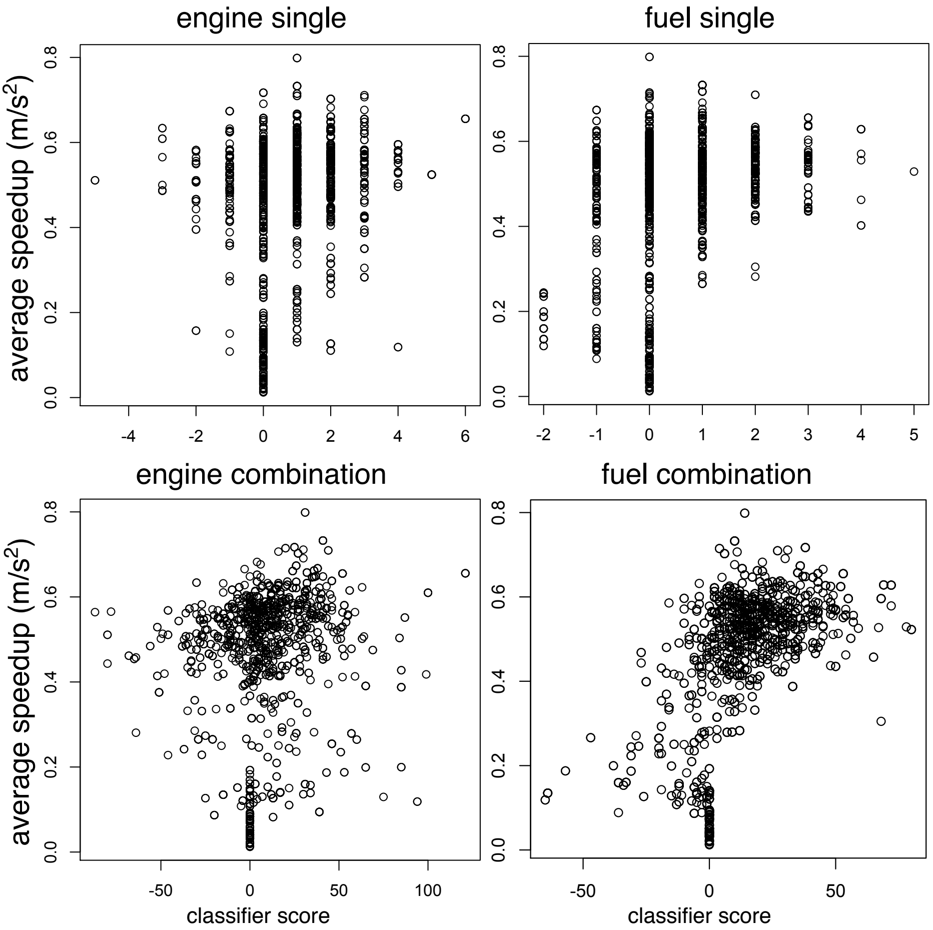


Figure 6.17: Scatterplots of the average speedup against the classifier values of both the single and the combination classifiers of the engine and the fuel sensors.

run of our PDRTAs with the first symbol that occurs after this point. The value of the classifier is then computed as the number of too-fast labels it reached minus the number of normal labels it reached so far:

$$\mathsf{value}(\tau) = \begin{cases} 1 & \text{if } \mathsf{label}(\tau_i) = \textit{too fast} \\ -1 & \text{if } \mathsf{label}(\tau_i) = \textit{normal} \\ 0 & \text{otherwise} \end{cases}$$

$$\textit{single}_i = \textit{single}_{i-1} + \mathsf{value}(\tau_i)$$

where $\mathsf{label}(\tau_i)$ is a function that returns the label of the state that τ_i ends in, τ_i is the timed string that is constructed by the discretization of the sensor value from the first until the ith pull-up measurement, and $\textit{single}_0 = 0$. The second method can be seen as a sliding window technique. This method combines the classification values of all strings of length 1 to 10 seen so far:

$$\textit{combination}_i = \textit{combination}_{i-1} + \sum_{0 \leq j \leq 10} \mathsf{value}(\tau_{i-j,i})$$

where $\tau_{i-j,i}$ is the prefix of τ_i starting at index $i - j$, if $i - j < 0$ then $\tau_{i-j,i} = \epsilon$, and $\textit{combination}_0 = 0$.

Figures 6.17 shows scatterplots of the average speedup value over a complete speedup (30 seconds) and the resulting classifier values for the single and combination values for both the fuel and the engine classifiers. From this figure we make some interesting observations. First of all, both of the fuel classifier values seem to be correlated with the truck speedup: the higher fuel classifier values are associated with larger average speedups. In addition, the figure shows that both of the engine classifier values are not very correlated with the truck speedups: higher values are associated with higher speedups, but only very slightly. From this we can conclude that the engine classifier cannot be used as a classifier for pull-up behavior (or that it is not sufficiently trained).

Quality of classification

We would like to determine the accuracy of the fuel classifier using the speedups data. In order to do so, we compare it with the method that is currently used by Squarell Technology to determine a driver's profile. More specifically, for each speedup, we simply count the number of speedups that are normal (less than $0.8 {}^{m}/_{s^2}$) and those that are too fast (greater than $1.2 {}^{m}/_{s^2}$) and correlate these counts with the fuel classifier values at the end of the speedup. The resulting correlations are plotted in Figure 6.18.

The plots in Figure 6.18 clearly show that the fuel combination classifier can be used to determine whether a driver is pulling-up too fast or in a normal way. The fuel single classifier does clearly not correlate as well as the fuel combination classifier. This is partially due to the fact that about half of the single runs still have a classification value of 0 at the end of a speedup. Most of these runs simply do not reach any labeled states. For the combination classifier, only a little over 10% of all combined runs obtain a value of 0 at the end of a speedup. Another reason might be that the PDRTA models are identified using timed strings that

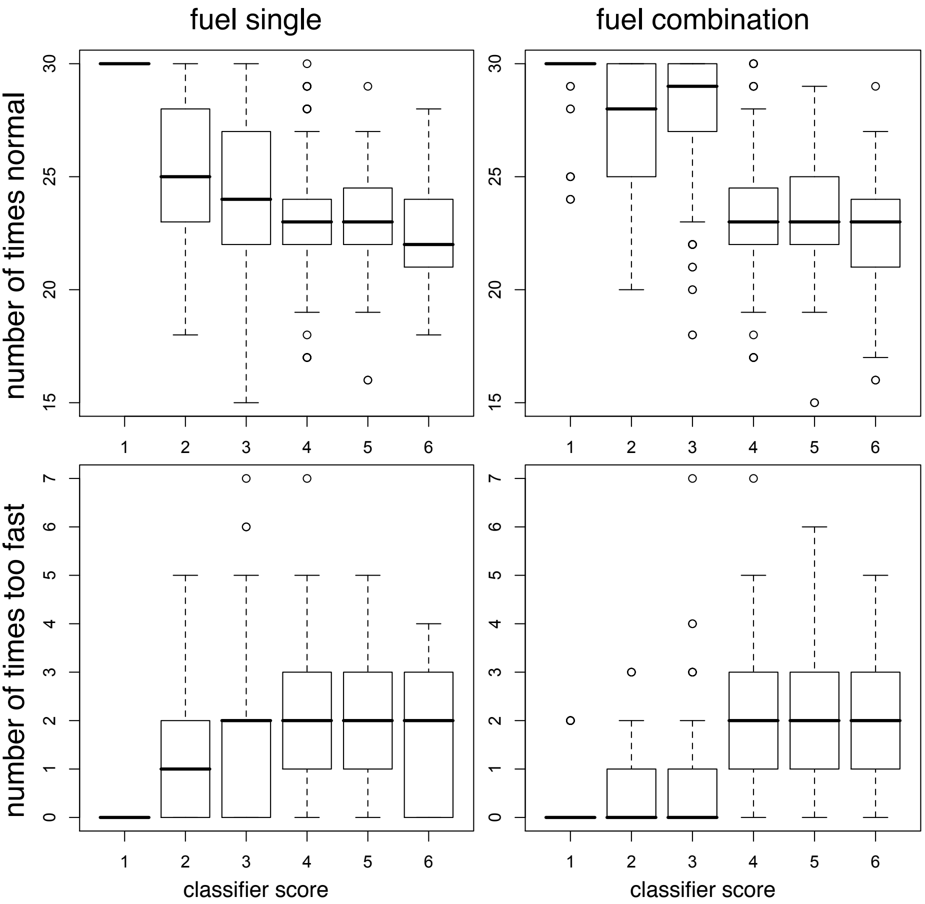


Figure 6.18: Boxplots of the number of times the truck pulled up normally (less than $0.8 m/s^2$) and too fast (greater than $1.2 m/s^2$) against the classifier values of both the single and the combination classifiers of the the fuel sensor. The classifier box values for the single classifier are: $(-\infty, -2] \rightarrow 1$, $[-1, -1] \rightarrow 2$, $[0, 0] \rightarrow 3$, $[1, 1] \rightarrow 4$, $[2, 2] \rightarrow 5$, $[3, \infty) \rightarrow 6$. For the combined classifier, they are as follows: $(-\infty, -20] \rightarrow 1$, $[-19, -10] \rightarrow 2$, $[-9, 0] \rightarrow 3$, $[1, 10] \rightarrow 4$, $[11, 20] \rightarrow 5$, $[21, \infty) \rightarrow 6$.

could start at any index, i.e., based on computational mechanics (see Section 2.4.4). Consequently, the combination classifier uses the data in a way that seems more suited to these models.

The combined classifier plots of Figure 6.18 suggest a simple rule for classification of too fast speedup:

- if the value is greater or equal to 1, then label the pull-up as too fast;
- label it as normal otherwise.

