Anomaly Detection as a One-class Problem in Discrete Event Systems

Timo Klerx

Dissertation
in Computer Science

submitted to the

Faculty of Computer Science,
Electrical Engineering and Mathematics

Paderborn University

in partial fulfillment of the requirements for the degree of

doctor rerum naturalium
(Dr. rer. nat.)

Paderborn, May 2017

Abstract

Technical systems become more and more complex. Even human experts cannot fully understand the behavior of some systems and thus, cannot define precise rules to detect when a system fails. On the other hand, sensors have become very cheap and machines can send their measurements using the Internet of Things (IoT) to any place in the world where the data can be analyzed. This price decay allows incorporating a lot of sensors into machines and perceiving the state of the machine very accurately. Furthermore, the sent data can be easily processed in data centers with huge computational power. To detect a system's failures while overcoming its complexity, experts only state when a system's behavior is *normal*. Solely using data of the normal operation a deviation from this normal behavior can be detected. Additionally, machines often execute a sequence of actions repeatedly while every action changes the machine state which can be detected using some built-in sensors.

To model systems, *discrete event systems* (DES) are a feasible system class that abstract from the real (continuous) machine states but usually preserve the important information using discrete states only. We model machines as such systems and observe sequences of machine events and the time between two subsequent events. Furthermore, we assume that the observed sequences only describe the normal behavior of the machines and thus, the described failure detection problem corresponds to a *one-class* problem.

In this thesis we present a new timed-automata-based approach consisting of three steps that tackle the described problem of detecting anomalies (e.g. failures or misuse) in timed sequences (e.g. consisting of machine events): We start with developing the Probabilistic Deterministic Timed Transition Automaton (PDTTA) that can represent the normal behavior of a discrete event system. Then, we develop the Probabilistic Deterministic Timed Transition Automaton Learning (ProDTTAL) algorithm that learns a PDTTA given a set of normal timed sequences by first learning solely the order of events followed by the time behavior of subsequent events. Last, the Automata-based Anomaly Detection Algorithm (AmAnDA) solves the anomaly detection problem and labels a given sequence as normal and abnormal with respect to a previously learned timed automaton model.

Compared to existing timed automaton models, the PDTTA represents the time behavior between events more accurately but is inferior in some elaborated scenarios. AmAnDA is the first anomaly detection algorithm that can be used for various automaton models (e.g. PDFA, PDTTA, PDRTA, PDTA) while improving the anomaly detection performance using vector-based one-class classifiers. In the experimental evaluation, we first develop a systematic approach to evaluate sequence-based anomaly detection algorithms. Then, we utilize this approach to compare ProDTTAL with the state-of-the-art algorithms Real-Time Identification from Positive Data (RTI+) and Bottom-Up Timed Learning Algorithm (BUTLA) on synthetic and real-world ATM data. The results of this comparison are twofold, as ProDTTAL outperforms RTI+ and BUTLA only in some scenarios, but is inferior in others.

Zusammenfassung

Technische Systeme werden immer komplexer, sodass selbst Experten das Verhalten mancher Systeme nicht vollständig verstehen. Daher ist es sehr schwer, präzise Regeln aufzustellen, die Fehler im System erkennen. Zugleich sind Sensoren sehr preiswert geworden und Maschinen können ihre Messungen durch das Internet of Things (IoT) überall hin senden, wo die Daten analysiert werden können. Durch diesen Preisverfall können Maschinen mit vielen Sensoren ausgestattet werden, sodass ihr Zustand sehr präzise bestimmt werden kann. Außerdem können gesendete Daten in Rechenzentren mit viel Rechenleistung verarbeitet werden. Um nun Fehler in Systemen trotz ihrer Komplexität zu erkennen, beschreiben Experten nur das *normale* Verhalten eines Systems. Abweichungen von diesem normalen Verhalten können entdeckt werden, wenn ausschließlich Daten der normalen Bedienung vorliegen. Zusätzlich führen Maschinen oft repetetive Sequenzen von Aktionen aus, wobei jede Aktion eine Zustandsänderung auslöst, die durch eingebaute Sensoren erkannt werden kann.

Um Systeme zu modellieren, können unter anderem *Diskrete Ereignis Systeme* (DES) genutzt werden. DES abstrahieren zum einen den realen Maschinenzustand, modellieren aber mit diskreten Zuständen meist die wichtigen Informationen. Wir modellieren Maschinen als DES und beobachten Sequenzen von Maschinenaktionen und die Zeit zwischen zwei Aktionen. Dabei nehmen wir an, dass alle beobachteten Sequenzen das Normalverhalten der Maschinen beschreiben, wodurch das beschriebene Problem der Fehlererkennung einem *Ein-Klassen* (eng. one-class) Problem entspricht.

In dieser Arbeit präsentieren wir einen dreiteiligen Ansatz basierend auf zeitlichen Automaten, der das zuvor beschriebene Problem der Anomalieerkennung (z.B. Fehler oder Missbrauch) für zeitliche Sequenzen (z.B. bestehend aus Maschinenereignissen) löst: Zuerst entwickeln wir den Probabilistischen Deterministischen Zeitlichen Transitions Automaten (eng. PDTTA), der das Normalverhalten von DES abbilden kann. Dann entwickeln wir den PDTTA-Lern-Algorithmus (ProDTTAL), der bei Eingabe einer Menge von normalen, zeitlichen Sequenzen einen PDTTA lernt. Dazu wird zuerst nur die Ereignisreihenfolge gelernt, gefolgt von dem Zeitverhalten aufeinander folgender Ereignisse. Abschließend löst der Automatenbasierte Anomalieerkennungsalgorithmus (eng. AmAnDA) das Problem der Anomalieerkennung, indem eine gegebene Sequenz als normal oder anormal gekennzeichnet wird im Bezug auf einen zuvor gelernten zeitlichen Automaten.

Verglichen mit bestehenden zeitlichen Automatenmodellen repräsentiert der PDTTA das Zeitverhalten zwischen Ereignissen genauer, ist in gewissen Szenarien aber unterlegen. AmAnDA ist der erste Algorithmus zur Anomalieerkennung, der für verschiedene Automatenmodelle (z.B. PDFA, PDTTA, PDRTA, PDTA) genutzt werden kann und gleichzeitig die Performance der Anomalieerkennung durch vektorbasierte Ein-Klassen-Klassifizierer verbessert. In der experimentellen Evaluation entwickeln wir zuerst einen systematischen Ansatz, um sequenzbasierte Anomalieerkennungsalgorithmen zu evaluieren. Anschließend verwenden wir diesen Ansatz, um ProDTTAL mit den modernsten Algorithmen RTI+ und BUTLA auf synthetischen und realen Daten zu evaluieren. Dabei übertrifft ProDTTAL RTI+ und BUTLA in einigen Szenarios, während er in anderen Szenarios unterliegt.

Contents

List of Figures

List of Tables

1 Introduction

Nowadays, technical systems become more and more complex because more and more functionalities are built into them. Every machine e.g. in manufacturing and every car contains a bunch of actuators and sensors that control some process and constantly measure it. In Germany, this digitization of processes in the industry is called "Industrie 4.0" which describes the process of constantly measuring technical systems and gaining benefit from the measured data. Today, terms like *Internet of Things* (IoT) describe that every machine is connected to the internet and can send its measured data to places where the data is analyzed. At the same time, these machines usually contain a lot of subsystems that solve certain tasks. Because of the increasing complexity of these systems it is difficult for computers and human experts to automatically detect failures or bugs. Failures can be caused by two different reasons: First, a bug in the system can cause the system itself to fail or second, the system can become vulnerable for attacks. [Gib15] reports of an airbus A400M crash because of a software failure that caused the turbines to stop. In [Gre15] hackers attacked the software system of a jeep, hijacked it and disabled the brakes of the driving car (for demonstration purposes). Most of these failures could be prevented if we monitor the potentially failing system as a whole. If we precisely know how the system should behave we can detect those failures.

Unfortunately, we encounter two problems if we want to monitor complex systems: First, we do not exactly know the desired behavior (called the *behavior model*) and all allowed states of the system because it is simply too complex. Second, no human can monitor a whole complex technical system, e.g. by looking at 100 sensors simultaneously and detect a failure in a combination of those sensor signals. Furthermore, it is too expensive and slow to have one human monitoring one machine at a time and communicating with other humans to detect a combination of abnormal signals. Therefore, an automatic approach is needed to monitor technical systems.

Most often, a failure can be seen as a deviation from the normal behavior which is called an *anomaly*. Figure 1.1 shows the process described so far. This process is called *model-based anomaly detection* (also known as *model-based diagnosis*). We are faced with a system which we can observe and we want to know when this system does not behave as expected. Additionally, we want to build a behavior model of the system (called *model formation*). Then, we can *simulate* the behavior model and compare the expected behavior with the *observed* (real) behavior of the system. If these two types of behavior deviate with respect to some criterion this deviation indicates an *anomaly*.

Up to now, we described the process of model-based anomaly detection which can be handled by humans as well as by computers. In the past, human experts received a system model and observed a system by watching sensor signals. With increasing

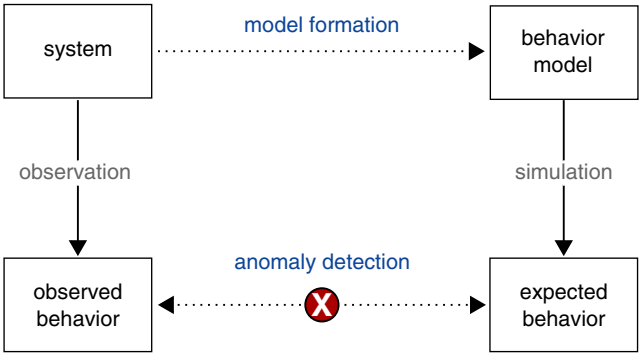


Figure 1.1: Essential steps of model-based anomaly detection (from [Kle+14]).

system complexity this process of model-based anomaly detection is automatically performed by computers in two steps: First, we automate the model formation step—building (or learning) a model from a given system. Second, we automate the step of anomaly detection—detecting a deviation between the real system and its model.

Both of these tasks can be solved separately: First, for a given real system a behavior model should be built—by a human or automatically. Second, given such a behavior model anomalies should be detected. In this thesis, we apply techniques from artificial intelligence to solve both tasks. Adapting the notation of the field of artificial intelligence, we deal with *automatic model formation* and *anomaly detection*.

The anomaly detection problem can be modeled as a *one-class problem*. Usually, a one-class setting occurs in the context of *classification*. In this classification setting we want to assign a *class label* to a sample (e.g. some execution steps of a process). In the one-class setting we only have samples from the *normal* class and do not know how anomalies look like. If we face a new sample we want to label it as normal or abnormal. Hence, we define anomalies as everything we would not label as normal.

In addition to automatically detecting anomalies, we need to identify a *system class* which can represent certain types of systems as an abstraction of the real-world system. Real-world systems can be modeled in different ways and depending on the chosen system model a different behavior model has to be applied. We focus on *Discrete Event Systems (DESs)* because they are a limited but powerful abstraction of real-world systems. In DES we usually deal with *sequences* of events that occur at various time stamps.

Summarizing, we focus on sequence-based approaches (incorporating time stamps) for anomaly detection (as a one-class problem). Moreover, we are looking for a model and an algorithm to detect ATM manipulation and fraud in a research project with the ATM manufacturer Diebold Nixdorf. ATMs are very complex systems that can be modeled as DES, but manually modeling the normal behavior of an ATM is almost impossible—even for domain experts. Additionally, ATMs emit a lot of data in form of timed sequences that can be used to learn a behavior model. As state-of-the-art approaches for model learning, the models themselves and the anomaly detection algorithms suffer certain drawbacks, we needed to develop a new automaton model, a learning algorithm for this model and an anomaly detection

algorithm that circumvents most of the drawbacks.

In the next section, we briefly present anomaly detection approaches that partly solve the anomaly detection problem tackled in this thesis.

1.1 Background

Anomaly detection has been applied in many technical systems. In 2000, the Defense Advanced Research Projects Agency (DARPA) organized a challenge to detect intrusion into a network system analyzing the TCP payload [Lip+00]. Even years after the competition in 2004, the Payload-based AnomalY Detector (PAYL) was developed to automatically detect network intrusion [WS04]. In [Nig+12], instead of detecting functional anomalies a hybrid automaton is used to detect abnormal energy consumptions in production plants. For identifying faults in web service processes a web service orchestration language is transformed into a synchronized automaton to detect faulty executions of specific services [Yan+09]. [Men+10] reports on model-based diagnosis of a real-world electrical power system in a NASA research center using Bayesian networks. In [AS12], an expert system is applied to reduce the time for detecting faulty parts in an automotive systems based on error codes.

In some of these approaches, rule engines are used where a human expert defines rules of the normal behavior. In others, the system behavior is not modeled sequence-based but the system's behavior is examined at a fixed frequency. We will apply our automatic anomaly detection approach in a sequence-based manner for detecting manipulations of Automated Teller Machines (ATMs) to prevent fraud (cf. Section 6.5).

Real-Time Identification from Positive Data (RTI+) [VdW10] is the first algorithm that solves the problem of automatic model generation for timed sequences. The algorithm can be applied for generating a so called Probabilistic Deterministic Real-Time Automaton (PDRTA) from a set of timed and normal sequences. But RTI+ only deals with the automatic model formation and does not solve the anomaly detection problem. Furthermore, it requires the input of histogram bins which are hard to define (even for human experts).

The Bottom-Up Timed Learning Algorithm (BUTLA) [Mai14] is the second algorithm that automatically generates models for given data containing timed sequences. The resulting model called Probabilistic Deterministic Timed Automaton (PDTA) can be used in the ANOmaly Detection Algorithm (ANODA) to detect anomalies for new sample sequences. However, BUTLA and ANODA have certain drawbacks which we will point out in the remainder of this thesis (cf. Sections 2.4.3.1 and 6.2.3.2).

1.2 Main Contributions

In this thesis, we deal with the problem of sequence-based anomaly detection in discrete event systems and its evaluation as a one-class problem. We can split our main contributions into six different parts:

Relating Research Areas We identify different research areas (cf. Chapter 3) that tackle the same or at least similar problems—namely Process Mining, Grammatical Inference and Sequence-based Anomaly Detection. Surprisingly, these research areas develop mostly independent and do not apply methods from the other area. We relate these research areas and give an overview on the different notations and approaches.

Automaton Model PDTTA We present a new automaton model called Probabilistic Deterministic Timed Transition Automaton (PDTTA) (cf. Section 4.2) that represents a good trade-off between the models learned by the state-of-the-art algorithms RTI+ (PDRTA) and BUTLA (PDTA). The PDTTA does not require the specification of global histograms (as a PDRTA) but represents time behavior locally. Furthermore, it models time behavior more precisely than a PDTA.