We would like to discover the classification quality of this simple rule. Unfortunately, we still do not exactly know which of the extracted pull-up behaviors are too fast, and which are not. Based on the plots of Figure 6.18 we determine this using the following rule:

- if the number of speedups that are normal (less than $0.8^{m}/_{s^2}$) is less or equal to 25, and the number of those that are too fast (greater than $1.2^{m}/_{s^2}$) is greater or equal to 1, then the pull-up behavior is too fast;
- it is normal otherwise.

Of course, since this rule is based on Figure 6.18, it is a little biased towards our fuel classifier.[2] The rule makes sense however, and we do get surprisingly high quality values (see, e.g., (Bishop 2006) for a description of these values):

$$\begin{array}{lll} \text{Accuracy:} & \frac{\textit{true positives}+\textit{true negatives}}{\textit{total number of examples}} = \frac{920+289}{1532} & \approx 0.789 \\ \text{Precision:} & \frac{\textit{true positives}}{\textit{true positives}+\textit{false positives}} = \frac{920}{920+262} & \approx 0.887 \\ \text{Recall:} & \frac{\textit{true positives}}{\textit{true positives}+\textit{false negatives}} = \frac{920}{920+61} & \approx 0.938 \end{array}$$

These values are very high considering we only used 23 example strings to label only 4 states of the fuel PDRTA. Most importantly, however, they clearly serve as a proof of concept of our techniques.

In addition, about 60% of all truck pull-up instances are too fast under this rule. This clearly shows the need for techniques that are able to detect these pull-ups in real-time in order to give immediate feedback to the truck driver.

6.7 Discussion

We applied the techniques we developed in the previous chapters to the problem of detecting driver behavior. Although our identification algorithm resulted in large PDRTA models, these PDRTAs were able to classify the driving pattern of pulling-up too fast or normally. Hence, they clearly do represent different types of driving behavior, which shows that our techniques can work in this setting.

A nice property of the way we constructed the PDRTA classifiers is that it requires very little expert knowledge, and still is able to distinguish interesting driving behavior. We only used 11 examples of normal pull-ups, 12 examples of

[2] The rule is however not entirely biased towards our classifier because we can easily device other rules that achieve even better scores. For instance only using the number of normal pull-ups results in an accuracy of 0.867.

too fast pull-ups, and a lot of unlabeled data in order to construct these classifiers. This method is therefore widely applicable since it is often very easy to obtain unlabeled data. All one needs is the ability to observe a process.

We used the identified PDRTAs to classify the pull-up behavior. We connected a laptop to the truck data bus and tried to detect the current behavior of the driver during a test drive in real-time. During this drive, we notified the driver in the cases that he was pulling-up too quickly. The results of this test are promising: we were able to detect pulling-up too fast and normal pull-up behavior using our classifiers within a few seconds.

Because this test is too preliminary to draw precise conclusions regarding the quality of the classifier, we also tested our classifiers on historical data. This test shows that especially the fuel sensor classifier achieves sufficient quality in order to be used in practice. The quality values are surprisingly high considering that we only used 23 example strings to label only 4 states of the fuel PDRTA. That our techniques are capable of achieving high quality values from such few data clearly serves as a proof of concept of our techniques.

CHAPTER 7

Conclusions

7.1 Overview

We started this thesis by giving an example application for our techniques, which is to construct a system that detects truck driving behavior in real-time. All of the techniques in this thesis have been developed with this application in mind, until we were finally able to test these techniques on this application in Chapter 6, where we used our techniques to automatically identify such a system from data. This identified system consists of deterministic timed automata (DTA) models.

We chose to model this system using DTAs because they form an intuitive description of the complex truck driving process. Hence, they can be visualized and inspected in order to provide *insight* into the different behaviors of this process. In addition, they can be used to perform different *calculations*, such as making predictions and model checking.

From the available data, we could also have opted to identify non-timed automata, such as deterministic finite state automata (DFAs). DTAs model time *explicitly* using time-recording object called *clocks*. These clocks record time in using numbers, i.e., in binary. In contract, DFAs can record time using additional states, i.e., in unary. Thus, DTAs are more *succinct* and thus more insightful than equivalent DFA models. As a consequence, also the time, space, and data required to identify DTAs can be exponentially smaller than the time, space, and data required to identify equivalent DFAs. In other words, DTAs can be identified more *efficiently* than equivalent DFAs.

Our approach to solve the problem of DTA identification is described succinctly by the following main contributions:

Chapter 3 We investigated which classes of DTAs can and cannot be identified efficiently from data. We proved that DTAs with two or more clocks (n-DTAs) cannot be identified efficiently, and that DTAs with a single clock (1-DTAs) can be identified efficiently.

Chapter 4 We developed an efficient algorithm RTI for identifying an efficiently identifiable class of DTAs, called deterministic real-time automata (DRTAs). To the best of our knowledge, RTI is the first algorithm that can identify a timed automaton model from a timed input sample. The algorithm requires labeled data as input and is guaranteed to converge to the correct DRTA in the limit. We tested RTI on artificially generated data in order to compare it with existing techniques and to determine its performance characteristics.

Chapter 5 We adapted the RTI algorithm to the setting of unlabeled data. The resulting algorithm is called RTI+ and identifies probabilistic deterministic real-time automata (PDRTAs) instead of DRTAs. Similar to RTI, this algorithm converges efficiently in the limit, in this case to the correct PDRTA. We also investigated variants of RTI+ on artificially generated data.

Chapter 6 We finally applied the RTI+ algorithm to the problem of identifying truck driver behavior. The result is a classifier for a simple type of driving behavior: pulling up too quickly. Using this classifier we were able to achieve an accuracy of around 80%, calculated using the labels determined by a simple but sensible classification rule.

All of these steps were necessary because very little was known about the identification problem for DTAs. For instance, had we not performed our theoretical study, we would not have known the convergence properties of our algorithms. Hence, we would not have had a guarantee that our identification algorithm converges to a correct model, which is essential for many applications.

We believe that the theoretical work in this thesis greatly advances the current state of knowledge about the DTA identification problem. Most notably, we now know which classes of DTAs can be efficiently identified, and how to identify them efficiently. It comes as a surprise that 1-DTAs can be identified efficiently because the standard method of transforming a DTA into an equivalent DFA (the region construction) still results in an exponentially larger DFA when applied to a 1-DTA. In general, our theoretical results tell us that identifying a 1-DTA from timed data is more efficient than identifying an equivalent DFA. Furthermore, the results show that anyone who needs to identify a DTA with two or more clocks should either be satisfied with sometimes requiring an exponential amount of data, or has to find some other method to deal with this problem, for instance by identifying a subclass of DTAs (such as 1-DTAs).

In addition, we showed experimentally that RTI significantly outperforms a straightforward sampling approach that uses existing identification methods. We believe that the main reason for this is that RTI uses a heuristic to determine the *timed properties* of a DRTA. In the sampling approach, these properties are fixed and lead to an exponential blowup. Intuitively, the use of a heuristic to overcome this blowup seems a good idea.

During the development of our techniques, we not only advanced the knowledge of DTA identification, but we also provided theorems, techniques, and ideas that are useful in many other domains. For instance, our efficiency proofs provide general results regarding the expressive power of clocks in DTAs that are useful in domains such as verification. In addition, we developed several new statistical techniques for use in RTI+ that advance the state-of-the-art in probabilistic deterministic finite state automaton (PDFA) identification. Furthermore, for our experiments on unlabeled data, we proposed a new measure for the quality of identified probabilistic models that values the identified model more highly than existing quality measures.

In the application of real-time detection of truck driver behavior, we presented a proof of concept for the use of our techniques. The result of 80% accuracy is remarkable considering we only used 23 example strings to label a handful of states in the PDRTAs. That our techniques are capable of achieving high quality values from such little data clearly serves as a proof of concept of our techniques.

We implemented these classifiers also in a system that detects pulling up to quickly in real-time and onboard of the truck. During test-drives these classifiers proved to be able to detect pulling-up to quickly, but it is difficult to assess their accuracy in these tests.

7.2 Future work

Overall, in trying to solve the problem of real-time classification of driving behavior, we believe we have made a significant number of contributions. However, the quality of our classifiers can still be increased. In addition, there are many interesting new directions for theoretical research and other possible applications that can be investigated. We now first give a few possibilities for future theoretical work, then we discuss some directions for future work that would increase the value of our techniques for potential applications.

7.2.1 Theory

We give three directions for future theoretical work: timed automaton identification theory, timed automaton identification algorithms, and probabilistic model evaluation. In the following, we discuss each of these directions in turn.

Timed automaton identification theory

Although our theoretical results and developed algorithm give a nice overview of the problem of identifying DTAs, there are still many interesting questions that should be answered in future work. A fundamental question is whether a n-DTA identification algorithm can be used to identify 1-DTAs efficiently, while representing them using n-DTAs. A similar result holds for DFAs and non-deterministic finite state automata (NFAs): while NFAs cannot be identified efficiently, an NFA identification algorithm can be used to identify DFAs efficiently (Yokomori 1993).[1] This is possible because NFAs and DFAs are language equivalent. The other way

[1] The proposed NFA identification algorithm is a query learning algorithm. However, any efficient query learning algorithm can be transformed into an algorithm that is efficient in the limit from labeled data (Goldman and Mathias 1996).

around, we could also identify NFAs using a DFA identification algorithm, and then return the smallest NFA that is language equivalent to the identified DFA. Such an approach is not efficient however, because the finding this minimal NFA is a very difficult problem; it is PSPACE-complete (Jiang and Ravikumar 1993). Moreover, in order to converge to an NFA-language, a DFA identification algorithm requires an exponential amount of data. In contrast, the NFA identification algorithm identifies some of these NFA-languages more efficiently.

Whether this also holds for 1-DTAs and n-DTAs is an open problem. It is possible that the amount of data required to converge to a correct n-DTA can be bounded polynomially in the size of the smallest 1-DTA for the same language. The identification of target states and clock guards can certainly be bounded in this way. It is more difficult to come up with timed strings that ensure the identification of the correct clock resets.

Another interesting question is whether 1-DTAs are really the largest class of efficiently identifiable DTAs. To the best of our knowledge, the identification of DERAs in (Grinchtein et al. 2006) is the only other work that deals with the identification of (subclasses of) DTAs. It would be interesting to search for other subclasses of DTAs besides 1-DTAs that can be identified efficiently. A good first step in this search is to look for classes of DTAs for which the proof of Proposition 3.6 does not hold. An example of such DTAs are DTAs with the restriction that clock guards can only compare only the valuations of a single clock to constants. These are more expressive than 1-DTAs (1-DTAs are language equivalent but with exponential blowup), but they are not expressive enough to construct the DTA used in the proof of Proposition 3.6. It would be interesting to investigate whether this class of DTAs is efficiently identifiable.

Timed automaton identification algorithms

We gave algorithms for identifying 1-DTAs, DRTAs, and PDRTAs. In contrast to the last two, the 1-DTA identification algorithm is mainly useful for proving our efficiency results. In practice, it will often fail to identify the correct DRTA because it relies on there being a specific set of timed strings (a characteristic set) in the input data. Since 1-DTAs are more commonly used as models for timed systems than (P)DRTAs, it would be interesting to extend RTI and RTI+ to the class of 1-DTAs. This can be done for example by trying to reset clocks on the transitions to blue states, in addition to trying to perform merges and splits (see Section 4.3). The automaton changes because of this reset and hence we can compute some evidence for this reset. We then only reset a clock if there is sufficient evidence for it. Since timed automata with resets can be very complex, it is unclear what the performance of this algorithm will be.

In a similar way, the algorithm can be further extended to the full class of n-DTAs by trying to reset multiple clocks. We already discussed an approach for this in Section 3.4. Our theoretical results show that in order to identify such an n-DTA, we sometimes require an exponential amount of data. We therefore expect this algorithm to perform worse than the 1-DTA identification algorithm. It would be very interesting to test its performance experimentally.

A completely different approach that can be adopted to identify n-DTAs is based on an observation regarding the power of one-clock and multi-clock DTAs (see Section 3.5). The idea is to identify a n-DTA by representing it using n 1-DTAs and taking their intersection. This can be viewed as a type of ensemble method (see, e.g., (Dietterich 2000)). Perhaps these 1-DTAs can all be learned efficiently, and perhaps some form of teamwork can then be used to give performance guarantees for the n-DTA identification problem. A similar idea (based on the nonclosure under union of sets identifiable from text) was one of the main motivations for team-learning (see Section 2.2.2). Investigating such an approach for the identification of n-DTAs is an interesting direction for future work.

Probabilistic model evaluation

In Chapter 5, we suggested test-set-tuned quality measures to determine the performance of identified probabilistic models. We suggested these measures because in our experiments it turned out that, in terms of data requirements, it is often easier to identify a model than to set its parameters correctly. Therefore, the traditional quality measures tend to over-evaluate smaller models. We believe this to be the main reason why a state-merging algorithm for the identification of probabilistic DFAs (state-merging) is often reported to be outperformed by simple N-gram models, see, e.g., (Kermorvant and Dupont 2002). Hence, it would be interesting to investigate the performance of DFA identification and simple N-grams in terms of our test-set-tuned measures.

Surprisingly, we have not encountered test-set-tuned performance measures anywhere else in the language identification literature. We believe the reason for this to be that in most studies non-deterministic models are identified, such as hidden Markov models (HMMs, see Section 2.3.3). In this case, tuning the parameters is almost as difficult as identifying the model itself. In contrast, tuning the parameters of a deterministic model is a very simple problem. Consequently, a non-deterministic model structure (without tuned parameters) means less because its behavior can change radically by modifying the model parameters. A related question is whether non-determinism also influences the AIC measure. Intuitively, a non-deterministic model is more compact than an equivalent deterministic model, and hence can represent more languages using fewer parameters. Since the AIC only takes the number of parameters into account, this does seem to influence the quality of this measure. As far as we know, nobody has investigated these issues.

7.2.2 Applications

The techniques we develop in this thesis have many possible practical applications (see Section 1.6). We now describe directions for future work that would increase the value of our techniques to improving truck driving behavior detection and to other potential applications.

Identifying behavior

In the real-time tests of Chapter 6, we did not observe a significant reduction in fuel usage. This can be a sign that PDRTA models are not well-suited to the detection of driving behavior. Indeed, we had to modify the PDRTA models a lot during the identification phase in order to obtain sensible models (see Section 6.4). We believe, however, that a large part is due to the fact that the fuel and engine sensors were under-sampled. Furthermore, in order to significantly reduce the fuel usage of drivers, the PDRTA models should be able to notify the driver at the beginning of the pull-up, i.e., within about 1 second. Because our PDRTAs were identified using data that contained one measurement every second, this means that our models need to make this decision after observing a single symbol. This is of course impossible. Therefore, an interesting direction for future work is to use more frequently sampled unlabeled (positive) data in order to identify new PDRTA models. Also, more sophisticated discretization and classifier combination routines are perhaps required. Using this new data, we hopefully obtain sensible PDRTA models that do lead to a reduction in fuel usage.

Insight by visualization

One of the motivations for identifying an automaton model is that such a model is insightful, i.e., it can be interpreted by visualization. Unfortunately, in our application in Chapter 6, visualizing them did not provide us with a lot of insight. Partly, this is due to the large size of the identified models. Another important reason, however, is that we lack a good visualization tool. In our applications, we used the Dotty[2] automatic graph layout program to generate figures of the identified PDRTAs. In these figures it is not possible to see how often which paths were traversed, nor to zoom in on specific parts of the PDRTAs using sensor data, nor to determine the value of the statistical evidence that was used to create states, etc. We believe that such a tool would be very beneficial for the insight that the identified PDRTA models can give. An interesting example of such a tool for simple automaton models can be seen in (Blaas, Botha, Grundy, Jones, Laramee and Post 2009).

Identified model checking

Another motivation for identifying automata is that they can be used to perform model checking. By first identifying a model and then running a model-checker, we can effectively perform this on unknown (black-box) systems. In this way, we can perform testing (see, e.g., (Springintveld et al. 2001)) of software systems without requiring a model for that system. All we need is the data (in terms of input-output or function calls) that is produced by this system, and our algorithms will identify a model for this system. This is a very interesting combination of techniques that has many potential applications in software engineering. For instance, using these two techniques we can determine whether an old legacy software system is dead-lock free.

Because our algorithms and methods are all based on sound principles from learning theory (efficient convergence in the limit), it should be possible to apply them in such settings.

[2] http://www.graphviz.org/

7.3 Final conclusion

This thesis presents novel theoretical and practical work in the new field of timed automata identification. The main goal of this thesis is to develop efficient methods for the identification of timed automata that can also be used in practice. The main results are the construction of these methods and the theoretical results that prove their efficiency. We implemented and applied these methods to the application of identifying a monitoring system for truck driving behavior. Our results and methods are of interest to anyone that wants to identify models (or classifiers) for timed systems. In addition, they are useful in other areas such as non-timed automata identification and model checking.

You have just finished reading this thesis. We hope it was an enjoyable and inspiring experience.

Index

Bibliography

Abe, N. and Warmuth, M. K.: 1990, On the computational complexity of approximating distributions by probabilistic automata, *Machine Learning* **9**(2-3), 205–260.

Alur, R.: 1999, Timed automata, *International Conference on Computer-Aided Verification*, Vol. 1633 of *LNCS*, Springer-Verlag, pp. 8–22.

Alur, R. and Dill, D. L.: 1994, A theory of timed automata, *Theoretical Computer Science* **126**, 183–235.

Alur, R., Fix, L. and Henzinger, T. A.: 1999, Event-clock automata: a determinizable class of timed automata, *Theoretical Computer Science* **211**(1), 253–273.

Alur, R. and Madhusudan, P.: 2004, Decision problems for timed automata: A survey, *Formal Methods for the Design of Real-Time Systems*, Vol. 3185 of *LNCS*, Springer, pp. 1–24.

Angluin, D.: 1987, Learning regular sets from queries and counterexamples, *Information and Computation* **75**, 87–106.

Angluin, D.: 1988, Queries and concept learning, *Machine Learning* **2**, 319–342.

Asarin, E., Caspi, P. and Maler, O.: 2001, Timed regular expressions, *Journal of the Association for Computing Machinery* **49**.

Bishop, C. M.: 2006, *Pattern Recognition and Machine Learning*, Springer.

Blaas, J., Botha, C., Grundy, E., Jones, M., Laramee, R. and Post, F.: 2009, Smooth graphs for visual exploration of higher-order state transitions, *IEEE Transactions on Visualization and Computer Graphics* **15**, 969–976.

Blumer, A., Ehrenfeucht, A., Haussler, D. and Warmuth, M. K.: 1989, Learnability and the Vapnik-Chervonenkis dimension, *Journal of the Association for Computing Machinery* **36**, 929–965.

Brand, M.: 1999, Structure learning in conditional probability models via an entropic prior and parameter extinction, *Neural Computation* **11**(5), 1155–1182.

Bshouty, N. H., Cleve, R., Gavaldà, R., Kannan, S. and Tamon, C.: 1996, Oracles and queries that are sufficient for exact learning, *Journal of Computer and System Sciences* **52**, 421–433.

Bugalho, M. and Oliveira, A. L.: 2005, Inference of regular languages using state merging algorithms with search, *Pattern Recognition* **38**, 1457–1467.