PDTTA Learning Algorithm ProDTTAL For learning a PDTTA from data we present the modular Probabilistic Deterministic Timed Transition Automaton Learning (ProDTTAL) in Section 4.3. ProDTTAL is a combination of learning the event structure with a PDFA learner and modeling the time behavior with Kernel Density Estimation (KDE). We analyze ProDTTAL's runtime (polynomial in terms of the size of the training set and some other factors; Section 4.3.2) and show that ProDTTAL converges in the limit (under some assumptions). Additionally, we give an overview on the different timed automata learning algorithms.

Automata-based Anomaly Detection Algorithm (AmAnDA) We present the Automata-based Anomaly Detection Algorithm (AmAnDA) as anomaly detection algorithm (cf. Section 4.4) that detects anomalies for a given timed automaton learned with RTI+, BUTLA or ProDTTAL and an input sequence. AmAnDA is built in a modular way and can use most state-of-the-art one-class classification algorithms. Not considering the runtime of the one-class algorithm, AmAnDA requires polynomial runtime in terms of the length of the given sequence (and some other factors that we present in Section 4.4.2.3).

Systematical Evaluation Method for Anomaly Detection Approaches We present a systematical evaluation method for anomaly detection approaches in Discrete Event System (DES) (cf. Chapter 5). To be able to evaluate anomaly detection approaches, we need a test data set that contains normal as well as anomalous sequences. Therefore, we identify five types of anomalies that can occur in DES and show how every type of these anomalies can be interspersed into data sets that do not contain anomalies. For interspersing anomalies, we propose two different approaches—namely the random anomaly insertion and the model-based anomaly insertion. Furthermore, we propose a method to systematically evaluate the scalability of automata-based algorithms in terms of runtime and memory consumption.

Empirical Comparison Our last contribution is the empirical comparison of the timed automata learning algorithms RTI+, BUTLA and ProDTTAL in Chapter 6. To the best of our knowledge, neither of these algorithms have been compared with each other. We evaluate the algorithms' performance on synthetically generated

data sets and on a real-world data set gathered on a publicly available Automated Teller Machine (ATM). Finally, we investigate how well the algorithms scale using our proposed method.

Parts of this thesis have already been published. In [KAK14; Kle+14], we present and refine the PDTTA, ProDTTAL and a naive version of AmAnDA. In [Kle+14] we additionally identify four types of anomalies that can occur in Discrete Event System (DES). Furthermore, we describe how to randomly intersperse these anomalies into an anomaly-free data set.

1.3 Structure

The structure of this thesis is closely related to the main contributions presented in the preceding section. First, we present existing concepts in Chapter 2. In particular, we review probability density functions followed by different system classes for modeling systems. Then we present automata classes and automaton models. After that we describe inference algorithms that infer automata of the different classes. We finish the chapter with the presentation of one-class classification algorithms.

In Chapter 3, we review related work. First, we relate the research areas of Process Mining, Grammatical Inference and Sequence-based Anomaly Detection. For every of these areas we present the mostly used models and algorithms and describe the different notations for events, sequences, data sets and the problem of anomaly detection. We also point out the differences between the research areas. Then, we review algorithms from other research areas that also solve the anomaly detection problem on sequence-based data sets. We start with approaches that deal with anomaly detection for vector-based data sets, followed by anomaly detection on data streams in which the data can be read only once. We conclude the chapter by presenting work on ATM fraud detection.

Chapter 4 contains three main contributions. First, we introduce the new automaton model PDTTA, show how this model is learned with the algorithm ProDTTAL and analyze ProDTTAL's runtime. After pointing out differences to the other automata learning algorithms RTI+ and BUTLA we develop the AmAnDA which solves the anomaly detection task given a timed automaton (that obeys some properties) and an unlabeled timed sequence. The chapter finishes with a discussion on the differences between one-class, two-class and multi-class sequence-based anomaly detection.

In Chapter 5, we present how sequence-based anomaly detection algorithms can be evaluated. First, we review different performance metrics for anomaly detection (including their strengths and weaknesses). Then, we present five types of anomalies that can occur in DES, followed by how every type of these anomalies can be inserted into data sets that do not contain any anomalies. We conclude the chapter with an approach how we can create experiments for evaluating the scalability of sequence-based anomaly detection algorithms.

In Chapter 6, we compare the approaches RTI+, BUTLA and ProDTTAL experimentally. We first describe the experiment setup and perform preliminary experiments on synthetic data which we sample from a manually created model. Then, we sample more data from similar automata and apart from interspersing

anomalies into normal data sets we directly sample anomalies from these automaton models. After evaluating the different approaches on real-world data gathered on a publicly available ATM, the chapter finishes with an evaluation how well the approaches scale in terms of runtime.

We conclude this thesis in Chapter 7 where we summarize the results of this thesis and give an outlook on future work.

2 Fundamentals

In this chapter we present the fundamental concepts we deal with in the remainder of this thesis. We first introduce probability density functions because they are used in some automaton models (cf. Section 2.1). Then we introduce different types of system classes where we mainly focus on discrete event systems (cf. Section 2.2). In Section 2.3, we present automaton models that are capable of modeling various system classes, followed by algorithms that are capable of inferring these automaton classes from data in Section 2.4. Finally, we present machine learning algorithms that can be used in one-class classification domains and, thus, are well suited for anomaly detection (cf. Section 2.5).

2.1 Probability Density Functions

Probability density functions (PDFs) are a common way to model probability distributions. PDFs should not be mixed up with probability functions. The values of a PDF may become greater than one which is not possible for probability functions. In PDFs, the probabilities are represented by the area under the PDF.

Given a *continuous* PDF f and a random variable X, the following equation holds:

$$Pr(X = a) = 0 \tag{2.1}$$

$$Pr(a \leq X \leq b) = \int_{a}^{b} f(x)dx \tag{2.2}$$

Equations (2.1) and (2.2) state that the PDF value for a single value is always zero, i.e. it cannot model the probability for a certain value (Eq. (2.1)). However, a PDF can give the probability of the random variable X being in the interval $[a; b]$ (Eq. (2.2)).

Additionally, it holds that the area under a PDF f must integrate to one:

$$\int_{-\infty}^{\infty} f(x)dx = 1 \tag{2.3}$$

Furthermore, the cumulative distribution function (CDF) $F_X(x)$ of a PDF f describes the probability that a random variable X will have outcome x or less.

$$F_X(x) = \int_{-\infty}^{x} f(x)dx \tag{2.4}$$

When given one-dimensional data, PDFs can be used to approximate the characteristics of the data. In the following we will present common types of PDFs.

Gaussian Distribution The Gaussian distribution (or Normal distribution) is modeled by two parameters: The mean μ and the standard deviation σ. The Gaussian distribution has its maximum at the mean and is symmetric around the mean. Furthermore, the density decreases the farther away from the mean the value is, but the value never becomes zero. The density function is defined as:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} \tag{2.5}$$

Uniform Distribution The uniform distribution is modeled by two parameters a and b with $b > a$ and models that all values between a and b have the same density value ($\frac{1}{b-a}$). Hence, the density function is defined as:

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{if } x \in [a; b] \\ 0 & \text{else} \end{cases} \tag{2.6}$$

Other Distributions There exist many other distributions with other parameters like exponential, χ^2, β distribution (and many more) that will not be explained in detail. Instead, we will focus on how complex PDFs can be estimated from given data. Hence, we present the kernel density estimation.

Kernel Density Estimation (KDE) Kernel Density Estimation (also called Parzen window) [Par62] is a method to estimate a PDF of unknown shape from given data points. As the name indicates a kernel K is used to estimate the PDF. Furthermore, a bandwidth parameter h is needed, that defines the smoothness of the resulting PDF. Opposed to the distributions above, the KDE is a non-parametric model because its shape is not only determined by a fixed set of parameters, but also by some training data (which is a kind of parameter, too)[1].

The idea of the kernel density estimation is to lay a kernel-defined distribution on every data point and to compute the density for an unknown point x by taking into account the distributions of the given data points around x. For data points $(x_1, \ldots, x_n)$, bandwidth h and kernel K the kernel density estimation f is defined as:

$$f(x) = \frac{1}{nh} \sum_{i=1}^{n} K(\frac{x - x_i}{h}) \tag{2.7}$$

Different kernels K exist, like the Gaussian, Cauchy, Picard kernel and others. In this thesis we mostly focus on the Gaussian kernel, which is defined as:

$$K(t) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}t^2} \tag{2.8}$$

[1] Note, that non-parametric models can also contain parameters, e.g. the bandwidth h. Parametric models only contain a fixed set of parameters and this set does not grow with increasing training data size.

If we use the Gaussian kernel in a kernel density estimator, Eq. (2.7) becomes:

$$f(x) = \frac{1}{nh} \sum_{i=1}^{n} \frac{1}{\sqrt{2\pi}} e^{-\frac{(x-x_i)^2}{2h^2}} \tag{2.9}$$

2.2 System classes

Systems theory is an established field of research that deals with describing systems by some characteristics and showing properties of the respective systems. A system can be seen as a black box with inputs and outputs (cf. Fig. 2.1). This blackbox can have certain properties which we will present in this section.

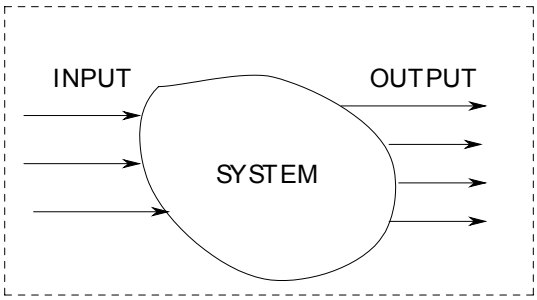


Figure 2.1: Abstract system view.

We can distinguish between *stateful* and *stateless* systems. As the latter rarely occur in reality because their expressiveness is limited, most technical systems are stateful. The stateful systems can be further distinguished into *discrete*, *continuous* or *hybrid* systems. Depending on whether the signals emitted by the system are discrete or continuous, the system satisfies the respective property. Hybrid systems are a mixture of discrete and continuous systems because they contain both discrete and continuous signals. Figure 2.2 shows a partial overview of different system classes according to [Cas07]. In the following we describe the different properties in the nodes of the tree.

Static vs. Dynamic Static (or state-/memoryless) systems produce an output that only depends on the current input and not on the input the system got before. The output of dynamic (or stateful/with memory) systems does not only depend on the current input, but also on the state of the system. This state originates from inputs the system has seen before.

Time-varying vs. Time-invariant Time-varying systems change their behavior with the time. This means that the system may change independently of the inputs. Hence, the output of the system does not only depend on the input and the system's state, but also on the point of time when the input was seen. Sometimes, this effect is also known as concept-drift. On the other hand, time-invariant systems always produce the same output if they are in a certain state and get a certain input.

Linear vs. Non-linear In linear systems the output of the system has a linear dependency to the input of the system. Hence, the system behavior can be modeled as a linear function from input and system state to the output. As systems can be of arbitrary complexity, this linearity is usually not present. Hence, for non-linear systems, the output does not need a linear dependency to the input and state.

Continuous-state vs. Discrete-state In continuous-state systems the representation of the state is continuous, hence the system has infinitely many states. Usually, the state is modeled by many real-valued variables. As it is very hard to model the behavior with real-valued numbers, discrete-state systems can be seen as an abstraction of continuous-state systems. In discrete-state systems the number of states is discrete and finite or countably infinite.

Time-driven vs. Event-driven Time-driven systems change their state after some time has elapsed. Usually, these systems obey some periodicity after which the state changes. Event-driven systems do not change their state after some time but only with the occurrence of events. The time gap between events may be arbitrary. As long as no event occurs the system stays in its current state.

Deterministic vs. Stochastic Deterministic systems always produce the same output in a certain state and for a given input. In stochastic systems the output of the system is described by some random variable(s). Hence, the output may be different for the same state and input, but not in an arbitrary manner but obeying the underlying random process.

Discrete-time vs. Continuous-time In discrete-time systems all system variables (input, state, output) are defined only at discrete points in time. Hence, the time can be modeled with the natural numbers. In continuous-time systems all system variables can be observed for any possible (continuous) point in time. However, in practice, the sensors to observe a system can only measure at a fixed frequency and therefore, they are discrete by their nature.

As we focus on discrete event systems in this thesis, we will shortly explain continuous systems and discrete systems thoroughly.

2.2.1 Continuous-state Systems

A continuous system model contains the input of various sensors where the sensor values stem from the real (physical) system. The system changes its state through actors that perform actions, but the actors themselves are not part of the system model. However, the actions of the actors may be observed with the sensors. Every sensor may have a different resolution and the system model acquires the sensor values with its own resolution. Thus, a continuous system consists of various time series of the respective sensors. One can think of such systems as analog systems where no preprocessing or abstraction (apart from the sensor resolution) is applied. Therefore, it may be difficult to learn behavior from the combination of the sensor values because changes in the system are continuous and the system has infinitely

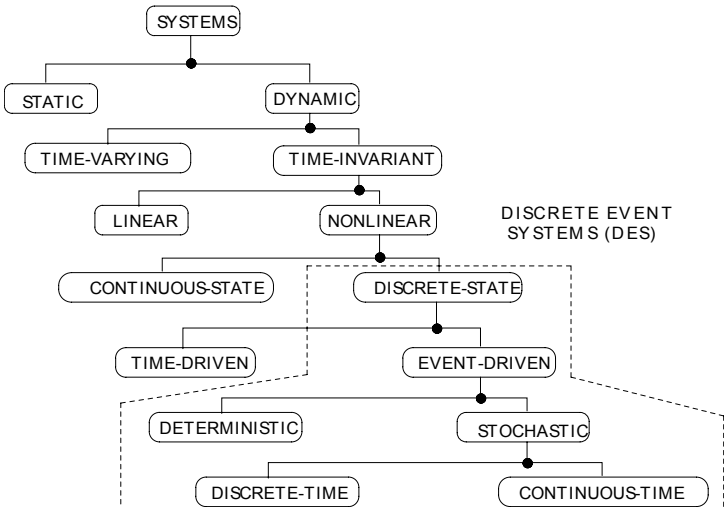


Figure 2.2: Overview of different system classes (according to [Cas07]).

many states. Furthermore, an abstraction of states may not be easy because an expert has to define when the system changes its abstract state. Hence, a system can be easily modeled as continuous system, but learning the model may be a hard task.

2.2.2 Discrete Event Systems

Discrete event systems (DES) are a limited, but still powerful abstraction to model certain systems. DES are often used to model technical systems where certain actors perform actions and change the state of the system with these actions. The state of the system may be observed by some discrete sensors. In [Cas07] DES are described to have three main properties:

1. *Discrete*: The number of possible states in the system is discrete (or finite).
2. *Dynamic*: The next state of the system depends on an observed event and the current state (but not on the states before).
3. *Event-driven*: State changes are caused by events only. The events occur at arbitrary points in time.

As shown in Fig. 2.2 DES can be further distinguished based on certain properties. In this thesis we will focus on discrete-time DES because most systems are not fully deterministic at a particular abstraction level. We only apply the discrete-time variant because at a very low level sensors measure only with some fixed frequency, thus the measurement rate is discrete.

Figure 2.3 shows a sample execution path of a DES with states q_i, events e_i and time points t_i. The *sample path* represents how the model is traversed and which states are visited. At the beginning the system is in state q_2. Then, event e_1 happens at timestamp t_1. The system changes its state to q_5 and stays in that state until event e_2 happens at timestamp t_2. This procedure continues as shown and at the end the system is in state q_6.

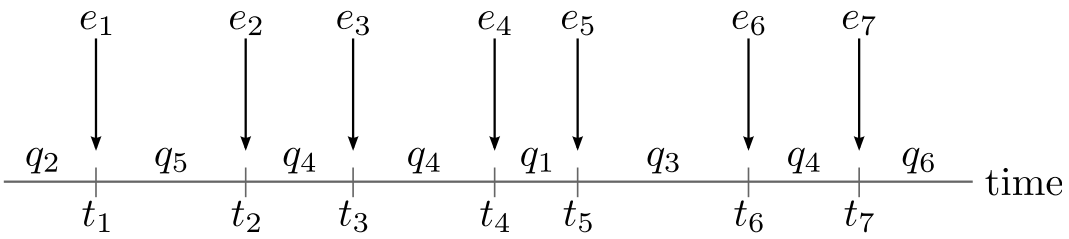


Figure 2.3: Sample path in an abstract DES for a given abstract sequence.

A simple example for a DES is the board game chess (cf. Fig. 2.4). Every board configuration corresponds to one state in the chess system. Every move of a chess piece corresponds to one event. A state change is only caused when a chess piece is moved and the possible moves (events) depend on the current board configuration (on the current state) and not on the previous moves (in which states the system was before). A change of the board configuration (state) only occurs if a chess piece is moved after some time by a player (an event occurs) and not by a (periodic) system behavior.

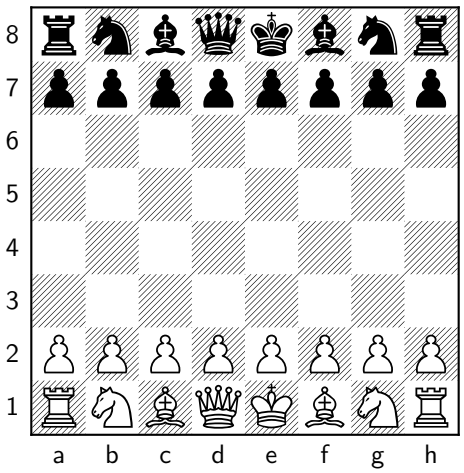


Figure 2.4: Initial state of a chess game.

One can think of a DES as a digital system where every sensor can only have finitely many values. The actors can still perform continuous actions, e.g. moving an object, but the sensors can only detect the object at certain (discrete) positions. On the one hand, we can regard a DES as a discrete abstraction from a real system that does not fully reflect the state of the underlying system. On the other hand, we often do not need the detailed state because it does not contain any information of additional value compared to the abstract state. Furthermore, it may be hard for a learning algorithm to extract insights from continuous states. Hence, the discretization of a DES may help learning algorithms to focus on the learning of the behavior of the system.

2.3 Automata classes

In this section we first present the basic concept of formal languages, followed by different automata classes starting with a simple Deterministic Finite Automaton (DFA) to the more complex Probabilistic Deterministic Real-Time Automaton (PDRTA) by adding probabilities, time information and transition guards based on time values. As non-deterministic automata are very hard to learn, they will not be part of this section.

2.3.1 Languages

Before we introduce automata classes we present the concept of formal languages in this section.

2.3.1.1 Untimed Languages

An untimed language $\mathcal{L}$ is a set that contains arbitrarily many words. A word is a sequence (or list) of symbols. Languages can be finite or infinite. A language $\mathcal{L}$ is usually defined over an alphabet Σ with each word w in the $\mathcal{L}$ (written $w \in \mathcal{L}$) containing only symbols of Σ. For example, the word $w_1 = abab$ is part of the language $\mathcal{L} = \{w \in \Sigma^* | w = (ab)^n, n \geq 0\}$. In [Cho56] Chomsky presents a hierarchy over formal languages, ranging from powerful languages (Chomsky type-0 called *recursively enumerable languages*) to less powerful but still mighty languages (Chomsky type-3 called *regular languages*). We only deal with languages of Chomsky type 3 (regular languages) because these languages can be expressed by finite automata and several algorithms exist for inferring finite automata from a given finite set of words. An abstraction of regular languages are the *stochastic* regular languages, which are recognized by probabilistic automata (cf. Section 2.3.2.2). They do not only contain the words w in the language $\mathcal{L}$ but also a probability density function over the all possible words Σ^*.

2.3.1.2 Timed Languages

Alur and Dill [AD94] present an overview over timed languages and timed automata (cf. Section 2.3.3). Timed languages contain timed words that contain symbols *and* time values. We refer to a slightly different notation of timed words [OW07] that better suits our formalism but is equivalent to the one presented in [AD94]. In [OW07] at first the definition of Alur and Dill is given but then they write a timed word as a sequence of *timed events*. Following this event notation a (finite) timed word consists of a word $w \in \Sigma^*$ consisting of symbols $w = e_1, \ldots, e_n$ and associated time values $\mathcal{T} = v_1, \ldots, v_n$. Hence, we have two ways of expressing a timed word w_t for symbols $w = e_1, \ldots, e_n$ and time values $\mathcal{T} = v_1, \ldots v_n$:

1. $w_t = (w, \mathcal{T}) = ((e_1, \ldots, e_n), (v_1, \ldots, v_n))$
2. $w_t = (e_1, v_1), \ldots, (e_n, v_n)$

We can now define a timed language in the same way as we did for untimed languages. In the following section we will first present untimed automata, followed by

timed automata. We will use the concept of timed languages for timed automata (Section 2.3.3) and for learning timed automata from a finite set of timed words (Section 2.4.3).

2.3.2 Untimed Automata

We first introduce untimed automata as a basic concept. Even though there are other automata classes we only present the ones necessary in this thesis, namely the Deterministic Finite Automaton (DFA), the Probabilistic Deterministic Finite Automaton (PDFA) as an extension of the DFA and the Non-Deterministic Finite Automaton (NFA) as the non-deterministic variant of the DFA.

2.3.2.1 Deterministic

A Deterministic Finite Automaton (DFA) [MP43] is the most simple automaton. Sometimes it is also called Finite State Machine (FSM). It is used in many applications and systems as it can describe the basic flow of a process. It starts in a start state, reads inputs and based on the inputs it proceeds to successor states until it reaches a final state or an input that is not valid. Formally, a DFA is defined in the following way:

Definition 1 (DFA)**.** *A Deterministic Finite Automaton is a five-tuple* $(\Sigma, Q, q_0, \delta, F)$ *with the following semantics:*

- Σ*: The finite, non-empty alphabet containing symbols.*
- Q*: The finite, non-empty set of states.*
- $q_0 \in Q$*: The start state.*
- $\delta : Q \times \Sigma \to Q$*: The state-transition function assigning a state and symbol the successor state (for some state-symbol combinations it may be undefined).*
- $F \subseteq Q$*: A set containing the final states (may be empty).*

Figure 2.5 shows a DFA with the states $\{q_0, q_1, q_2\}$ and the events $\{e_1, e_2, e_3\}$. q_0 is the start state and the only final state is q_2. Starting in q_0 and observing the event e_1, the automaton changes to q_1. In q_1 the automaton changes to state q_0 again if event e_2 is observed or to the final state q_2 if event e_3 is observed. The tuple representation of this DFA is:

$$A = (\{e_1, e_2, e_3\}, \{q_0, q_1, q_2\}, q_0, \{(q_0, e_1, q_1), (q_1, e_2, q_0), (q_1, e_3, q_2)\}, \{q_2\})$$

The language represented by A is $\mathcal{L}(A) = \{w \in \Sigma^* | w = e_1 u e_3; u = (e_2, e_1)^n; n \geq 0\}$

2.3.2.2 Probabilistic

Rabin invented the Probabilistic Deterministic Finite Automaton (PDFA) [Rab63] as an extension of a DFA (cf. Section 2.3.2.1), where probabilities are added to the transitions and the final states. Hence, a sequence as input cannot only be rejected

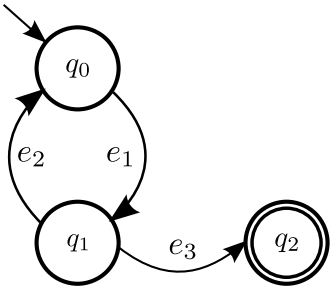


Figure 2.5: DFA with three states $\{q_0, q_1, q_2\}$ and three symbols $\{e_1, e_2, e_3\}$.

or accepted, but it is also possible to compute the likelihood of this sequence for the given PDFA, indicating how likely it is that the Probabilistic Deterministic Timed Automaton (PDTA) generated the sequence. Even though it seems odd, a PDFA is deterministic because it does not contain two transitions from one state q with the same symbol e.

Definition 2 (PDFA)**.** *A Probabilistic Deterministic Finite Automaton is a five-tuple* $(\Sigma, Q, q_0, \delta, \pi)$ *with* Σ, Q, q_0 *and* δ *from the DFA (Definition 1) and the set of final states* F *replaced by* π *that has two profiles.*

- $\pi_1 \colon Q \times \Sigma \to [0, 1]$*: The transition probability function assigning every transition a probability.*
- $\pi_2 \colon Q \to [0, 1]$*: The final state probability function assigning every state a probability.*

Sometimes, instead of introducing a second signature for the final state probability, an end symbol, e.g. # is introduced that does not belong to the alphabet and models the final state probabilities. Hence, the following expressions are equivalent for the end symbol #:

$$\pi_2(q) \mathrel{\hat{=}} \pi_1(q, \#) \tag{2.10}$$

If it is clear from the context we write π instead of π_1 or π_2.

Figure 2.6 shows a PDFA with the same events and states as the DFA in Fig. 2.5. For the transition probabilities we denote the probability with p at every transition. The probability function π contains the following elements:

$$\pi = \{(q_0, e_1, 1.0), (q_1, e_2, 0.7), (q_1, e_3, 0.3), (q_0, 0.0), (q_1, 0.0), (q_2, 1.0)\}$$

The transition probability function can be interpreted as how probable it is that a transition is taken while the final state probability function indicates the probability of a sequence ending in that current state. Furthermore, π is constrained that the transition and final probabilities in every state q must sum up to one:

$$\pi(q) + \sum_{e \in \Sigma} \pi(q, e) = 1 \tag{2.11}$$

Given a PDFA A it is possible to compute the probability of every sequence that is generated by A. The intuition is to aggregate the probabilities at every transition by

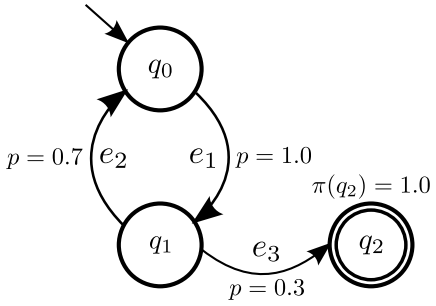


Figure 2.6: PDFA with three states (q_0, q_1, q_2), three symbols (e_1, e_2, e_3) and probabilities p.

multiplication and finally multiply the final probability in the current state. If there is no transition for some symbol, the probability of the sequence is zero. Hence, the probability $P_A(s)$ generating a sequence $s = e_1, \ldots, e_n$ from a PDFA A $(\Sigma, Q, q_0, \delta, \pi)$ is:

$$P_A(s) = \begin{cases} \prod_{i=1}^{n} \pi(q^i, e_i) \cdot \pi(q^n), & \text{if } (q^i, e_i) \in \delta \text{ with } q^{i+1} = \delta(q^i, e_i) \text{ and } q^1 = q_0 \\ 0, & \text{else} \end{cases} \tag{2.12}$$

2.3.2.3 Non-Deterministic

A Non-Deterministic Finite Automaton (NFA) [RS59] is the non-deterministic variant of the DFA. Opposed to the DFA, a combination of state and symbol can have more than one successor state in the NFA. Nevertheless, the NFA is language-equivalent to the DFA, hence they can model the same set of languages. Formally, an NFA is defined in the following way:

Definition 3 (NFA)**.** *A Non-Deterministic Finite Automaton is a five-tuple* $(\Sigma, Q, q_0, \delta, F)$ *with* Σ, Q, q_0 *and* F *from the DFA (Definition 1) and the transition function* δ *replaced by the following one:*

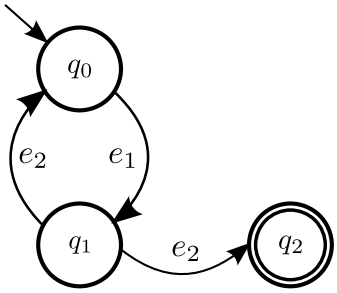
- $\delta\colon Q \times \Sigma \to \mathcal{P}(Q)$*: The state-transition function assigning a state and symbol arbitrarily many successor states.*

As for the DFA a probabilistic variant of the NFA exists—the PNFA. It adds probabilities to the automaton that sum up to one per state. Figure 2.7 shows an NFA (Fig. 2.7a) and its probabilistic variant (Fig. 2.7b) that is similar to the DFA shown above. Opposed to the shown DFA, the state q_1 has two outgoing transitions with symbol e_2, leading to states q_0 and q_2, respectively.

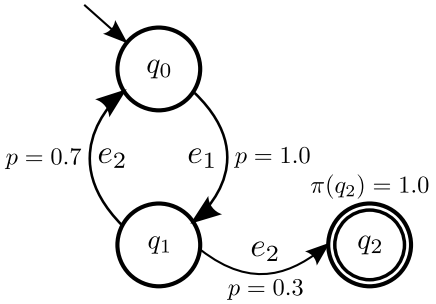
In the following we describe automata that not only incorporate the event structure but also the associated time values.

2.3.3 Timed Automata

Timed automata add timing information to automata. They were first described in [AD94] where they can contain many clocks on which the time (a real valued



(a) Non-Deterministic Finite Automaton (NFA).