Carrasco, R. and Oncina, J.: 1994, Learning stochastic regular grammars by means of a state merging method, *Proceedings of the 2nd International Colloqium on Grammatical Inference*, Vol. 862 of *LNCS*, Springer-Verlag, pp. 139–150.

Cassandras, C. G. and Lafortune, S.: 2008, *Introduction to Discrete Event Systems*, second edn, Springer.

Castro, J. and Gavaldà, R.: 2008, Towards feasible PAC-learning of probabilistic deterministic finite automata, *Grammatical Inference: Algorithms and Applications*, Vol. 5278 of *LNAI*, Springer, pp. 163–174.

Clark, A. and Thollard, F.: 2004, PAC-learnability of probabilistic deterministic finite state automata, *Journal of Machine Learning Research* pp. 473–497.

Cover, T. M. and Thomas, J. A.: 2006, *Elements of Information Theory*, second edn, Wiley & Sons.

de la Higuera, C.: 1997, Characteristic sets for polynomial grammatical inference, *Machine Learning* **27**(2), 125–138.

de la Higuera, C.: 2005, A bibliographical study of grammatical inference, *Pattern Recognition* **38**(9), 1332–1348.

Dietterich, T. G.: 2000, Ensemble methods in machine learning, *First International Workshop on Multiple Classifier Systems*, Vol. 1857 of *LNCS*, Springer, pp. 1–15.

Dima, C.: 2001, Real-time automata, *Journal of Automata, Languages and Combinatorics* **6**(1), 2–23.

Dupont, P., Denis, F. and Esposito, Y.: 2005, Links between probabilistic automata and hidden Markov models: probability distributions, learning models and induction algorithms., *Pattern Recognition* **38**, 1349–1371.

Fisher, R. A.: 1948, Combining independent tests of significance, *American Statistician* **2**, 30.

Fu, K. S.: 1974, *Syntactic Methods in Pattern Recognition*, Academic Press.

Gold, E. M.: 1967, Language identification in the limit, *Information and Control* **10**(5), 447–474.

Gold, E. M.: 1978, Complexity of automaton identification from given data, *Information and Control* **37**(3), 302–320.

Goldman, S. A.: 1999, Computational learning theory, *Algorithms and Theory of Computation Handbook*, CRC Press.

Goldman, S. A. and Mathias, H. D.: 1996, Teaching a smarter learner, *Journal of Computer and System Sciences* **52**(2), 255–267.

Grinchtein, O., Jonsson, B. and Petterson, P.: 2006, Inference of event-recording automata using timed decision trees, *CONCUR*, Vol. 4137 of *LNCS*, Springer, pp. 435–449.

Grünwald, P.: 2007, *The Minimum Description Length Principle*, MIT Press.

Guédon, Y.: 2003, Estimating hidden semi-Markov chains from discrete sequences, *Journal of Computational and Graphical Statistics* **12**(3), 604–639.

Hays, W. L.: 1994, *Statistics*, fifth edn, Wadsworth Pub Co.

Jain, S., Osherson, D., Royer, J. S. and Sharma, A.: 1999, *Systems that learn*, MIT Press.

Jiang, T. and Ravikumar, B.: 1993, Minimal NFA problems are hard, *SIAM Journal of Computation*, Vol. 22, pp. 1117–1141.

Jurafsky, D. and Martin, J. H.: 2000, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speach Recognition*, Prentice Hall.

Kearns, M. J. and Vazirani, U. V.: 1994, *An introduction to computational learning theory*, MIT Press.

Kearns, M., Mansour, Y., Ron, D., Rubinfeld, R., Shapire, R. E. and Sellie, L.: 1994, On the learnability of discrete distributions, *Symposium on Theory of Computing*, ACM, pp. 273–282.

Kermorvant, C. and Dupont, P.: 2002, Stochastic grammatical inference with multinomial tests, *Proceedings of the 6th International Colloquium on Grammatical Inference*, Vol. 2484 of *LNAI*, Springer-Verlag, pp. 149–160.

Lang, K. J., Pearlmutter, B. A. and Price, R. A.: 1998, Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm, *Grammatical Inference*, Vol. 1433 of *LNCS*, Springer.

Larsen, K. G., Petterson, P. and Yi, W.: 1997, Uppaal in a nutshell, *International journal on software tools for technology transfer* **1**(1-2), 134–152.

Leucker, M.: 2007, Learning meets verification, *Formal methods for components and objects*, Vol. 4709 of *LNCS*, Springer, pp. 127–151.

Li, M. and Vitányi, P. M.: 1991, Learning simple concepts under simple distributions, *SIAM Journal on Computing* **20**(5), 911–935.

Mitchell, T.: 1997, *Machine Learning*, McGraw Hill.

Mörchen, F. and Ultsch, A.: 2004, Mining temporal patterns in multivariate time series, *Advances in artificial intelligence*, Vol. 3238 of *LNCS*, Springer, pp. 127–140.

Oncina, J. and Garcia, P.: 1992, Inferring regular languages in polynomial update time, *Pattern Recognition and Image Analysis*, Vol. 1 of *Series in Machine Perception and Artificial Intelligence*, World Scientific, pp. 49–61.

Parekh, R. and Honavar, V.: 2000, On the relationship between models for learning in helpful environments, *ICGI*, Vol. 1891 of *LNCS*, Springer, pp. 207–220.

Parekh, R. and Honavar, V.: 2001, Learning DFA from simple examples, *Machine Learning* **44**(1-2), 9–35.

Pitt, L. and Warmuth, M.: 1989, The minimum consistent DFA problem cannot be approximated within and polynomial, *Annual ACM Symposium on Theory of Computing*, ACM, pp. 421–432.

Pnueli, A., Asarin, E., Maler, O. and Sifakis, J.: 1998, Controller synthesis for timed automata, *IFAC Symposium on System Structure and Control*, Elsevier, pp. 469–474.

Rabiner, L. R.: 1989, A tutorial on hidden Markov models and selected applications in speech recognition, *Proceedings of the IEEE*, Vol. 77, pp. 257–286.

Roddick, J. F. and Spiliopoulou, M.: 2002, A survey of temporal knowledge discovery paradigms and methods, *IEEE Transactions on Knowledge and Data Engineering* **14**(4), 750–767.

Ross, S. M.: 1997, *Introduction to Probability Models*, Academic Press.

Sen, K., Viswanathan, M. and Agha, G.: 2004, Learning continuous time Markov chains from sample executions, *Proceedings of the The Quantitative Evaluation of Systems*, pp. 146–155.

Shalizi, C. R. and Crutchfield, J. P.: 2001, Computational mechanics: pattern and prediction, structure and simplicity, *Journal of statistical physics* **104**(3-4), 817–879.

Shalizi, C. R. and Shalizi, K. L.: 2004, Blind construction of optimal nonlinear recursive predictors for disctere sequences, *AUAI*, AUAI Press, pp. 504–511.

Sipser, M.: 1997, *Introduction to the Theory of Computation*, PWS Publishing.

Springintveld, J., Vaandrager, F. W. and D'Argenio, P. R.: 2001, Testing timed automata, *Theoretical Computer Science* **254**(1-2), 225–257.

Sudkamp, T. A.: 2006, *Languages and Machines: an introduction to the theory of computer science*, third edn, Addison-Wesley.

Takami, J. T. and Sagayama, S.: 1992, A successive state splitting algorithm for efficient allophone modeling, *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Vol. 1, IEEE, pp. 573–576.

Thomas, W.: 1991, Automata on infinite objects, *Handbook of theoretical computer science (vol. B): formal models and semantics*, MIT Press.

Valiant, L. G.: 1984, A theory of the learnable, *Communications of the ACM* **27**, 1134–1142.

van der Aalst, W. and Weijters, A.: 2005, Process mining, *Process-Aware Information Systems: Bridging People and Software through Process Technology*, Wiley & Sons, pp. 235–255.

Vapnik, V. N.: 1998, *Statistical Learning Theory*, John Wiley & Sons.

Verwer, S., de Weerdt, M. and Witteveen, C.: 2005, Timed automata for behavioral pattern recognition, *Proceedings of the Belgium-Dutch Conference on Artificial Intelligence*, BNVKI, pp. 291–296.

Verwer, S., de Weerdt, M. and Witteveen, C.: 2007, An algorithm for learning real-time automata, *Proceedings of the Sixteenth Annual Machine Learning Conference of Belgium and the Netherlands*, pp. 128–135.

Verwer, S., de Weerdt, M. and Witteveen, C.: 2008a, Efficiently learning simple timed automata, *Induction of Process Models*, University of Antwerp, pp. 61–68.

Verwer, S., de Weerdt, M. and Witteveen, C.: 2008b, Polynomial distinguishability of timed automata, *Grammatical Inference: Theory and Applications*, Vol. 5278 of *Lecture Notes in Computer Science*, Springer, pp. 238–251.

Verwer, S., de Weerdt, M. and Witteveen, C.: 2009, One-clock deterministic timed automata are efficiently identifiablt in the limit, *Language and automata theory and applications*, Vol. 5457 of *LNCS*, Springer, pp. 740–751.

Whitlock, M.: 2005, Combining probsbility from independent tests: the weighted Z-method is superior to Fisher's approach, *Journal of Evolutionary Biology* **18**(5), 1368–1373.

Wiehagen, R. and Zeugmann, T.: 1995, Learning and consistency, *Algorithmic Learning for Knowledge-Based Systems*, Vol. 961 of *LNAI*, Springer, pp. 1–24.

Yates, F.: 1934, Contingency table involving small numbers and the χ^2 test, *Journal of the Royal Statistical Society* **1**, 217–235.

Yokomori, T.: 1993, Learning non-deterministic finite automata from queries and counterexamples, *Machine Intelligence*, University Press, pp. 196–189.

Yokomori, T.: 1995, On polynomial-time learnability in the limit of strictly deterministic automata, *Machine Learning* **19**(2), 153–179.

Summary

This thesis contains a study in a subfield of artificial intelligence, learning theory, machine learning, and statistics, known as system (or language) identification. System identification is concerned with constructing (mathematical) models from observations. Such a model is an intuitive description of a complex system. One of the main nice properties of models is that they can be visualized and inspected in order to provide insight into the different behaviors of a system. In addition, they can be used to perform different calculations, such as making predictions, analyzing properties, diagnosing errors, performing simulations, and many more. Models are therefore extremely useful tools for understanding, interpreting, and modifying different kinds of systems. Unfortunately, it can be very difficult to construct a model by hand. This thesis investigates the difficulty of automatically identifying models from observations.

Observations of some process and its environment are given. These observations form sequences of events. Using system identification, we try to discover the logical structure underlying these event sequences. A well-known model of such a logical structure is the deterministic finite state automaton (DFA). A DFA is a language model. Hence, its identification (or inference) problem has been well studied in the grammatical inference field. Knowing this, we want to take an established method to learn a DFA and apply it to our event sequences. However, when observing a system there often is more information than just the sequence of symbols (events): the time at which these symbols occur is also available. A DFA can be used to model this time information implicitly. A disadvantage of such an approach is that it can result in an exponential blowup of both the input data and the resulting size of the model. In this thesis, we propose a different method that uses the time information directly in order to produce a timed model.

We use a well-known DFA variant that includes the notion of time, called the timed automaton (TA). TAs are commonly used to model and reason about real-time systems. A TA models the timed information explicitly, i.e., using numbers. Because numbers use a binary representation of time, such an explicit representa-

tion can result in exponentially more compact models than an implicit representation. Therefore, also the time, space, and data required to identify TAs can be exponentially smaller than the time, space, and data required to identify DFAs. This efficiency argument is our main reason we are interested in identifying TAs.

The work in this thesis makes four major contributions to the state-of-the-art on this topic:

1. It contains a thorough theoretical study of the complexity of identifying TAs from data.
2. It provides an algorithm for identifying a simple TA from labeled data, i.e., from event sequences for which it is known to which type of system behavior they belong.
3. It extends this algorithm to the setting of unlabeled data, i.e., from event sequences with unknown behaviors.
4. It shows how to apply this algorithm to the problem of identifying a real-time monitoring system.

We now discuss these four contributions in more detail.

Theory We study the problem of identifying a TA in the paradigm of identification in the limit. The focus of this study is on the efficiency of this identification procedure. The main contributions of this study are summarized in the following list:

- We prove that deterministic TAs with two or more timed elements are not efficiently identifiable;
- We prove that deterministic TAs with a single clock (1-DTAs) are efficiently identifiable;
- We provide an algorithm ID_1-DTA for identifying these 1-DTAs efficiently.

These contributions are of importance for anyone who is interested in identifying timed systems (and DTAs in particular). Most importantly, the efficiency results tell us that identifying a 1-DTA from timed data is more efficient than identifying an equivalent DFA. Furthermore, the results show that anyone who needs to identify a DTA with two or more clocks should be satisfied with sometimes being inefficient. In general, our results are statements regarding the expressive power of single-clock and multi-clock DTAs. These statements are important by themselves, we believe that there can be other problems (such as verification) that may benefit from our results.

Algorithms for labeled data Based on the theoretical results, we develop a novel identification algorithm RTI(real-time identification) for a simple type of 1-DTA, known as a deterministic real-time automaton (DRTA). These automata can be used to model systems for which the time between consecutive events is important for the system's behavior. RTI is based on the current state-of-the-art in

DFA identification, known as evidence-driven state-merging (EDSM). To the best of our knowledge, ours is the first algorithm that can identify a TA model from timed data. In addition, it does so efficiently in time and space, and it converges efficiently to the correct DRTA in the limit.

We evaluate the performance of RTI experimentally on artificially generated data. The results of these experiments show that RTI performs good when either the number of distinct events, or the number of states is small. In addition, the experimental results show that RTI significantly outperforms identifying an equivalent DFA using EDSM. Thus, the before-mentioned theoretical results are also show clearly in our experiments.

Algorithms for unlabeled data We adapt the RTI algorithm to the more frequently occurring setting of unlabeled data. This setting occurs more often in practice because manual labeling of event sequences takes a lot of time. The result of this adaptation is the RTI+ algorithm. This algorithm is still efficient, and it converges efficiently to the correct DRTA in the limit with probability 1. RTI+ uses statistical tests in order to identify DRTAs. Although these tests are designed specifically for the purpose of identifying a DRTA from unlabeled data, they can be modified in order to identify other models such as DFAs. Hence, they also contribute to the current state-of-the-art in DFA identification.

We test the performance of the RTI+ algorithm with different statistical tests on artificially generated data. An interesting conclusion of these experiments is that in terms of data requirements it is often easier to identify a model than to set its parameters (event probabilities) correctly. Because of this, we argue that the traditional quality measures for probabilistic models are unsuited for testing the quality of the identified models. Therefore, we propose a different way to compute the quality of an identified DRTA that separates the problem of tuning the parameters from the problem of identifying the model. The results with this measure shows which of the introduced statistics achieves the best performance. The achieved performance using this statistic is shown to be sufficient in order to apply RTI+ in practice.

Applying the algorithms in practice We apply our algorithms to the problem of identifying a real-time monitoring system for driver behavior. The data is recorded by trucks of the transport company Van der Luyt. From this data, we identify DRTA models using RTI+. Then we collect a small amount of examples of an interesting example driving pattern: pulling-up too fast or normally from a traffic light. We only use a few labeled examples to label some states of the identified DRTA models. The resulting models are shown to be useful for monitoring whether the driver is pulling-up too fast. This application serves as a proof of concept of our techniques.

Our techniques have many interesting applications such as gaining insight into a real-time process, recognizing different process behaviors, identifying process models, and analyzing black-box systems.

Samenvatting

Dit proefschrift beschrijft een onderzoek naar systeemidentificatie. Systeemidentificatie is een kruising van de vakgebieden kunstmatige intelligentie, leertheorie, machineleren en statistiek. Het doel is het construeren van (mathematische) modellen op basis van observaties. Zo'n model vormt een intuïtieve beschrijving van een complex proces. Een van de vele fijne eigenschappen van modellen is dat deze gevisualiseerd en geïnspecteerd kunnen worden om zo inzicht te verkrijgen tot het proces. Verder kunnen er verscheidene berekeningen gemaakt worden met behulp van modellen. Voorbeelden zijn het voorspellen van toekomstige gebeurtenissen, het analyseren van systeemeigenschappen, het diagnosticeren van fouten, en het doen van simulaties. Modellen zijn daarom handige tools voor het begrijpen, interpreteren en modificeren van verschillende soorten systemen. Jammer genoeg is het voor mensen vaak moeilijk om deze modellen te construeren. Dit proefschrift onderzoekt hoe moeilijk het is om modellen automatisch te identificeren uit observaties.

Gegeven zijn observaties van een proces en zijn omgeving. Deze observaties vormen sequenties van gebeurtenissen. Door middel van systeemidentificatie willen we de logische structuur, die achter deze sequenties ligt, achterhalen. Een bekend voorbeeld van zo'n logische structuur is de deterministische eindige toestandsautomaat, of deterministic finite state automaton (DFA). Een DFA is een taalmodel, en het DFA identificatieprobleem is daarom een bekend probleem in het grammaticale inferentie vakgebied. Voor onze modelidentificatie kunnen we daarom een gevestigd DFA inferentiealgoritme nemen en toepassen op onze gebeurtenissequenties. DFA's kunnen echter alleen sequenties van symbolen modelleren. Wanneer men een proces observeert, is er echter meer informatie aanwezig, namelijk de tijdstippen waarop de verschillende gebeurtenissen (symbolen) plaatsvinden. Een DFA kan gebruikt worden om deze tijdsinformatie impliciet te representeren. Jammer genoeg resulteert dit in een exponentiële blow-up van zowel de modelgrootte als de benodigde grootte van de input data (aantal sequenties). In dit proefschrift stellen we een alternatief voor dat de tijdsinformatie direct gebruikt om een tijdsmodel te identificeren.

We gebruiken hiervoor een bekende variant van de DFA die overweg kan met tijdsinformatie: de tijdsautomaat, of timed automaton (TA). TA's worden vaak gebruikt om real-time systemen te modelleren en deze te analyseren. Een TA modelleert de tijdsinformatie expliciet, door gebruik te maken van getallen. Deze representatie is daarom binair, de DFA representatie is unair. De TA modellen zijn daarom exponentieel meer compact dan equivalente DFA representaties. Hierdoor kunnen ook de tijd-, ruimte-, en databenodigdheden voor een TA identificatiealgoritme exponentieel kleiner zijn dan een equivalent DFA identificatiealgoritme. Dit efficiency argument is de hoofdreden waarom we geïnteresseerd zijn in het identificeren van TA's.

Dit proefschrift bevat vier grote bijdragen aan de staat van de wetenschap over dit onderwerp:

1. Het bevat een grondige theoretische studie naar de complexiteit (moeilijkheid) van het identificeren van TA's.

2. Het omschrijft een algoritme om een eenvoudig type TA te leren van gelabelde data. Van elke sequentie in deze data is bekend tot welk gedrag van het proces ze behoren.

3. Het breidt dit algoritme uit met de mogelijkheid om toegepast te worden wanneer de data ongelabeld zijn, dus wanneer bovenstaande onbekend is.

4. Het laat door middel van een proof-of-concept zien hoe dit algoritme gebruikt kan worden om een real-time controle systeem te identificeren.