(b) Probabilistic Non-Deterministic Automaton (PNFA).

Figure 2.7: (Probababilistic) Non-Deterministic Automaton ((P)NFA).

number) ticks when the automaton resides in some state. Then, a transition is taken based on the symbol *and* the time on a clock (expressed by guards on the transition). Transitions could also reset some clocks if they are taken. As we do not need the whole expressiveness of timed automata in this thesis, we will not explain general timed automata in detail, but instead a less expressive formalism that is sufficient for the content of this thesis.

A more strict version of timed automata is the One-Clock Deterministic Timed Automaton (1-DTA) [AD94] (sometimes also called Deterministic Real-time Automaton (DRTA)). It has only one clock that represents the time delay between two consecutive events and the guards on the transitions constrain the time delay on the clock. The clock is reset after any transition was taken, hence it counts the waiting time in a certain state. The time values must be taken from the natural numbers. In theory, this may be a limitation, but in practice, the time is always discrete because of discrete measurements.

Definition 4 (1-DTA)**.** *A One-Clock Deterministic Timed Automaton is a six-tuple* $(\Sigma, Q, q_0, \delta, F, T)$ *with* $\Sigma, Q,\ q_0, \delta$ *and* F *from the DFA and the time guards* $T : Q \times \Sigma \to \mathbb{N} \times \mathbb{N}$*.*

T constrains the time when a transition may be taken, e.g. $(q_0, a) \mapsto 4, 6$ means that if the automaton is in state q_0 and reads symbol a, then the time must be between 4 and 6 time units. The time values must be greater or equal than zero and the first value of a time guard must be smaller or equal to the second value.

To keep determinism of a 1-DTA, the successor state must be unique for any combination of symbol and time value. This determinism is achieved by constraining the time guards on the outgoing transitions for one state and symbol such that they do not overlap, i.e. $T(q_0, a) = [4, 6]$ and $T(q_0, a) = [5, 8]$ is forbidden, whereas $T(q_0, a) = [4, 6]$ and $T(q_0, a) = [7, 8]$ is allowed. Whereas the guards must not overlap for a single symbol, overlapping guards for different symbols are allowed, e.g. $T(q_0, a) = [1, 5]$ and $T(q_0, b) = [3, 7]$. In this way the following state can still be determined deterministically and thus, the whole automaton is still deterministic.

2.3.3.1 Probabilistic

Mainly two different probabilistic automaton models exist so far. They both belong to the class of *Stochastic Deterministic Timed Automaton* [Vod13]. In this section we present the PDTA [Mai14] and the *Probabilistic Deterministic Real-Time Automaton (PDRTA)* [VdW10]. Both automaton models are an extension of the 1-DTA but additionally contain probabilities for events. While a PDTA contains probabilities at the transitions, a PDRTA contains probabilities for an event in every state and additionally probabilities for a time value in every state. Thus, a PDRTA contains two probability distributions whereas a PDTA only contains one.

Definition 5 (PDTA)**.** *A Probabilistic Deterministic Timed Automaton is a seven-tuple* $(\Sigma, Q, q_0, \delta, F, T, \pi)$ *with* $\Sigma, Q, q_0, \delta, F$ *and* T *from the 1-DTA and the transition probability function* π *from the PDFA that models the probability of a transition.*

Figure 2.8 shows a PDTA with five states ($\{q_0, q_1, q_2, q_3, q_4\}$) and four events ($\{a, b, c, d\}$). For simplicity, there are no final state probabilities in states that have outgoing transitions (q_0, q_2). Every transition is labeled with an event, probability p and a time interval (or time guard).

As for the PDFA the probabilities of all outgoing transitions (including the final probability) for a state must sum up to one. Thus, π must take a third argument, namely the time guard. In [Mai14] the constraints on the probabilities for transitions are defined in a different way. The following formula for state q is given:

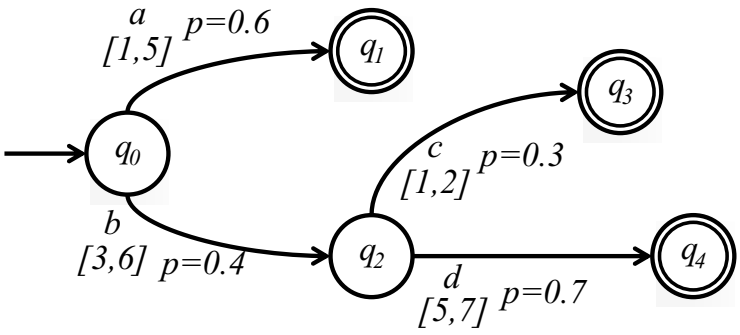


Figure 2.8: PDTA with five states, four events, transition probabilities and time guards (similar to [Mai14]).

$$\pi(q) + \sum_{q_i, q_j \in Q, e \in \Sigma} \pi(q_i, q_j, e) = 1 \tag{2.13}$$

Equation (2.13) should express that the probabilities of all transitions from state q to any successor state and the final state probability of q must sum up to one. But the formula does not cover the case that for a symbol e there may be more than one transition because there are different time guards for e. We present the correct formula (cf. Eq. (2.14)) in our notation that also comprises the different time guards in a state q.

$$\pi(q) + \sum_{e \in \Sigma} \sum_{t=(q,e,*,*) \in T} \pi(q, e, t) = 1 \tag{2.14}$$

Even though a PDTA cannot be directly transformed into a PDRTA, the PDRTA is an extension of the PDTA to some extent. Only the modeling of the event probabilities is slightly different, but the rest of the two automata classes is the same. As the PDRTA contains information about the probability of time values, it is more expressive than a PDTA. In [VdW10] a PDRTA is defined in the following way:

Definition 6 (PDRTA). *A Probabilistic Deterministic Real-Time Automaton is a four-tuple* $(\mathcal{A}', H, \mathcal{S}, \mathcal{T})$ *with* $\mathcal{A}'$ *a 1-DTA,* H *a finite set of time intervals in the form* $[v, v']$*, called histograms,* $\mathcal{S}$ *a finite set of event probability distributions* $\mathcal{S}_q = \{Pr(S = e|q) \colon e \in \Sigma, q \in Q\}$ *and* $\mathcal{T}$ *a finite set of time-bin probability distributions* $\mathcal{T}_q = \{Pr(T \in h|q) : h \in H, q \in Q\}$.

Figure 2.9 shows a PDRTA with two states ($\{q_0, q_1\}$), two events ($\{a, b\}$) and four histogram bins ($\{[0, 1], [2, 3], [4, 5], [6, 10]\}$). The symbol probabilities are shown next to the corresponding state, e.g. for state q_0 the probability of observing the event a is the same as observing b (0.5). The probabilities for the time bins in state q_0 are shown on the left, the probabilities for the time bins in state q_1 on the right.

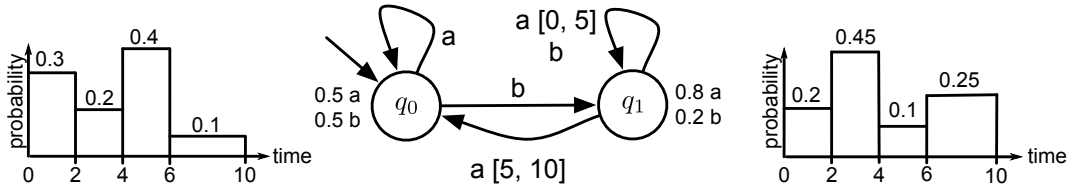


Figure 2.9: PDRTA with two states, two events and four time buckets (similar to [VdW10]).

As the time is modeled with histograms and there is no further information how the time values are distributed in the histograms, a uniform distribution of the time values is assumed in every histogram. Hence, the probability for a certain time value t in a state q is:

$$Pr(T = t|q) = \frac{Pr(T \in h|q)}{v' - v + 1} \tag{2.15}$$

where $h = [v; v'] \in H$ such that $t \in [v; v']$. The event e and the time value t are modeled conditionally independent given the state q. Hence, the probability of observing an event-time tuple (e, t) in a state q is the product of the probability of the event in q and the time value in q:

$$Pr((e, t)|q) = Pr(S = e|q) \cdot Pr(T = t|q) \tag{2.16}$$

Then, the probability of a timed sequence of event-time tuples is the product of the probabilities of the tuples of the sequence. Please note that the probabilities of symbols and time values do not depend on which transition was taken, but on which symbol and time value are observed in which state. Hence, the symbol probability depends on the state q and the symbol e and the time probability also depends on q and the time value t.

2.3.4 Hybrid Automata

In addition to the mentioned models there also exist hybrid models for every type of model. Hybrid automata additionally model continuous signals from continuous inputs. Every state contains a function θ modeling the continuous signal in the respective state. The hybrid automaton is most often called Probabilistic Deterministic Hybrid Timed Automaton (PHyTA) [Mai14] or Stochastic Deterministic Hybrid Automaton (SDHA) [Vod13]. In discrete event systems continuous signals do not exist and thus, hybrid models are not explained in more detail. For more information see Section 3.4.3.2.

Summary

In this section we introduced automaton models ranging from simple DFAs without time information to the more complex PDRTAs. As there are different names for very similar automaton model, Table 2.1 gives an overview on some of the different names of automata and their occurrences. In the remainder of this thesis, we will mostly focus on the PDFA, PDTA and PDRTA.

Probabilistic Deterministic Finite Automaton (PDFA) [Rab63] Stochastic Deterministic Finite Automaton (SDFA) [Vod13]
Probabilistic Deterministic Real-Time Automaton (PDRTA) [VdW10] Probabilistic Deterministic Timed Automaton (PDTA) [Mai14] Stochastic Deterministic Timed Automaton (SDTA) [Vod13]

Table 2.1: Different notations for automata classes

2.4 Automata Inference Algorithms

In this section algorithms that learn or infer automata are presented. The algorithms learn different types of automata, depending on the input and the algorithm itself. Before presenting the algorithms we describe some common basics that are used in most of the algorithms.

2.4.1 Common Elements

In classical machine learning tasks the input data consists of instances of equal length that contain data for arbitrary attributes. An instance entry v_i in any instance $\mathbf{x}$ contains the value for the attribute a_i. In this thesis we mostly deal with sequence data that cannot be expressed as instances of equal length because sequences can have arbitrarily varying length. We distinguish between timed and untimed sequences.

2.4.1.1 Notation

An untimed sequence s_e is a succession of events e_i from an alphabet Σ. The number of occurrences of an event is not limited, so it may occur more than once or not at

all. The important aspects of the sequence are the occurring events and their order. For an alphabet $\Sigma = \{a, b, c\}$ the untimed sequences $s_{e_1} = acb$ and $s_{e_2} = aabba$ are two examples of length three and five, respectively.

A timed sequence s_t not only contains events but also time values related to every event. The events are contained in an alphabet and the time values stem from $\mathbb{N}_0$. As for an untimed sequence, the length is arbitrary and not known in advance. The order and the occurring events are still important, but additionally the time value associated with each event is important. We will write a *timed* sequence in the form $s_t = (e_1, v_1), (e_2, v_2), \ldots, (e_n, v_n)$ with e_i representing the symbol at position i and v_i representing the associated time value. For an alphabet $\Sigma = \{a, b, c\}$ the timed sequence $s_{t_3} = (a, 2), (c, 1), (b, 20)$ contains three events (a, c, b) and their associated time values $(2, 1, 20)$. The timed sequence s_{t_3} contains the untimed sequence $s_{e_1} = abc$ but is enriched with time values. The length of s_{t_3} is still three (as for s_{e_1}). To denote a set of timed sequences, we write S_t. A set of sequences without time information is denoted as S_e. The size of a (timed) sequence set is defined as the number of sequences it contains. As we deal with sequences that are usually generated by an underlying model M, we define positive (or normal) sequences:

Definition 7 (Normal Sequence)**.** *A sequence s^+ is normal (or positive) with respect to a set of sequences S which are generated by an underlying model M if s^+ stems from M.*

The respective positive input sequence sets are denoted as S_t^+ and S_e^+. Sometimes we deal with sequences that stem from a different model:

Definition 8 (Abnormal Sequence)**.** *A sequence s^- is abnormal (or negative) with respect to a set of sequences S which are generated by an underlying model M if s^- does not stem from M.*

The respective negative input sequence sets are denoted as S_t^- and S_e^-. If a set contains both negative and positive sequences it is denoted as $S_t^{+/-}$ and $S_e^{+/-}$, respectively.

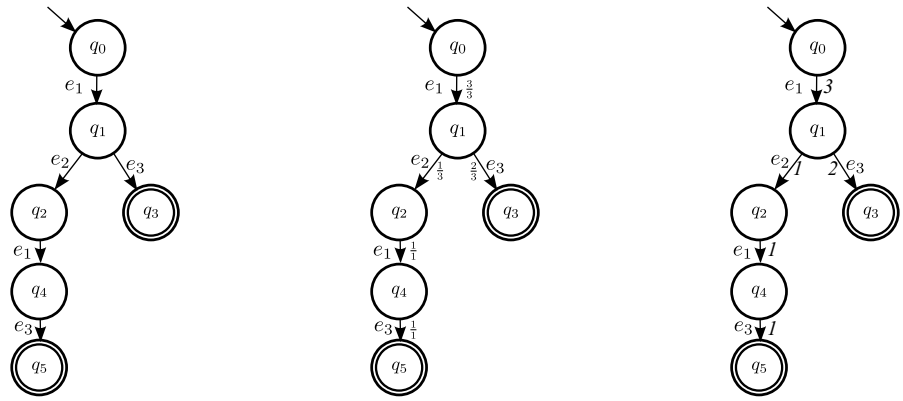
2.4.1.2 Probabilistic Prefix Tree Acceptor

Most algorithms that process (un-)timed sequences first build a *Probabilistic Prefix Tree Acceptor (PPTA)* in an initial step. It is a tree that accepts sequences and contains probabilities for every transition. The PPTA is similar to the *Frequency Prefix Tree Acceptor (FTA)* that contains frequencies instead of probabilities. Sometimes a PPTA is called a *Prefix Tree Acceptor (PTA)* even though it is still probabilistic. In the following we will also adapt to this notation and use PTA if we mean PPTA or FTA[2].

Definition 9 (PPTA)**.** *A Probabilistic Prefix Tree Acceptor (also Prefix Tree Acceptor) is a special kind of a PDFA (cf. Definition 2). It is a tree with the start state being the root node, has a unique path from any state to any other state and does not contain cycles. Apart from that it is a PDFA.*

[2]FTAs can be easily converted into PPTAs and most often, a PPTA is stored as FTA internally.