We behandelen nu elk van deze punten in meer detail.

Theorie In dit proefschrift bestuderen we het probleem om een TA te identificeren binnen het "identificatie in de limiet framework". De nadruk van deze studie ligt op de efficiëntie van deze identificatie. De hoofdbijdragen van deze studie worden kort samengevat door de volgende lijst:

- Wij bewijzen dat deterministische TA's met twee of meer tijdselementen, genaamd klokken, niet efficiënt geïdentificeerd kunnen worden.

- Wij bewijzen dat deterministische TA's met slechts één klok (1-DTA's) efficiënt identificeerbaar zijn.

- Wij ontwikkelen een algoritme om 1-DTA's efficiënt te identificeren.

Deze bijdragen zijn van belang voor iedereen die geïnteresseerd is in het identificeren van real-time systemen. De belangrijkste conclusie van deze resultaten is dat een 1-DTA efficiënter geleerd kan worden dan een equivalent DFA. Verder laten deze bijdragen zien dat iedereen die toch een DTA met twee of meer klokken wil identificeren altijd een inefficiënt algoritme zal moeten gebruiken. In het algemeen hebben onze resultaten betrekking op de expressiviteit van TA's. Deze resultaten zijn niet alleen nuttig voor het identificatieprobleem, maar ook voor andere problemen, zoals het verificatieprobleem.

Gelabelde data Gebaseerd op onze theoretische resultaten ontwikkelen we een vernieuwend identificatiealgoritme RTI (real-time identificatie) voor een eenvoudig type 1-DTA, bekend als deterministic real-time automaton (DRTA). Deze automaten kunnen gebruikt worden om processen te modelleren waarin de tijdsduur tussen twee opeenvolgende gebeurtenissen van belang is voor het gedrag van het proces. Het RTI algoritme is gebaseerd op het state-of-the-art algoritme voor het identificeren van DFA's, genaamd evidence-driven state-merging (EDSM). Voor zover wij weten, is dit het eerste algoritme dat een klasse van TA's kan identificeren uit gebeurtenissequenties met tijdsinformatie. Het doet dit ook nog eens efficiënt in tijd en ruimte, en het convergeert efficiënt naar de juiste DRTA.

We doen een experimentele analyse van RTI op artificiële data. Deze resultaten laten zien dat RTI goed presteert als ofwel het aantal symbolen (gebeurtenissen) ofwel het aantal mogelijke toestanden in de DRTA klein is. Verder laten we zien dat RTI vele malen beter presteert dan het identificeren van een equivalent DFA met behulp van EDSM. Dus de hiervoor genoemde theoretische resultaten zijn ook in experimenten duidelijk zichtbaar.

Ongelabelde data We passen het RTI algoritme aan om ook overweg te kunnen met ongelabelde data. Dit komt vaker voor in de praktijk omdat het labelen van data vaak een enorm karwei is. Het resultaat is het RTI+ algoritme. Dit algoritme is nog steeds efficiënt in tijd en ruimte, en convergeert efficiënt naar de juiste DRTA met kans 1. RTI+ maakt gebruik van specialistische statistische toetsen om de toestanden van een DRTA te herkennen. Hoewel deze toetsen specifiek geschreven zijn voor het identificeren van DRTA's, zijn ze ook toepasbaar voor het identificeren van andere modellen zoals DFA's. Deze toetsen bouwen voort op de huidige state-of-the-art in DFA identificatie.

We evalueren de performance van het RTI+ algoritme met verschillende statistische toetsen of artificiële data. Een interessante conclusie van deze experimenten is, dat het in termen van data vaak eenvoudiger is om een model te leren, dan om de parameters van dat model goed in te stellen. Op basis hiervan beargumenteren we dat de gebruikelijke maat om geïdentificeerde parameters te vergelijken niet zinvol is om de kwaliteit van geïdentificeerde modellen te bepalen. Wij kiezen dan ook een maat die de kwaliteit van de geïdentificeerde DRTA los koppelt van de kwaliteit van de geïdentificeerde parameters. De resultaten met deze maat laten zien welke statistische grootheid het best in de praktijk gebruikt kan worden. Deze statistische grootheid presteert voldoende goed voor veel toepassingsgebieden.

De praktijk We passen het RTI+ algoritme toe op het probleem van het in real-time analyseren van het bestuurdersgedrag van vrachtwagenchauffeurs. De data zijn verzameld door trucks van transportbedrijf Van der Luyt. Uit deze data identificeren we DRTA-modellen met behulp van RTI+. Vervolgens gebruiken we deze modellen om een interessant en eenvoudig patroon te herkennen: te hard en normaal optrekken. Hierbij worden slechts een paar gelabelde sequenties gebruikt om toestanden in de DRTA-modellen van labels te voorzien. De resulterende modellen blijken "te hard" en "normaal" optrekken van elkaar te kunnen onderscheiden met voldoende zekerheid. Dit geldt als proof-of-concept van de door ons ontwikkelde methode.

Deze methode heeft vele interessante toepassingen waaronder inzicht verschaffen in real-time processen, het herkennen van verschillende gedragstypen, het identificeren van procesmodellen, en het analyseren van black-box systemen.

Curriculum Vitae

Sicco Verwer was born on January 3rd 1981 in Papendrecht the Netherlands. He finished high-school (VWO) at the Rotterdams Montessori Lyceum in 1998, after which he went to study computer science at Delft University of Technology.

After four year of playing games, playing music in disco's, teaching programming and mathematics, and studying very hard, he made a good impression on an associate professor in Delft. This professor was Cees Witteveen. He sent Sicco on an internship to the USA to work for Lockheed Martin Aerospace in Fort Worth Texas. This internship strengthened Sicco's affection for theoretical computer science and artificial intelligence.

After the internship, Cees Witteveen offered Sicco to do a Master's project on the real-time analysis of transportation vehicles. The thesis resulting from this project laid the groundwork for the thesis you are reading right now. After finishing his Master's degree in 2004, Sicco continued the work of this project as a PhD student supervised by Cees Witteveen and Mathijs de Weerdt still in Delft. During his PhD work, Sicco learned a lot about learning theory, writing papers, and teaching courses. He assisted in a couple of courses every year, most notably the courses on the theory of computation and complexity theory.

Sicco now works as a postdoc at Eindhoven University of Technology together with Toon Calders on data mining with independence constraints.

SIKS dissertations series

1998

1 Johan van den Akker (CWI[3]) *DEGAS - An Active, Temporal Database of Autonomous Objects*

2 Floris Wiesman (UM) *Information Retrieval by Graphically Browsing Meta-Information*

3 Ans Steuten (TUD) *A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective*

4 Dennis Breuker (UM) *Memory versus Search in Games*

5 Eduard W. Oskamp (RUL) *Computerondersteuning bij Straftoemeting*

1999

1 Mark Sloof (VU) *Physiology of Quality Change Modelling; Automated Modelling of Quality Change of Agricultural Products*

2 Rob Potharst (EUR) *Classification using Decision Trees and Neural Nets*

3 Don Beal (UM) *The Nature of Minimax Search*

4 Jacques Penders (UM) *The Practical Art of Moving Physical Objects*

5 Aldo de Moor (KUB) *Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems*

6 Niek J.E. Wijngaards (VU) *Re-Design of Compositional Systems*

7 David Spelt (UT) *Verification Support for Object Database Design*

8 Jacques H.J. Lenting (UM) *Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation*

2000

[3] Abbreviations: SIKS - Dutch Research School for Information and Knowledge Systems; CWI - Centrum voor Wiskunde en Informatica, Amsterdam; EUR - Erasmus Universiteit, Rotterdam; KUB - Katholieke Universiteit Brabant, Tilburg; KUN - Katholieke Universiteit Nijmegen; OU - Open Universiteit; RUL - Rijksuniversiteit Leiden; RUN - Radboud Universiteit Nijmegen; TUD - Technische Universiteit Delft; TU/e - Technische Universiteit Eindhoven; UL - Universiteit Leiden; UM - Universiteit Maastricht; UT - Universiteit Twente, Enschede; UU - Universiteit Utrecht; UvA - Universiteit van Amsterdam; UvT - Universiteit van Tilburg; VU - Vrije Universiteit, Amsterdam.

1 Frank Niessink (VU) *Perspectives on Improving Software Maintenance*

2 Koen Holtman (TU/e) *Prototyping of CMS Storage Management*

3 Carolien M.T. Metselaar (UvA) *Sociaal-organisatorische Gevolgen van Kennistechnologie; een Procesbenadering en Actorperspectief*

4 Geert de Haan (VU) *ETAG, A Formal Model of Competence Knowledge for User Interface Design*

5 Ruud van der Pol (UM) *Knowledge-Based Query Formulation in Information Retrieval*

6 Rogier van Eijk (UU) *Programming Languages for Agent Communication*

7 Niels Peek (UU) *Decision-Theoretic Planning of Clinical Patient Management*

8 Veerle Coupé (EUR) *Sensitivity Analyis of Decision-Theoretic Networks*

9 Florian Waas (CWI) *Principles of Probabilistic Query Optimization*

10 Niels Nes (CWI) *Image Database Management System Design Considerations, Algorithms and Architecture*

11 Jonas Karlsson (CWI) *Scalable Distributed Data Structures for Database Management*

2001

1 Silja Renooij (UU) *Qualitative Approaches to Quantifying Probabilistic Networks*

2 Koen Hindriks (UU) *Agent Programming Languages: Programming with Mental Models*

3 Maarten van Someren (UvA) *Learning as Problem Solving*

4 Evgueni Smirnov (UM) *Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets*

5 Jacco van Ossenbruggen (VU) *Processing Structured Hypermedia: A Matter of Style*

6 Martijn van Welie (VU) *Task-Based User Interface Design*

7 Bastiaan Schonhage (VU) *Diva: Architectural Perspectives on Information Visualization*

8 Pascal van Eck (VU) *A Compositional Semantic Structure for Multi-Agent Systems Dynamics*

9 Pieter Jan 't Hoen (RUL) *Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes*

10 Maarten Sierhuis (UvA) *Modeling and Simulating Work Practice BRAHMS: a Multiagent Modeling and Simulation Language for Work Practice Analysis and Design*

11 Tom M. van Engers (VU) *Knowledge Management: The Role of Mental Models in Business Systems Design*

2002

1 Nico Lassing (VU) *Architecture-Level Modifiability Analysis*

2 Roelof van Zwol (UT) *Modelling and Searching Web-based Document Collections*

3 Henk Ernst Blok (UT) *Database Optimization Aspects for Information Retrieval*

4 Juan Roberto Castelo Valdueza (UU) *The Discrete Acyclic Digraph Markov Model in Data Mining*

5 Radu Serban (VU) *The Private Cyberspace Modeling Electronic Environments Inhabited by Privacy-Concerned Agents*

6 Laurens Mommers (UL) *Applied Legal Epistemology; Building a Knowledge-based Ontology of the Legal Domain*

7 Peter Boncz (CWI) *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*

8 Jaap Gordijn (VU) *Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas*

9 Willem-Jan van den Heuvel (KUB) *Integrating Modern Business Applications with Objectified Legacy Systems*

10 Brian Sheppard (UM) *Towards Perfect Play of Scrabble*

11 Wouter C.A. Wijngaards (VU) *Agent Based Modelling of Dynamics: Biological and Organisational Applications*

12 Albrecht Schmidt (UvA) *Processing XML in Database Systems*

13 Hongjing Wu (TU/e) *A Reference Architecture for Adaptive Hypermedia Applications*

14 Wieke de Vries (UU) *Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems*

15 Rik Eshuis (UT) *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*

16 Pieter van Langen (VU) *The Anatomy of Design: Foundations, Models and Applications*

17 Stefan Manegold (UvA) *Understanding, Modeling, and Improving Main-Memory Database Performance*

2003

1 Heiner Stuckenschmidt (VU) *Ontology-Based Information Sharing in Weakly Structured Environments*

2 Jan Broersen (VU) *Modal Action Logics for Reasoning About Reactive Systems*

3 Martijn Schuemie (TUD) *Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy*

4 Milan Petkovic (UT) *Content-Based Video Retrieval Supported by Database Technology*

5 Jos Lehmann (UvA) *Causation in Artificial Intelligence and Law – A Modelling Approach*

6 Boris van Schooten (UT) *Development and Specification of Virtual Environments*

7 Machiel Jansen (UvA) *Formal Explorations of Knowledge Intensive Tasks*

8 Yong-Ping Ran (UM) *Repair-Based Scheduling*

9 Rens Kortmann (UM) *The Resolution of Visually Guided Behaviour*

10 Andreas Lincke (UT) *Electronic Business Negotiation: Some Experimental Studies on the Interaction between Medium, Innovation Context and Cult*

11 Simon Keizer (UT) *Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks*

12 Roeland Ordelman (UT) *Dutch Speech Recognition in Multimedia Information Retrieval*

13 Jeroen Donkers (UM) *Nosce Hostem – Searching with Opponent Models*

14 Stijn Hoppenbrouwers (KUN) *Freezing Language: Conceptualisation Processes across ICT-Supported Organisations*

15 Mathijs de Weerdt (TUD) *Plan Merging in Multi-Agent Systems*

16 Menzo Windhouwer (CWI) *Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouse*

17 David Jansen (UT) *Extensions of Statecharts with Probability, Time, and Stochastic Timing*

18 Levente Kocsis (UM) *Learning Search Decisions*

2004

1 Virginia Dignum (UU) *A Model for Organizational Interaction: Based on Agents, Founded in Logic*

2 Lai Xu (UvT) *Monitoring Multi-party Contracts for E-business*

3 Perry Groot (VU) *A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving*

4 Chris van Aart (UvA) *Organizational Principles for Multi-Agent Architectures*

5 Viara Popova (EUR) *Knowledge Discovery and Monotonicity*

6 Bart-Jan Hommes (TUD) *The Evaluation of Business Process Modeling Techniques*

7 Elise Boltjes (UM) *Voorbeeld*$_{IG}$ *Onderwijs; Voorbeeldgestuurd Onderwijs, een Opstap naar Abstract Denken, vooral voor Meisjes*

8 Joop Verbeek (UM) *Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale Politiële Gegevensuitwisseling en Digitale Expertise*