A non-probabilistic *prefix tree acceptor* is a special prefix tree for accepting event sequences. It is a tree representation for a set of untimed event sequences and contains all information about the event order, but lacks the information how often events and sequences occur. A PPTA contains this frequency information implicitly because it contains the probabilities of events and sequences. Figure 2.10 shows a PTA, a PPTA and an FTA with root node q_0 and $\Sigma = \{e_1, e_2, e_3\}$. The PTA (Fig. 2.10a), PPTA (Fig. 2.10b) and FTA (Fig. 2.10c) accept the same sequences (e_1, e_4 and e_1, e_2, e_1, e_3) but the PTA lacks the probability or frequency information at the transitions.



(a) Prefix Tree Acceptor (PTA).

(b) Probabilistic Prefix Tree Acceptor (PPTA).

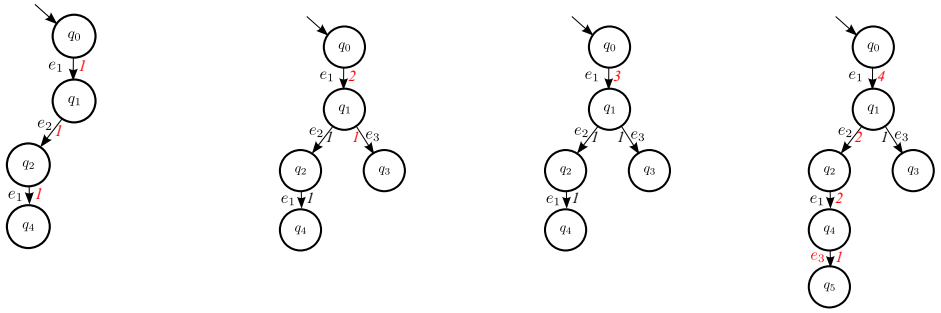
(c) Frequency Prefix Tree Acceptor (FTA).

Figure 2.10: Differences between PTA, PPTA and FTA.

A PTA is created from a set of event sequences S_e^+ that only contains positive examples. It is a perfect representation for S_e^+, thus only accepting sequences from S_e^+ and rejecting all other sequences. The initial PTA only contains the start node q_0. For every sequence in S_e^+ the current PTA is traversed as long as possible until the sequence ends or there is no possibility to traverse an edge. For every transition taken the occurrence counter is increased by one and if a node is reached at the end of the sequence the stopping counter of the current state is increased by one. If there is no transition for a symbol the rest of the sequence is added as a subtree with occurrence count of one for every transition and a stopping counter of zero for every state except the last state—its stopping counter is initialized with one.

Figure 2.11 illustrates the process of an FTA creation for the input set $S_e^+ = \{e_1e_2e_1, e_1e_3, e_1, e_1e_2e_1e_3\}$. The sequences are added one after the other. Changes in the frequency counter at the transitions are marked red. If no transition exists for a symbol in a given state, the transition is created and points to a newly created state. Figure 2.11a shows the initialization with the first sequence $e_1e_2e_1$. After adding the sequences e_1e_3 and e_1, Fig. 2.11d shows the addition of the last sequence $e_1e_2e_1e_3$ which is also the final FTA. At the end, the probabilities can be calculated from the occurrence and stopping counters. In some algorithms the counters instead of the probabilities are needed, thus the probabilities are not always calculated for a PTA, but only on-the-fly.

A PTA can be constructed in linear time because every sequence is processed just



(a) Initialization with sequence $e_1e_2e_1$. (b) Adding sequence e_1e_3. (c) Adding sequence e_1. (d) Adding sequence $e_1e_2e_1e_3$.

Figure 2.11: FTA generation for sequences $S_e^+ = \{e_1e_2e_1, e_1e_3, e_1, e_1e_2e_1e_3\}$.

once and no further (non-linear) computation is done afterwards. Thus, the runtime is $\mathcal{O}(n)$ with $n = |S_e^+|$ the number of sequences in S_e^+. However, not only the number of sequences n matters but also their length. In worst case—every sequence occurs once—the PTA cannot generalize anything and thus, creates a new state for every symbol (even if the symbol was seen before in another sequence). Let l_{max} be the length of the longest sequence, then the number of states for a PTA is $\mathcal{O}(|S_e^+| \cdot l_{max})$. So on a more granular level the runtime for creating a PTA is polynomial in the length of the longest sequence and size of the sequence set S_e^+.

2.4.1.3 Hoeffding Bound

As some of the presented algorithms use the Hoeffding bound, we will not describe it for every algorithm but instead only once. Hoeffding's inequality [Hoe63] is a concentration inequality that provides a bound on how strong the mean deviates from the expected value. From this inequality, a test can be derived to check whether an empirical outcome f significantly differs from a probability value p with a confidence parameter α and n observations:

$$\left| p - \frac{f}{n} \right| < \sqrt{\frac{1}{2n} \log \frac{2}{\alpha}} \tag{2.17}$$

As we compare two empirical outcomes f, f' (the frequencies of state visits) when merging automata we apply the Hoeffding bound with n, n' many observations for two outcomes and check whether they are significantly different with respect to α:

$$\left| \frac{f}{n} - \frac{f'}{n'} \right| < \sqrt{\frac{1}{2n} \log \frac{2}{\alpha}} + \sqrt{\frac{1}{2n'} \log \frac{2}{\alpha}} \tag{2.18}$$

$$\Leftrightarrow \left| \frac{f}{n} - \frac{f'}{n'} \right| < \sqrt{\frac{1}{2} \log \frac{2}{\alpha}} \cdot \left(\frac{1}{\sqrt{n}} + \frac{1}{\sqrt{n'}} \right) \tag{2.19}$$

After introducing common elements we continue with the algorithms that mostly use PTAs in an initial step and then merge states in the PTA to achieve a more compact representation.

2.4.2 Learning Untimed Automata

In this section we present algorithms that learn Probabilistic Deterministic Finite Automata (PDFA, cf. Definition 2) from a positive set of event sequences S_e^+. The algorithms just employ event sequences without any time information. Despite the input set S_e^+ they may require other parameters which are then described in the respective subsection. There exist various algorithms that also learn PDFAs but we focus on ALERGIA and Minimal Divergence Inference (MDI) as they are used later in this thesis because of their comprehensibility and simplicity.

2.4.2.1 ALERGIA

ALERGIA [CO94] by Carrasco was one of the first algorithms where a state merging technique was applied. The algorithm consists of two steps and needs an additional confidence parameter α. In a first step a preefix tree acceptor (PTA) is constructed from a set of input sequences. In a second step the states of the PTA are recursively merged based on the Hoeffding bound with α as criterion. Given the PTA, for every pair of two states (more specific: a pair of a state and one of the predecessors of this state) the algorithm recursively checks whether these two states are compatible. Two states are regarded as compatible if the number of arriving and ending sequences in the states are equal according to the Hoeffding bound, the number of arriving sequences and the number of outgoing transitions for every symbol are also equal according to the Hoeffding bound and if the two criteria also hold for the successors of the respective states. Let n and n' be the number of arriving sequences at the respective state and f and f' the number of ending or outgoing sequences for a symbol. The Hoeffding bound is used for testing whether two estimates are significantly different (cf. Eq. (2.19)).

If two states are found to be compatible, the ingoing and outgoing transitions are bent to or away from the first state. This bending may create non-determinism for the first state because there may have already been an outgoing transition for some symbol x and the second state may have also had an outgoing transition with symbol x. In this case, the two target states that are reached with symbol x are also merged in the same manner. Algorithm 1 shows the pseudo code of ALERGIA. At first, the PTA is initialized from the input set S_e^+ (line 1). Then, two nested loops iterate over all state pairs (lines 2 and 3) and check for compatibility of the states (line 4). If two states are found to be compatible, they are merged (line 5) and the resulting automaton is made deterministic again (line 6). A merge of two states q_i and q_j is performed by bending all incoming transitions of q_i and q_j into a newly created state q_k while keeping the source states of the transitions. The same is performed for all outgoing transitions of q_i and q_j such that those transitions leave q_k while keeping their destination states.

Runtime Complexity ALERGIA runs in $\mathcal{O}(n^3)$ [CO94] with n the number of states in the PTA constructed from S_e^+. Comparing every pair of two states takes $\mathcal{O}(n^2)$ time, every such comparison takes at most $\mathcal{O}(n)$ time (recursively checking the successor states for compatibility) and the comparison is possibly followed by a recursive merge step that also takes $\mathcal{O}(n)$, so we end up with $\mathcal{O}(n^2 \cdot (n+n)) = \mathcal{O}(n^3)$.

ALGORITHM 1: ALERGIA

Data: Positive set of (untimed) event sequences S_e^+, significance parameter α
Result: PDFA A

```
1  A' ⟵ PTA(S_e^+);
2  for i = 1 to n − 1 do
3      for j = 0 to i − 1 do
4          if compatible(q_i, q_j, α) then
5              A' ⟵ A'.merge(q_i, q_j);
6              A'.determinize();
7              break j-loop;
```

ALGORITHM 2: ALERGIA-compatible

Data: States q_i, q_j, significance parameter α
Result: true/false

```
1  n ⟵ arriving(q_i);
2  n' ⟵ arriving(q_j);
3  f ⟵ ending(q_i);
4  f' ⟵ ending(q_j);
5  if HoeffdingBound(n, n', f, f', α) then
6      return false;
7  foreach e ∈ Σ do
8      out ⟵ outgoing(q_i);
9      out' ⟵ outgoing(q_j);
10     if HoeffdingBound(n, n', out, out', α) then
11         return false;
12     if not compatible(δ(q_i, e), δ(q_j, e), α) then
13         return false;
14 return true;
```

In general, the less the PTA can generalize the input sample and the less ALERGIA finds compatible states and merges them, the longer the (real) runtime of ALERGIA. In [CO94] the experimental runtime is found to be $\mathcal{O}(n)$ but the experiments are biased. The input sample S_e^+ is very big but the resulting PTA is quite small. Hence, most of the time is needed for constructing the PTA and the merging of a small PTA is done quite fast.

A drawback of ALERGIA is that it only checks whether two states are compatible by looking at the two states and their successors. It is not checked whether a merge improves the PDFA with respect to the training set. Hence, Thollard et al. proposed the MDI which is presented next.

2.4.2.2 Minimal Divergence Inference (MDI)

The Minimal Divergence Inference [TDH00] was proposed to outperform ALERGIA because ALERGIA only has a local view on two states and their predecessors but not a global view on how the automaton generalizes from the training set. Instead of applying a local merge criterion, the *Kullback-Leibler* (KL) divergence is used to compare two automata before and after a merge step to decide whether the performed merge step was useful. Given two automata A and A' the KL divergence $D(A||A')$ is defined as:

$$D(A||A') = \sum_{s_e \in \Sigma^*} P_A(s_e) \cdot \log \frac{P_A(s_e)}{P'_A(s_e)} \tag{2.20}$$

with $P_A(s_e)$ as defined in Section 2.3.2.2. As it is computationally expensive to compute the KL divergence for every merge in this way it can be incrementally computed. Let A_1 be the automaton obtained by merging a state in the initial PTA A_0 and let A_2 be an automaton that results from a possible state merge in A_1. The divergent increment Δ going from A_1 to A_2 then is:

$$\Delta(A_1, A_2) = D(A_0||A_2) - D(A_0||A_1) \tag{2.21}$$

The computation of $\Delta(\cdot,\cdot)$ still involves the computation of $D(\cdot||\cdot)$ twice, thus requiring a lot of computation. With a trick, the difference $\Delta(A_0, A_2)$ between the initial PTA A_0 and any automaton A_2 can be computed only considering the intermediate automaton A_1 and the states that were merged when going from A_0 to A_2:

$$\Delta(A_0, A_2) = D(A_0||A_1) + \sum_{q_i \in Q_{012}} \sum_{e \in \Sigma \cup \{\#\}} c_i \cdot \pi_0(q_i, e) \cdot \log \frac{\pi_1(q_{\underline{i}}, e)}{\pi_2(q_{\underline{i}}, e)} \tag{2.22}$$

Q_{012} is the set of states in A_0 which have been merged to obtain A_2 from A_1, c_i is the probability of reaching q_i from the root node, and π_i is the transition probability function in A_i and $q_{\underline{i}}$ is the representative state that remains when merging q_i and some other state. With dynamic programming the divergence values can be stored and have to be computed only once for every pair and a lot of recomputation is not necessary.

The decision whether to keep a merge when obtaining A_2 from A_1 is considered compatible with the training data if the divergence increment relative to the size reduction (i.e. decrease in number of states) is small enough. The number of states of the automaton is denoted as $|\cdot|$. Formally, the merge is performed if the following inequality holds for some compatibility threshold α:

$$\frac{\Delta(A_1, A_2)}{|A_1| - |A_2|} < \alpha \tag{2.23}$$

Algorithm 3 shows the pseudo code of MDI. As for ALERGIA all states are inspected (lines 2 and 3), but in contrast to ALERGIA the merge is always executed (line 4). Then the resulting automaton is determinized (non-determinism is removed; line 5) and the old and the new automaton are compared for compatibility using Eq. (2.23) (line 6). If the new automaton is a better representation, it is kept and

the old automaton is discarded (line 7).

ALGORITHM 3: MDI

Data: Positive set of (untimed) event sequences S_e^+, significance parameter α
Result: PDFA A

```
1 A ⟵ PTA(S_e^+);
2 for i = 0 to n − 1 do
3     for j = 0 to i − 1 do
4         A' ⟵ A.merge(q_i, q_j);
5         A'.determinize();
6         if compatible(A, A', α) then
7             A ⟵ A';
8             break j-loop;
```

Runtime In [TDH00] Thollard et al. state that "the overall complexity of this algorithm, evaluated as the number of state pairs which are considered for merging, is $O(N^2)$ where N denotes the size of the PPTA". Although this statement is true, it can be misleading. One could think that the runtime is $\mathcal{O}(n^2)$ with n the size of the PPTA, but this is not true. The runtime is "evaluated as the number of state pairs which are considered for merging". Every such consideration for merging takes at most again $\mathcal{O}(|\Sigma| \cdot n)$ steps with n the number of states in the PTA because every consideration must compute Eq. (2.22). This computation consists of iterating over the alphabet Σ and the set of states that changed from one automaton to the other Q_{012} that may be in $\mathcal{O}(n)$. Hence, the overall runtime of MDI is $\mathcal{O}(n^3 \cdot |\Sigma|)$ with n the number of states of the PTA.

2.4.2.3 Other Algorithms

Apart from MDI and ALERGIA there exist several other algorithms that infer a PDFA from a positive set of untimed event sequences S_e^+. As MDI and ALERGIA are two of the more famous ones they are presented in detail while others are not. Nevertheless, a short excerpt of other inference approaches is given in Section 3.3.3.1.