9 Martin Caminada (VU) *For the Sake of the Argument; Explorations into Argument-based Reasoning*

10 Suzanne Kabel (UvA) *Knowledge-rich Indexing of Learning-objects*

11 Michel Klein (VU) *Change Management for Distributed Ontologies*

12 The Duy Bui (UT) *Creating Emotions and Facial Expressions for Embodied Agents*

13 Wojciech Jamroga (UT) *Using Multiple Models of Reality: On Agents who Know how to Play*

14 Paul Harrenstein (UU) *Logic in Conflict. Logical Explorations in Strategic Equilibrium*

15 Arno Knobbe (UU) *Multi-Relational Data Mining*

16 Federico Divina (VU) *Hybrid Genetic Relational Search for Inductive Learning*

17 Mark Winands (UM) *Informed Search in Complex Games*

18 Vania Bessa Machado (UvA) *Supporting the Construction of Qualitative Knowledge Models*

19 Thijs Westerveld (UT) *Using generative probabilistic models for multimedia retrieval*

20 Madelon Evers (Nyenrode) *Learning from Design: facilitating multidisciplinary design teams*

2005

1 Floor Verdenius (UvA) *Methodological Aspects of Designing Induction-Based Applications*

2 Erik van der Werf (UM) *AI techniques for the game of Go*

3 Franc Grootjen (RUN) *A Pragmatic Approach to the Conceptualisation of Language*

4 Nirvana Meratnia (UT) *Towards Database Support for Moving Object data*

5 Gabriel Infante-Lopez (UvA) *Two-Level Probabilistic Grammars for Natural Language Parsing*

6 Pieter Spronck (UM) *Adaptive Game AI*

7 Flavius Frasincar (TU/e) *Hypermedia Presentation Generation for Semantic Web Information Systems*

8 Richard Vdovjak (TU/e) *A Model-driven Approach for Building Distributed Ontology-based Web Applications*

9 Jeen Broekstra (VU) *Storage, Querying and Inferencing for Semantic Web Languages*

10 Anders Bouwer (UvA) *Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments*

11 Elth Ogston (VU) *Agent Based Matchmaking and Clustering - A Decentralized Approach to Search*

12 Csaba Boer (EUR) *Distributed Simulation in Industry*

13 Fred Hamburg (UL) *Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen*

14 Borys Omelayenko (VU) *Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics*

15 Tibor Bosse (VU) *Analysis of the Dynamics of Cognitive Processes*

16 Joris Graaumans (UU) *Usability of XML Query Languages*

17 Boris Shishkov (TUD) *Software Specification Based on Re-usable Business Components*

18 Danielle Sent (UU) *Test-selection strategies for probabilistic networks*

19 Michel van Dartel (UM) *Situated Representation*

20 Cristina Coteanu (UL) *Cyber Consumer Law, State of the Art and Perspectives*

21 Wijnand Derks (UT) *Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics*

2006

1 Samuil Angelov (TU/e) *Foundations of B2B Electronic Contracting*

2 Cristina Chisalita (VU) *Contextual issues in the design and use of information technology in organizations*

3 Noor Christoph (UvA) *The role of metacognitive skills in learning to solve problems*

4 Marta Sabou (VU) *Building Web Service Ontologies*

5 Cees Pierik (UU) *Validation Techniques for Object-Oriented Proof Outlines*

6 Ziv Baida (VU) *Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling*

7 Marko Smiljanic (UT) *XML schema matching – balancing efficiency and effectiveness by means of clustering*

8 Eelco Herder (UT) *Forward, Back and Home Again - Analyzing User Behavior on the Web*

9 Mohamed Wahdan (UM) *Automatic Formulation of the Auditor's Opinion*

10 Ronny Siebes (VU) *Semantic Routing in Peer-to-Peer Systems*

11 Joeri van Ruth (UT) *Flattening Queries over Nested Data Types*

12 Bert Bongers (VU) *Interactivation - Towards an e-cology of people, our technological environment, and the arts*

13 Henk-Jan Lebbink (UU) *Dialogue and Decision Games for Information Exchanging Agents*

14 Johan Hoorn (VU) *Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change*

15 Rainer Malik (UU) *CONAN: Text Mining in the Biomedical Domain*

16 Carsten Riggelsen (UU) *Approximation Methods for Efficient Learning of Bayesian Networks*

17 Stacey Nagata (UU) *User Assistance for Multitasking with Interruptions on a Mobile Device*

18 Valentin Zhizhkun (UvA) *Graph transformation for Natural Language Processing*

19 Birna van Riemsdijk (UU) *Cognitive Agent Programming: A Semantic Approach*

20 Marina Velikova (UvT) *Monotone models for prediction in data mining*

21 Bas van Gils (RUN) *Aptness on the Web*

22 Paul de Vrieze (RUN) *Fundaments of Adaptive Personalisation*

23 Ion Juvina (UU) *Development of Cognitive Model for Navigating on the Web*

24 Laura Hollink (VU) *Semantic Annotation for Retrieval of Visual Resources*

25 Madalina Drugan (UU) *Conditional log-likelihood MDL and Evolutionary MCMC*

26 Vojkan Mihajlovic (UT) *Score Region Algebra: A Flexible Framework for Structured Information Retrieval*

27 Stefano Bocconi (CWI) *Vox Populi: generating video documentaries from semantically annotated media repositories*

28 Borkur Sigurbjornsson (UvA) *Focused Information Access using XML Element Retrieval*

2007

1 Kees Leune (UvT) *Access Control and Service-Oriented Architectures*

2 Wouter Teepe (RUG) *Reconciling Information Exchange and Confidentiality: A Formal Approach*

3 Peter Mika (VU) *Social Networks and the Semantic Web*

4 Jurriaan van Diggelen (UU) *Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach*

5 Bart Schermer (UL) *Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance*

6 Gilad Mishne (UvA) *Applied Text Analytics for Blogs*

7 Natasa Jovanovic' (UT) *To Whom It May Concern - Addressee Identification in Face-to-Face Meetings*

8 Mark Hoogendoorn (VU) *Modeling of Change in Multi-Agent Organizations*

9 David Mobach (VU) *Agent-Based Mediated Service Negotiation*

10 Huib Aldewereld (UU) *Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols*

11 Natalia Stash (TU/e) *Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System*

12 Marcel van Gerven (RUN) *Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty*

13 Rutger Rienks (UT) *Meetings in Smart Environments; Implications of Progressing Technology*

14 Niek Bergboer (UM) *Context-Based Image Analysis*

15 Joyca Lacroix (UM) *NIM: a Situated Computational Memory Model*

16 Davide Grossi (UU) *Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems*

17 Theodore Charitos (UU) *Reasoning with Dynamic Networks in Practice*

18 Bart Orriens (UvT) *On the development and management of adaptive business collaborations*

19 David Levy (UM) *Intimate relationships with artificial partners*

20 Slinger Jansen (UU) *Customer Configuration Updating in a Software Supply Network*

21 Karianne Vermaas (UU) *Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005*

22 Zlatko Zlatev (UT) *Goal-oriented design of value and process models from patterns*

23 Peter Barna (TU/e) *Specification of Application Logic in Web Information Systems*

24 Georgina Ramírez Camps (CWI) *Structural Features in XML Retrieval*

25 Joost Schalken (VU) *Empirical Investigations in Software Process Improvement*

2008

1 Katalin Boer-Sorbán (EUR) *Agent-Based Simulation of Financial Markets: A modular, continuous-time approach*

2 Alexei Sharpanskykh (VU) *On Computer-Aided Methods for Modeling and Analysis of Organizations*

3 Vera Hollink (UvA) *Optimizing hierarchical menus: a usage-based approach*

4 Ander de Keijzer (UT) *Management of Uncertain Data - towards unattended integration*

5 Bela Mutschler (UT) *Modeling and simulating causal dependencies on process-aware information systems from a cost perspective*

6 Arjen Hommersom (RUN) *On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective*

7 Peter van Rosmalen (OU) *Supporting the tutor in the design and support of adaptive e-learning*

8 Janneke Bolt (UU) *Bayesian Networks: Aspects of Approximate Inference*

9 Christof van Nimwegen (UU) *The paradox of the guided user: assistance can be counter-effective*

10 Wauter Bosma (UT) *Discourse oriented Summarization*

11 Vera Kartseva (VU) *Designing Controls for Network Organizations: a Value-Based Approach*

12 Jozsef Farkas (RUN) *A Semiotically oriented Cognitive Model of Knowlegde Representation*

13 Caterina Carraciolo (UvA) *Topic Driven Access to Scientific Handbooks*

14 Arthur van Bunningen (UT) *Context-Aware Querying; Better Answers with Less Effort*

15 Martijn van Otterlo (UT) *The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains*

16 Henriette van Vugt (VU) *Embodied Agents from a User's Perspective*

17 Martin Op't Land (TUD) *Applying Architecture and Ontology to the Splitting and Allying of Enterprises*

18 Guido de Croon (UM) *Adaptive Active Vision*

19 Henning Rode (UT) *From document to entity retrieval: improving precision and performance of focused text search*

20 Rex Arendsen (UvA) *Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met een overheid op de administratieve lasten van bedrijven*

21 Krisztian Balog (UvA) *People search in the enterprise*

22 Henk Koning (UU) *Communication of IT-architecture*

23 Stefan Visscher (UU) *Bayesian network models for the management of ventilator-associated pneumonia*

24 Zharko Aleksovski (VU) *Using background knowledge in ontology matching*

25 Geert Jonker (UU) *Efficient and Equitable exchange in air traffic management plan repair using spender-signed currency*

26 Marijn Huijbregts (UT) *Segmentation, diarization and speech transcription: surprise data unraveled*

27 Hubert Vogten (OU) *Design and implementation strategies for IMS learning design*

28 Ildiko Flesh (RUN) *On the use of independence relations in Bayesian networks*

29 Dennis Reidsma (UT) *Annotations and subjective machines- Of annotators, embodied agents, users, and other humans*

30 Wouter van Atteveldt (VU) *Semantic network analysis: techniques for extracting, representing and querying media content*

31 Loes Braun (UM) *Pro-active medical information retrieval*

32 Trung B. Hui (UT) *Toward affective dialogue management using partially observable markov decision processes*

33 Frank Terpstra (UvA) *Scientific workflow design; theoretical and practical issues*

34 Jeroen de Knijf (UU) *Studies in Frequent Tree Mining*

35 Benjamin Torben-Nielsen (UvT) *Dendritic morphology: function shapes structure*

2009

1 Rasa Jurgelenaite (RUN) *Symmetric Causal Independence Models*

2 Willem Robert van Hage (VU) *Evaluating Ontology-Alignment Techniques*

3 Hans Stol (UvT) *A Framework for Evidence-based Policy Making Using IT*

4 Josephine Nabukenya (RUN) *Improving the Quality of Organisational Policy Making using Collaboration Engineering*

5 Sietse Overbeek (RUN) *Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality*

6 Muhammad Subianto (UU) *Understanding Classification*

7 Ronald Poppe (UT) *Discriminative Vision-Based Recovery and Recognition of Human Motion*

8 Volker Nannen (VU) *Evolutionary Agent-Based Policy Analysis in Dynamic Environments*

9 Benjamin Kanagwa (RUN) *Design, Discovery and Construction of Service-oriented Systems*

10 Jan Wielemaker (UvA) *Logic programming for knowledge-intensive interactive applications*

11 Alexander Boer (UvA) *Legal Theory, Sources of Law & the Semantic Web*

12 Peter Massuthe (TU/e, Humboldt-Universtät zu Berlin) *Operating Guidelines for Services*

13 Steven de Jong (UM) *Fairness in Multi-Agent Systems*

14 Maksym Korotkiy (VU) *From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)*

15 Rinke Hoekstra (UvA) *Ontology Representation - Design Patterns and Ontologies that Make Sense*

16 Fritz Reul (UvT) *New Architectures in Computer Chess*

17 Laurens van der Maaten (UvT) *Feature Extraction from Visual Data*

18 Fabian Groffen (CWI) *Armada, An Evolving Database System*

19 Valentin Robu (CWI) *Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets*

20 Bob van der Vecht (UU) *Adjustable Autonomy: Controling Influences on Decision Making*

21 Stijn Vanderlooy (UM) *Ranking and Reliable Classification*

22 Pavel Serdyukov (UT) *Search For Expertise: Going beyond direct evidence*

23 Peter Hofgesang (VU) *Modelling Web Usage in a Changing Environment*

24 Annerieke Heuvelink (VU) *Cognitive Models for Training Simulations*

25 Alex van Ballegooij (CWI) *"RAM: Array Database Management through Relational Mapping"*

26 Fernando Koch (UU) *An Agent-Based Model for the Development of Intelligent Mobile Services*

27 Christian Glahn (OU) *Contextual Support of social Engagement and Reflection on the Web*

28 Sander Evers (UT) *Sensor Data Management with Probabilistic Models*

29 Stanislav Pokraev (UT) *Model-Driven Semantic Integration of Service-Oriented Applications*

30 Marcin Zukowski (CWI) *Balancing vectorized query execution with bandwidth-optimized storage*

31 Sofiya Katrenko (UvA) *A Closer Look at Learning Relations from Text*

32 Rik Farenhorst and Remco de Boer (VU) *Architectural Knowledge Management: Supporting Architects and Auditors*

33 Khiet Truong (UT) *How Does Real Affect Affect Affect Recognition In Speech?*

34 Inge van de Weerd (UU) *Advancing in Software Product Management: An Incremental Method Engineering Approach*

2010

1 Matthijs van Leeuwen (UU) *Patterns that Matter*

2 Ingo Wassink (UT) *Work flows in Life Science*

3 Joost Geurts (CWI) *A Document Engineering Model and Processing Framework for Multimedia documents*

4 Olga Kulyk (UT) *Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments*

5 Claudia Hauff (UT) *Predicting the Effectiveness of Queries and Retrieval Systems*

6 Sander Bakkes (UvT) *Rapid Adaptation of Video Game AI*

7 Wim Fikkert (UT) *A Gesture interaction at a Distance*

8 Krzysztof Siewicz (UL) *Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments*

9 Hugo Kielman (UL) *Politiële gegevensverwerking en Privacy, Naar een effectieve waarborging*

10 Rebecca Ong (UL) *Mobile Communication and Protection of Children*

11 Adriaan Ter Mors (TUD) *The world according to MARP: Multi-Agent Route Planning*

12 Susan van den Braak (UU) *Sensemaking software for crime analysis*

13 Gianluigi Folino (RUN) *High Performance Data Mining using Bio-inspired techniques*

14 Sander van Splunter (VU) *Automated Web Service Reconfiguration*

15 Lianne Bodenstaff (UT) *Managing Dependency Relations in Inter-Organizational Models*

16 Sicco Verwer (TUD) *Efficient Identification of Timed Automata, theory and practice*