2.4.3 Learning Timed Automata

In this section we present algorithms that learn Probabilistic Deterministic Timed Automata (PDTA, cf. Definition 6) from a positive set of timed event sequences S_t^+. Compared with the algorithms presented so far, the following algorithms employ timed event sequences, thus process time information. Despite the input set S_t^+ they may require other parameters which are then described in the respective subsection. To the best of our knowledge only the two algorithms presented here (Real-Time Identification from Positive Data (RTI+) and Bottom-Up Timed Learning Algorithm (BUTLA)) learn a PDRTA from a positive set of timed sequences.

2.4.3.1 BUTLA

The Bottom Up Timed Learning Algorithm (BUTLA) was proposed by Maier in [Mai14]. In the literature there exists another version of BUTLA that includes a split operation (cf. [Mai+11]), but we will describe the most recent version presented in [Mai14]. The main idea of BUTLA is first to globally detect discrepancies in time values for any event, then to split this event into two or more new events and finally apply a bottom up state merging step utilizing the Hoeffding bound.

BUTLA is a state merging algorithm that consists of four steps:

1. Global preprocessing of time values for every event
2. PTA-creation based on the preprocessing
3. Merging states from the created PTA
4. Adding time guards based on the mean time values of every event

In the following we will describe every step in detail.

Preprocessing In the preprocessing step the time values for every symbol are gathered globally. Then for every symbol and all the gathered time values a kernel density estimator with Gaussian kernel is computed. Instead of a fixed bandwidth h, a variable smoothing factor of 5% of every gathered value on the density estimator is used. The authors claim that this smoothing is well suited for the identification of normal behavior of production plants (cf. [Mai14]). Hence, the modified kernel density estimation (h exchanged with $0.05 \cdot x_i$) with a Gaussian kernel and time values $(x_1, \ldots, x_n)$ is[3]:

$$KDE(t) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{\sqrt{2\pi} \cdot 0.05 \cdot x_i} \cdot e^{-\frac{(t-x_i)^2}{2 \cdot (0.05 \cdot x_i)^2}} \tag{2.24}$$

In the next step the local minima of the KDE are calculated. It is assumed that a *mode* is between two minima. Every such mode represents a new symbol and for every mode a new symbol is added to the alphabet. The point where a minimum is located is used as a strict border for a symbol even though the underlying distribution is assumed to be Gaussian. Figure 2.12 shows the kernel density estimation of time values that occurred for a single symbol e. The minima in the density estimation are located (M_1, M_2) and three new symbols e_1, e_2, e_3 are created that represent the respective modes. The symbol e is transformed into e_1 if the time value of e lies between zero and the first minimum M_1. For e_2 the time values of e have to be between M_1 and M_3, and for e_3 the time value has to be greater than M_2.

PTA-creation The PTA is created in the same way as described in Section 2.4.1.2.

[3] In [Mai14] the formula is presented with an undefined variable x and the denominator in the exponent is not squared. We try to correct the formula and interpret it as presented here. Another possible interpretation (with h^2 exchanged with $0.05 \cdot x_i$) would be:
$KDE(t) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{\sqrt{2\pi \cdot 0.05 \cdot x_i}} \cdot e^{-\frac{(t-x_i)^2}{2 \cdot 0.05 \cdot x_i}}$

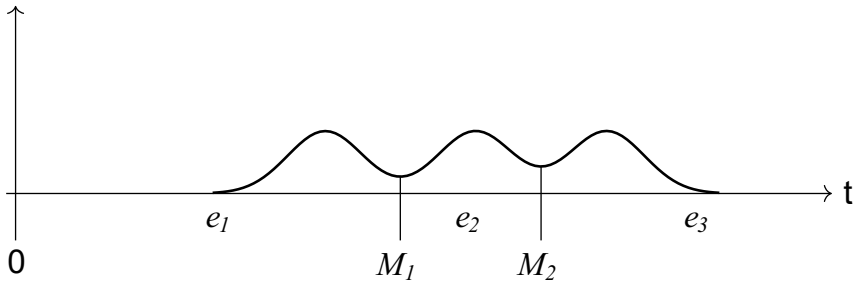


Figure 2.12: BUTLA event split for event e into three subevents e_1, e_2, e_3 (similar to [Pap16]).

State merging The states of the PTA are merged based on the Hoeffding bound (cf. Section 2.4.1.3). The merge criterion is similar to ALERGIA (cf. Section 2.4.2.1) but instead of comparing the number of outgoing transitions the number of incoming transitions are used (cf. Algorithm 4). The other difference to ALERGIA is that the loops that iterate over the states are reversed such that the BUTLA operates in a bottom-up order instead of top-down. The result is from the same model class as ALERGIAs result: a PDFA.

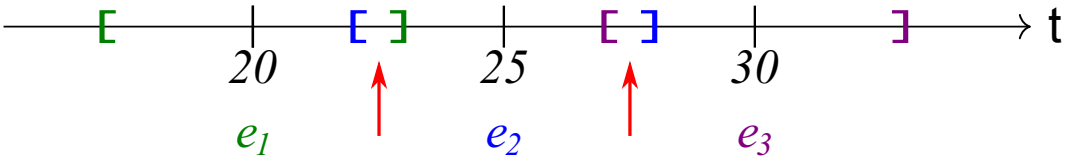
ALGORITHM 4: BUTLA-compatible

Data: States q_i, q_j, significance parameter α
Result: true/false

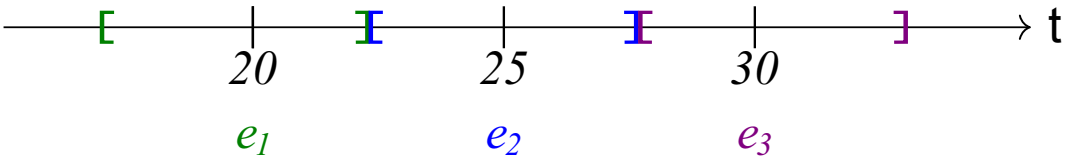
```
n ⟵ arriving(q_i);
n' ⟵ arriving(q_j);
f ⟵ ending(q_i);
f' ⟵ ending(q_j);
if HoeffdingBound(n, n', f, f', α) then
    return false;
foreach e ∈ Σ do
    in ⟵ incoming(q_i);
    in' ⟵ incoming(q_j);
    if HoeffdingBound(n, n', in, in', α) then
        return false;
    if not compatible(δ(q_i, e), δ(q_j, e), α) then
        return false;
return true;
```

Time guards After a PDFA is learned with BUTLA, the time guards have to be added to the PDFA in order to obtain a PDRTA. BUTLA adds time guards based on the mean μ and the standard deviation σ of the time values for a given symbol. Hence, for every symbol, the time guard is defined as the interval $[\mu - k \cdot \sigma; \mu + k \cdot \sigma]$ with $k \in \mathbb{R}^+$. Assuming a symbol was split into several symbols in the preprocessing,

and based on the choice of k, intervals for the same symbol may overlap. As overlaps are forbidden by the definition of a PDTA we assume that BUTLA sets the border of the two intervals to be the point where the symbol was split into two symbols in the preprocessing step. Figure 2.13 shows the time guards for a symbol e that has been



(a) PDTA with incorrect (overlapping) intervals.



(b) PDTA with correct (non-overlapping) intervals.

Figure 2.13: Incorrect and correct interval generation for a PDTA with BUTLA.

split into e_1, e_2 and e_3. Assuming that the standard deviation σ for e_1, e_2 and e_3 are the same (which is not necessarily the case) with $k = 1$ and $\mu_1 = 20, \mu_2 = 25, \mu_3 = 30$ and $\sigma = 6$ the time guards for e would overlap as shown in Fig. 2.13a. The red arrows point at the overlapping time intervals. We assume that the correct PDTA would look as shown in Fig. 2.13b—by setting the border of the time guards to the point where the event was split in the preprocessing.

Runtime Complexity As BUTLA is similar to ALERGIA with additional preprocessing, a slightly modified compatible function and a bottom up merging order, the merge step of BUTLA has the same runtime as ALERGIA. Thus, the merge step of BUTLA runs in $\mathcal{O}(n^3)$ with n the number of states in the constructed PTA.

In Lemma 1 in [Mai14] the author states that BUTLA runs in $\mathcal{O}(n_0^3)$ with n_0 the number of states in the constructed PTA. We state that the given runtime does not completely cover the runtime of BUTLA. The author references Proposition 8 which gives the runtime complexity for the preprocessing step. This preprocessing is not analyzed in detail. The author states that the minima of Eq. (2.24) are found in $\mathcal{O}(n)$ but he does not prove it. Finding the minima of a function like Eq. (2.24) is a non-trivial problem and may be harder than $\mathcal{O}(n)$.

When analyzing the runtime of BUTLA, the author correctly states the merge step runtime to be $\mathcal{O}(n_0^3)$, but the author even misuses his own proposition. According to Proposition 8 the preprocessing runtime is $\mathcal{O}(n)$ with n the number of *observations*. As the number of observations n may be much higher than the number of states n_0 in the PTA, the term $\mathcal{O}(n)$ cannot be neglected.

Hence, if Proposition 8 in [Mai14] holds, the runtime complexity of BUTLA is $\mathcal{O}(n_0^3 + n)$ with n_0 the number of states of the PTA and n the number of observations. As the runtime of the preprocessing was not thoroughly analyzed in Proposition 8,

the runtime of BUTLA is $\mathcal{O}(n_0^3 + \text{Prep}(n))$ with $\text{Prep}(n)$ the runtime complexity for the preprocessing step.

2.4.3.2 RTI+

The RTI+ algorithm was the first algorithm that inferred a PDRTA from positive data. It is an adaptation of the RTI algorithm (cf. Section 3.3.3.1) to be trained only with positive data.

As most of the presented algorithms, RTI+ builds a PTA and merges this PTA. While merging states, not only the symbols and their occurrences are considered, but also the time values. Therefore, RTI+ uses an *Evidence-Driven State-Merging* (EDSM) approach to merge states based on their symbol and time properties. To consider time values, the time information must be encoded in the prefix tree acceptor. Hence, RTI+ uses a timed prefix tree acceptor, which is a PTA with time guards (that are initialized with the maximum time values, and refined later). Opposed to the other algorithms presented before, RTI+ does not only apply a merge but also a split operation. The split operation is the opposite of a merge operation, thus it does not generalize but refines the model. Instead of merging two states (and their successors) into one new state, the split operation divides one state into two states and also creates new successor trees.

RTI+ applies a red-blue framework. In this framework a state is red, blue or white. Red states are confirmed states, blue states are the (non-red) successors of red states and white states are all other states. While the algorithm runs, more states will be confirmed (i.e. more blue states become red), then more white states become blue and at the end, all states are red.

RTI+ proceeds in the following way: At the beginning, all states except the root state are white. In a loop, RTI+ iterates as long as there are non-red states:

- Color the successor states of all red states blue.
- Pick the most visited transition $\delta = (q_r, q_b, e, g)$ from a red state (q_r) to a blue state (q_b).
- Evaluate all possible splits of δ. Perform the most significant split, if it exists.
- Evaluate all possible merges of the blue state q_b with all red states. Perform the most significant merge, if it exists.
- If no split or merge has been performed, color q_b red.

If two states are merged, there is no choice how the merge is performed. In the merged model the two states are combined and their subautomata are also combined.

If a transition is split there are several possibilities where to split the transition. More precisely, the transition can be split between any two time values that are in between the lower and upper bound of the time guard. Let $[n; n']$ be the time guard of the transition to split. Then the split can be performed between n and $n+1$ or $n'-1$ and n' or all other time values that are between n' and n. Hence, there are $n' - n$ many possibilities of a split, that have to be executed and evaluated.

For the evaluation of splits and merges, RTI+ applies the *likelihood ratio test*.

Likelihood Ratio Test The likelihood ratio test computes the likelihood ratio (LR) of the model before (M) and after (M') an operation (split or merge) with respect to the training data S^+ and checks for significance of the null hypothesis using the degrees of freedoms of the models by applying the χ^2 test.

$$LR = \frac{likelihood(S^+, M)}{likelihood(S^+, M')} \tag{2.25}$$

The likelihood for the whole training data as a set of timed sequences can be computed by multiplying the likelihoods for every sequence for a given PDRTA (cf. Eq. (2.16)). This ratio is compared to the parameters (or degrees of freedom) using the χ^2 test. This test indicates whether a change in the automaton *significantly* improves the model, hence it outputs a p value.

The degrees of freedom of a PDRTA are the number of choices in every inner state (in a state without any outgoing transitions there is no choice). They are defined as the number of symbols one could observe in a state q and the number of time bins (or histograms) a time value could fall into. Thus, for an inner state q we have $|\Sigma| - 1$ choices for the symbols and $|H| - 1$ choices for the time values, resulting in $(|\Sigma| - 1) \cdot (|H| - 1)$ choices for an inner state q. For a PDRTA with k inner states, we get $k \cdot (|\Sigma| - 1) \cdot (|H| - 1)$ degrees of freedom. When comparing two models M and M' we only need to take into account the difference in the degrees of freedom. Let n and n' be the degrees of freedom of the PDRTAs M and M'. The likelihood ratio test determines whether it is a good idea to spend more degrees of freedom for a significantly better model. Formally, we introduce function F which provides the p value for the χ^2 with $n' - n$ degrees of freedom:

$$F_{\chi^2(n'-n)}(x) = 1 - CDF(\chi^2(n' - n), x) \tag{2.26}$$

The result of F with $-2 \cdot \ln(LR)$ is the p value (or significance) that the operation was good. Hence, the following terms are compared:

$$F_{\chi^2(n'-n)}(-2 \cdot \ln(LR)) \lesseqgtr \alpha \tag{2.27}$$

Depending on whether a merge or split operation was performed, α must be smaller or greater than the p value. As a merge decreases and a split increases the degrees of freedom, the comparison with α must be handled in two different ways. If a merge is tested the merge is significantly good if the p value is greater than α. For a split the p value must be smaller than α.

The authors introduced an additional technique called *pooling* for handling transitions with small frequencies. Transitions with small frequencies contain a lot of unused parameters and especially near leaf nodes the test for significance cannot be performed well. The idea of the pooling is to combine the frequencies of time bins and symbols if their frequencies are low and treat them as a single symbol or time occurrence. Hence, the number of parameters is reduced. For further details we refer to [VdW10].

According to [VdW10] the runtime complexity of RTI+ is polynomial. Even though it is not stated in which term the runtime is expressed, we assume that the runtime is polynomial in the size of the input S^+ because the likelihood ratio test includes

repeated iterating over the input data. A tighter runtime bound is not given. The convergence of RTI+ is only conjectured.

2.4.4 Algorithm Analysis Frameworks

In this section we review two important algorithm analysis frameworks for machine learning, namely *learning in the limit* and *PAC learning*. The frameworks demand different solution qualities of the learned model and when a model is considered to have converged, among others.

2.4.4.1 Learning in the Limit

Learning in the limit (in the case of automata learning also *language identification in the limit*) is a framework first presented in [Gol67]. In this framework the learning algorithm (or learner) may observe infinitely many examples (without having to ask for them). After each seen example the learner has to provide a new model (automaton). A learner identifies a language in the limit if it converges to the correct identification of the language (i.e. the automaton representing this language) after a finite number of examples. Nevertheless, the learner itself does not know when this convergence is reached. In [Gol78] Gold showed that it is impossible to identify a DFA in the limit from positive samples (i.e. from a regular language) only.

2.4.4.2 PAC Learning

Probably approximately correct learning (PAC learning) was proposed by Valiant in [Val84]. In contrast to the learning in the limit, the PAC framework helps analyzing the solution quality of the model even if the learner does not receive infinitely many samples. The PAC framework also analyzes how many samples are needed to find a "good" model with high probability.

As the learning task usually becomes harder with less samples, the learner does not need to learn the exact underlying model (automaton): Instead, with high probability ("probably") the learner will find a good model ("approximately") that is not too far away from the underlying (true) model. Usually, two parameters δ and ε are used to describe that with probability $1 - \delta$ the learner finds a model that has an error of ε or less compared to the underlying model if the learner received a number of samples polynomial in $\frac{1}{\varepsilon}$ and $\frac{1}{\delta}$. Furthermore, the framework is called polynomial PAC learning if the runtime of the learner has to be polynomial in the number of examples received.

2.5 One-class Classification

A classification problem describes the process of assigning a class label for a given instance. Usually, a classification model (or classifier) is built in a training phase, where labeled examples are presented. Based on the data presented in the training phase, the classifier usually tries to find borders that separate the classes. In a test phase, one ore more unlabeled example are presented and the classifier shall decide which label to assign to the given example(s).

We distinguish between two-class (binary), multi-class and one-class problems. In the common two-class problem, only two labels (most often 0 and 1) are allowed and examples with both labels are presented in the training phase. In the multi-class classification setting, we do not only deal with two, but more than two labels (e.g. $0, 1, 2, \dots$), that are also presented in the training phase. When testing the classifier it labels every unknown test example with one of the classes it has seen during the training phase (0 or 1 for a two-class problem, 0,1,2,... for a multi-class problem). For one-class problems, labeled examples (in the training phase) are only available for one class (usually denoted as the positive class). While testing, the classifier can only denote whether the given example belongs to the positive class or not (meaning it belongs to some other class).

In classification tasks we usually deal with an input consisting of instances of fixed length (every example is a such an instance). As for normal (and abnormal) sequences (cf. Definitions 7 and 8), we can define normal (and abnormal) instances with respect to the underlying model in the following way:

Definition 10 (Normal Instance)**.** *An instance* $\mathbf{x}^+$ *is normal (or positive) with respect to a set of instances* $\mathbf{X}$ *which are generated by an underlying model* M *if* $\mathbf{x}^+$ *stems from* M*.*

Definition 11 (Abnormal Instance)**.** *An instance* $\mathbf{x}^-$ *is abnormal (or negative) with respect to a set of instances* $\mathbf{X}$ *which are generated by an underlying model* M *if* $\mathbf{x}^-$ *does not stem from* M*.*

In the following we will present algorithms that can be applied to solve one-class classification problems. Some of these algorithms are adaptations of two-class algorithms.

2.5.1 One-class Classification Algorithms

In this section we present the basic concepts of algorithms that are applied in the remainder of this thesis—namely the support vector machine (SVM) and clustering.

2.5.1.1 Support Vector Machine

The support vector machine (SVM) [BGV92] is a supervised binary classifier that constructs one or more hyperplanes to separate the data points into two classes. In its basic implementation (called linear SVM), the SVM can only separate data that can be separated by a hyperplane into two classes. The hyperplane can be written as:

$$\mathbf{w}^T x + b = 0 \tag{2.28}$$

with $\mathbf{w}$ the normal vector and some bias b. Depending on where a point lies relative to the hyperplane it is assigned to one of the two classes. As infinitely many hyperplanes exist between two linearly separable classes, the SVM tries to find that hyperplane which maximizes the distance (or margin) to the nearest points of both classes. Figure 2.14 shows two linearly separable classes and two possible hyperplanes with hyperplane A having a higher margin to both classes than hyperplane B.

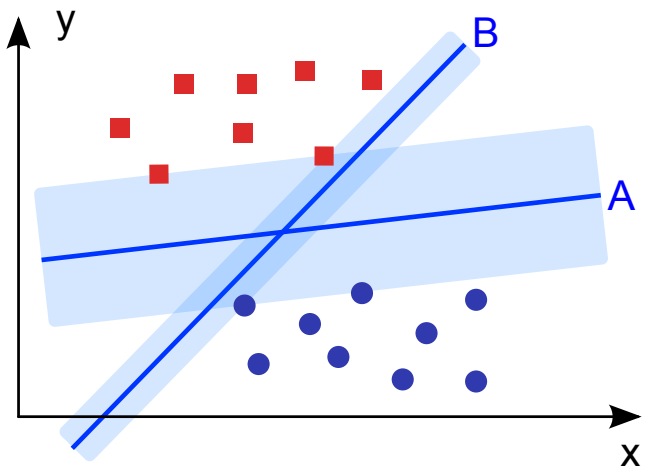


Figure 2.14: Two possible hyperplane separations for a linearly separable data set (Figure from `https://de.wikipedia.org/wiki/Support_Vector_Machine` (Ennepetaler86, CC BY 3.0, `http://creativecommons.org/licenses/by/3.0/`)).

If linear separability is assumed, the so called *hard-margin SVM* is used which computes the hyperplane and does not allow any points to lie on the wrong side of the hyperplane. If the data is not linearly separable the idea is to allow a wrong classification for some points. Hence, slack variables $\xi_i > 0$ are introduced that allow some of the data points to lie within the margin. Additionally, a parameter C controls the trade-off between maximizing the margin and allowing training points within the margin. The SVM with slack variables ξ_i and parameter C is called *soft-margin SVM* because the hyperplane softly separates the classes (allowing some classification mistakes). As the hard-margin SVM is a special case of the soft-margin SVM with $C = \infty$, we only present the formulas for the soft-margin SVM. Let the data set be of the form $\{(\mathbf{x}_i, y_i) | i = 1, \ldots, m; y_i \in \{-1, 1\}\}$ with $\mathbf{x}_i$ being the input instance and y_i the according label. To compute the separating hyperplane the following optimization problem has to be solved with $\mathbf{w}$ the normal vector as in Eq. (2.28):

$$\begin{aligned} &\text{minimize } \frac{||\mathbf{w}||^2}{2} + C \sum_{i=1}^{n} \xi_i \\ &\text{subject to: } \quad y_i(\langle \mathbf{w}, \mathbf{x_i} \rangle - b) \geq 1 \text{ for all } i \end{aligned} \tag{2.29}$$

This minimization problem can be solved using quadratic programming. Instead of the primal, we can also solve the dual problem which usually requires less calculations when making predictions with the SVM. Therefore, we first formulate the normal vector $\mathbf{w}$ as linear combination of the training samples:

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x_i} \tag{2.30}$$

Then, we can reformulate the problem using Lagrange multipliers and the following

equation has to be maximized:

$$\text{maximize} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \mathbf{x_i}, \mathbf{x_j} \rangle \tag{2.31}$$

$$\text{subject to:} \quad \sum_{i=1}^{n} \alpha_i \cdot y_i = 0, \text{ and } 0 \leq \alpha_i \leq C \text{ for all } i$$

Moreover, $\alpha_i \neq 0$ exactly when $\mathbf{x}_i$ lies on the hyperplane or in the margin ($\xi_i > 0$).

For the classification of a new point it is checked on which side of the hyperplane the point is located. This is done according to the following equation:

$$f(\mathbf{x}) = \text{sign}(\langle \mathbf{w}, \mathbf{x} \rangle + b) \tag{2.32}$$

$$= \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i \langle \mathbf{x_i}, \mathbf{x} \rangle + b \right) \tag{2.33}$$

Every α_i is a weight in the classification function of the SVM. Usually, most $\alpha_i = 0$, hence those $\mathbf{x}_i$ are not necessary for the computation of f. Only some α_i and their x_i have to be stored for the classification function. These $\mathbf{x_i}$ are called *support vectors*.

Kernel Trick The soft- and hard-margin SVM only find a good representation of the data if the data is more or less linearly separable. For linearly non-separable data sets, a feature mapping function ϕ is introduced that transforms the input instances into a higher dimension:

$$\phi \colon \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}, \mathbf{x} \mapsto \phi(\mathbf{x}) \text{ with } d_2 > d_1 \tag{2.34}$$

In this higher dimensional space it is often possible to linearly separate the data points that were not linearly separable before. Implementing the feature mapping ϕ into the SVM is straight forward. Every dot product of the input instances $\langle \mathbf{x_i}, \mathbf{x_j} \rangle$ is replaced by the dot product of the transformed input instances $\langle \phi(\mathbf{x_i}), \phi(\mathbf{x_j}) \rangle$. But the calculation of this product is very expensive.

To circumvent this costly calculation, we apply the kernel trick. Therefore, we utilize Mercer's theorem [Mer09], thus we use some kernel function K that satisfies Mercer's condition (the kernel matrix has to be positive semi-definite) and is easier to calculate than the dot product in $\mathbb{R}^{d_2}$. Instead of explicitly computing the feature mapping ϕ and the dot product, we can compute the kernel function $K(\mathbf{x_i}, \mathbf{x_j})$ which is equivalent to the dot product in $\mathbb{R}^{d_2}$:

$$K(\mathbf{x}, \mathbf{x_i}) = \langle \phi(\mathbf{x}), \phi(\mathbf{x_i}) \rangle \tag{2.35}$$

Then, the feature mapping ϕ and the dot product can be replaced by the kernel function K for every calculation of the SVM. This replacement must be done for the optimization step and for the classification function. For example, the classification

function (Eq. (2.33)) becomes:

$$f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i K(\mathbf{x_i}, \mathbf{x}) + b\right) \tag{2.36}$$

A popular example is the *RBF-kernel*, which is is defined as follows (with σ a free parameter):

$$K(\mathbf{x_i}, \mathbf{x_j}) = \exp\left(\frac{||\mathbf{x_i} - \mathbf{x_j}||^2}{2\sigma^2}\right) \tag{2.37}$$

One-class SVM A special variant of the SVM can also be used for one-class classification [Sch+99]. Instead of having input instances containing the labels $\{-1, +1\}$ they only contain the label $+1$. Figure 2.15 illustrates the idea of the one-class SVM. All input instances are separated from the origin and the distance of

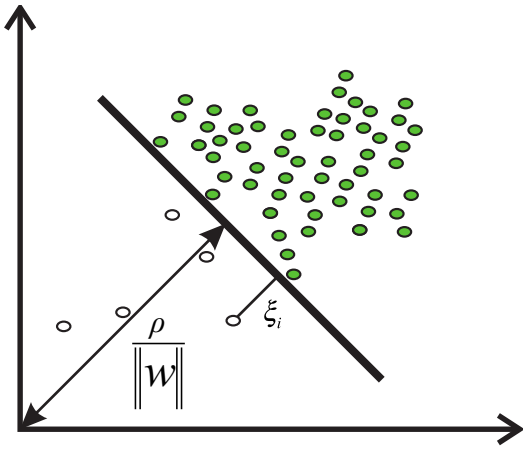


Figure 2.15: One-class SVM illustration (similar to [San+15]).

the created hyperplane to the origin shall be maximized. The slack variables ξ_i allow for some errors as in the binary SVM. The classification function returns $+1$ only for that region where most training examples are located, thus the density of already seen examples is high.

The minimization problem (cf. Eq. (2.28)) is reformulated by using ν instead of C and ρ instead of b (acc. to [Sch+99]; with ϕ the feature mapping function as in Eq. (2.34)):

$$\begin{aligned} &\text{minimize } \frac{||\mathbf{w}||^2}{2} + \frac{1}{\nu n}\sum_{i=1}^{n}\xi_i - \rho \\ &\text{subject to:} \\ &y_i(\langle \mathbf{w}, \phi(\mathbf{x_i})\rangle - b) \geq \rho - \xi_i \text{ for all } i \end{aligned} \tag{2.38}$$

ν is also known as the target rejection rate and plays an important role. It represents an upper bound on the fraction of training samples that are treated as outliers and it is a lower bound on the number of training examples that are used as

support vectors ($\alpha_i \neq 0$ in the dual optimization problem). Although the ν-SVM is still applicable to the binary setting, the one-class SVM is also known as ν-SVM because of the importance of ν.

The classification function for the ν-SVM is also similar to the binary SVM (Eq. (2.36)):

$$f(\mathbf{x}) = \text{sign}\left(\langle \mathbf{w}, \phi(\mathbf{x}) \rangle - \rho\right) \tag{2.39}$$

$$= \text{sign}\left(\sum_{i=1}^{n} \alpha_i K(\mathbf{x_i}, \mathbf{x}) - \rho\right) \tag{2.40}$$

2.5.1.2 Clustering

Clustering is an unsupervised learning technique that forms clusters that consist of similar samples. All the samples belonging to the same cluster are similar based on the distance metric that has to be defined (e.g. Euclidean distance). As clustering is an unsupervised method, a class label for samples is not needed, hence it can also be used as one-class classifier. In this thesis we present two approaches how to transform a clustering algorithm into a one-class classifier. These approaches may be applicable to many other clustering algorithms, but we only show this for the two algorithms presented: k-Means [Llo82] and DBSCAN [Est+96].

The first approach is to cluster the instances from the positive class (training set) and additionally add the unknown test sample. If the test sample is a noise point (or anything similar), it is classified as an outlier. This approach is applicable to all clustering algorithms that label instances as core or noise points, e.g. DBSCAN.

For the second approach the clustering is performed once with all instances from the positive class only. As additional hyperparameter a distance threshold t has to be set. For every unknown test instance the distance to the nearest characteristic cluster point is computed. If this distance is greater than the predefined threshold t, the test instance is classified as outlier. A characteristic point may be the cluster center or a non-noise point (depending on the chosen clustering algorithm). For k-means (among others) this approach is described in [KMD10].

Now we present the two clustering algorithms—DBSCAN and k-Means—that can be transformed into a one-class classification algorithm using one of the mentioned approaches.

DBSCAN DBSCAN [Est+96] is a density based clustering algorithm. The idea is to check for every point how many neighboring points are in its neighborhood. If there are enough points, the point is classified as *core* point. If there are not enough points, the point can be a *border* point if it has at least one core point in its neighborhood. If there are too few points and no core point in the neighborhood, the point is a *noise* point. A cluster is formed by assigning the same cluster number to all core and border points that are in each others (transitive) neighborhood.

Two parameters must be specified: ε and n, where ε defines the neighborhood as the distance or radius in which neighboring points are searched, and n defines the size of the neighborhood, thus the minimal number of points in distance not greater than ε.

Figure 2.16 shows the classification of points when applying DBSCAN with $n = 3$. There are 3 points in the ε-neighborhood of point A, hence A is a core point. The same holds for the other red points which are also core points. Point B and C are border points because their distance to a core point is at most ε. Point N is a noise point because less than 3 points and no core point is in distance of ε.

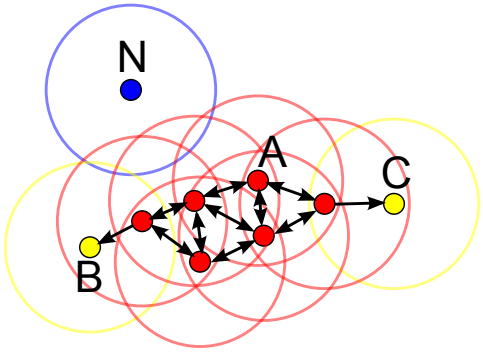


Figure 2.16: Point classification in DBSCAN with $n = 3$ (Figure from `https://en.wikipedia.org/wiki/DBSCAN` (by Chire, CC BY-SA 3.0, `http://creativecommons.org/licenses/by-sa/3.0/`)).

Figure 2.17 shows the outcome of applying DBSCAN on a two dimensional data set. The data set is split into two clusters (blue and red) with noise points in gray. As DBSCAN is a density based clustering algorithm, the form of the clusters is influenced by the density of the points.

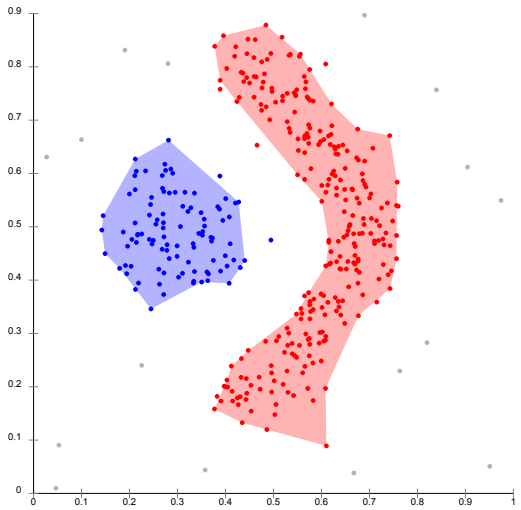


Figure 2.17: Example outcome of DBSCAN with two clusters (Figure from `https://en.wikipedia.org/wiki/DBSCAN` (by Chire, CC BY-SA 3.0, `http://creativecommons.org/licenses/by-sa/3.0/`)).

Properties The runtime complexity of DBSCAN is $\mathcal{O}(n^2)$ in its naive implementation with n the number of points. With the help of indexing structures, the runtime complexity becomes $\mathcal{O}(n \cdot \log n)$. The order in which the points are processed does

not heavily influence the result of DBSCAN. Due to a different processing ordering, only the border points may be assigned to different clusters and the cluster numbers may change.

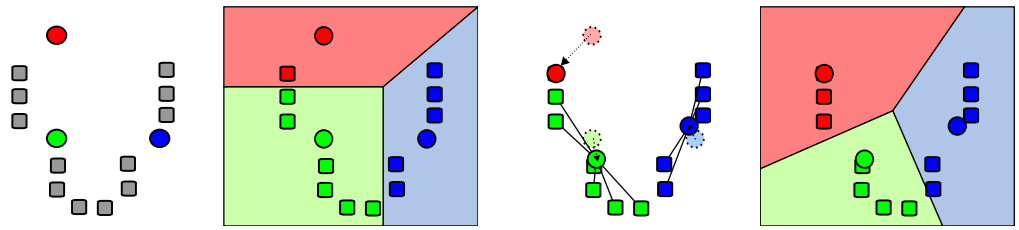
k-Means k-Means [Llo82] is a partitioning algorithm that divides a given data set into k partitions. Opposed to DBSCAN the number of clusters k must be known in advance. k-Means tries to minimize the overall squared distance of points to their respective cluster center, formally:

$$\text{minimize} \sum_{i=1}^{k} \sum_{\mathbf{x}_j \in S_i} \|\mathbf{x}_j - \boldsymbol{\mu}_i\|^2 \tag{2.41}$$

with $\mathbf{x}_j$ the points, S_i the cluster assignments and $\boldsymbol{\mu}_i$ the centers of the clusters. k-Means starts with k mean points (cluster centers) by some initialization strategy. Then it iteratively performs the following two steps.

1. Assign every point to the cluster with the nearest cluster center.
2. Recompute the mean (cluster center) for every cluster as the mean of all points assigned to this cluster.

Figure 2.18 shows the clustering of a small data set (the squares represent data points) with k-Means and $k = 3$. The initial centers (colored circles) are chosen randomly (cf. Fig. 2.18a). Then the assignment of points to the cluster centers is performed in Fig. 2.18b (squares are colored based on the assigned cluster). Based on this assignment the means of the clusters are recomputed in Fig. 2.18c, hence, the centers are "moved". After some iterations of assignments and recomputations, the final result of the algorithm with the respective assignments and centers is shown in Fig. 2.18d.



(a) Initialize k-Means with 3 centers. (b) Assign the points to the nearest cluster. (c) Recompute the mean of every cluster. (d) Result after some iterations.

Figure 2.18: Sample execution of k-Means with 3 clusters (Figure from `https://en.wikipedia.org/wiki/K-means_clustering` (by Weston.pace, CC BY-SA 3.0, `http://creativecommons.org/licenses/by-sa/3.0/`)).

Properties k-Means is very sensitive to the initialization of the cluster centers and may become stuck in local optima. Depending on how the cluster centers are initially chosen, the result can notably differ. Furthermore, the worst case runtime

complexity is exponential in the number of points n (for only two dimensions and certain initializations) [Vat11]. Hence, a lot of variants have been proposed to circumvent some of these drawbacks.

Variants As k-Means is a very popular clustering algorithm, many variants exist. Instead of computing means as centers, the median can be computed (k-Medians), the cluster centers can be initialized in a more elegant way than randomly (k-Means++), the data points may belong to a cluster with some degree (Fuzzy C-Means), etc. If the number of clusters k is unknown the elbow heuristic is a common method to iteratively increase k and check whether data is modeled much better (with respect to some metric) with the higher k or whether the model quality increases only marginally. If the data is not modeled better the previous k should be chosen. In this thesis we apply two special variants of k-Means (G-Means and X-Means) which also deal with "guessing" the number of clusters.

G-means and X-means G-Means [HE04] and X-Means [PM00] are extensions of k-Means without the necessity to specify the number of clusters k. Even though both work different in some sense, they both tackle the same problem of not knowing the number of clusters in advance by assuming some distribution of the data.

X-Means was the first adaptation of k-Means to guess the number of clusters. It utilizes the Bayesian Information Criterion (BIC) [KW95] or a similar criterion (e.g. Akaike information criterion (AIC) [Aka98]). The criterion indicates how well the model fits to the data while penalizing a large number of parameters because a more complex model will always fit the data better than a simple one. To obtain the correct number of clusters X-Means starts with an initial clustering with k-Means and a small k. Then X-Means locally splits every cluster into two new clusters and runs k-Means locally for the two new clusters. If this split improved the information criterion of the local clusters (thus it fits the data better and does not increase the model complexity too much), there is evidence that there really exist two (or more) clusters, locally. Hence, in case of improvement the split is accepted, otherwise rejected. Then a new iteration is performed until all further splits are rejected.

G-Means is another extension of k-Means that guesses the number of clusters. G-Means outperforms X-Means because the BIC in X-Means does "not penalize strongly enough the model's complexity" [HE04]. While X-Means applies an information criterion, G-Means assumes a subset of the data follows a Gaussian distribution. Therefore, G-Means also splits every cluster center into two centers and performs a local k-Means clustering. Instead of using an information criterion G-Means performs a statistical test for each cluster center based on the Anderson-Darling statistic [AD52]. The test checks whether the data around the center follows a Gaussian distribution or whether the split into two centers was significantly better. If the data is Gaussian (according to the test), the old center is kept, otherwise it is replaced by the two new centers.

Even though it was shown that G-Means outperforms X-Means on some data sets, we will apply both algorithms in Chapter 5 because in most cases the results depend on the data set the algorithm was applied on.

2.5.2 Anomaly Detection

Anomaly detection is usually defined as a one-class problem where the anomalies are not known in advance. For some special problems the anomalies may be known, but in general one cannot describe all possible anomalies but instead only model the normal behavior. Hence, the anomaly detection problem can most often be formulated as a one-class problem. Sometimes it is also referred to as outlier detection.

In Chapter 3 we describe approaches that deal with anomaly detection for fixed-length instances. In the following we review the first anomaly detection algorithm for timed sequences.

2.5.2.1 Anomaly Detection for Timed Automata

To the best of our knowledge only one algorithm for detecting sequence anomalies with timed automata exists. Most automata learning algorithms were invented in the grammatical inference community. This community mainly focuses on inferring automata but not on detecting anomalies using the inferred automata. The ANOmaly Detection Algorithm (ANODA) [Mai14] is an anomaly detection algorithm only for PDTAs.

ANODA is a very naive algorithm that checks for anomalies only considering the existence of transitions. It operates in the following way: ANODA requires a PDTA and a timed test sequence as input. The PDTA is traversed starting in the initial state q_0. Then every time-event tuple from the timed sequence is read and it is checked whether a transition fits to the current time-event tuple. If so, ANODA proceeds to the next time-event tuple. If such a transition does not exist, ANODA outputs "unknown event" or "wrong timing" depending on whether the event was unknown or the timing did not fit to a transition.

In this way, ANODA can detect simple anomalies that did not occur in the training data of the learned model. On the other hand, if only one abnormal sequence was part of the training data and is not removed in the model learning step (which does not happen for BUTLA), this abnormal sequence would be labeled as normal by ANODA because a path in the automaton model exists for this sequence. Hence, ANODA is not robust to outliers that (somehow) became part of the automaton model.

In Section 4.4.2, we present a new anomaly detection algorithm called Automata-based Anomaly Detection Algorithm (AmAnDA) that solves the same anomaly detection problem for timed event sequences as ANODA does. Opposed to ANODA, AmAnDA can handle different automata types (PDTAs, PDRTAs and Probabilistic Deterministic Timed Transition Automatons (PDTTAs)) and does not only take into account the existence of a transition but also the probabilities of all time-event tuples in the test sequence.

In this chapter, we described the fundamentals of this thesis—probability density functions, different system classes, automata classes to represent system classes, learning algorithms for automata classes and one-class classification algorithms. In the next chapter, we review related work and describe different research areas that partly tackle the sequence-based anomaly detection problem.

3
Related Work

In this chapter, we review related work and relate different research areas that all deal with sequence-based anomaly detection. Sequence-based anomaly detection has been studied and applied in several research areas. However, the main focus of these research areas differs, the knowledge transfer between them is small, and other notations and algorithms are used in the respective areas.

This chapter gives an overview of the different research areas—namely *Process Mining*, *Grammatical Inference* and *Sequence-based Anomaly Detection*. Their notations, similarities, differences, and algorithms are presented in Sections 3.2 to 3.4. For every research area, we present which type of data is usually processed (Sections 3.2.1, 3.3.1 and 3.4.1), how anomaly detection is performed (Sections 3.2.2, 3.3.2 and 3.4.2) and which models and algorithms are used (Sections 3.2.3, 3.3.3 and 3.4.3). In Section 3.5, we present anomaly detection approaches that do not operate on sequences, but solve the anomaly detection task e.g. on vectors of fixed length.

3.1 Research Area Overview

The problem of sequence-based anomaly detection is mostly tackled in the research areas Process Mining, Grammatical Inference and Sequence-based Anomaly Detection. However, the main focus of the research areas is different and so is the way how they solve the anomaly detection problem. Before we present the details of the respective research areas in Sections 3.2 to 3.4, we outline to what extent the research areas solve the anomaly detection problem for timed sequences.

Process Mining focuses on extracting process models from given process logs and analyzing and interpreting these models. The process logs may not only contain events and time values but also other information. The time values are used for predicting when a process will be finished or for analyzing bottlenecks in the process flow. The anomaly detection task is solved using *conformance checking* approaches that check whether a process log and a predefined model are still conformal. To the best of our knowledge, the conformance checking algorithms do not consider time values, but only check for the order of events.

Grammatical Inference deals with the problem of inferring language models from samples of a given language with the goal to infer a model that represents exactly the language from which the samples were drawn. Hence, the main goal of Grammatical Inference is the identification of the original language rather than the anomaly detection. A language most often consists of (untimed) words ((untimed) sequences), but some approaches also solve the grammatical inference task for timed languages (a set of timed sequences).

Sequence-based Anomaly Detection solves the problem of detecting anomalies in sequence data—depending on the approach for untimed or timed sequences. In this research area, anomalies shall be detected in unknown sequences after learning a model from training sequences that do not contain anomalies.

In the following sections, we describe the different research areas more extensively including notation, models and algorithms, followed by other anomaly detection approaches that originate from other areas that anomaly detection, e.g. intrusion detection or ATM fraud detection.

3.2 Process Mining

Process Mining bridges the gap between *process modelling* and *data mining* and deals with the modeling and extraction of process models from log data. A good overview is given by van der Aalst [Aal11]. Usually, a *process model* is extracted from software and business process log data (*process discovery*). The extracted models can be analyzed by humans or the models are employed to detect irregularities in running processes by checking whether new log data is still conformal with the process model (*conformance checking*). Usually, the processes are modeled in the Business Process Model and Notation (BPMN) language.

In this section, we give an overview on process mining. Therefore, we first introduce the data that usually belongs to a process log, then review a process discovery algorithm (α-algorithm) and different conformance checking approaches.

3.2.1 Data Format

Process data is usually saved in a *process log* (or *event log*). Apart from meta data a process log consists of arbitrary many traces. Traces are sometimes referred to as single process instances or cases. A trace has a unique identifier (case id) and consists of one or more events. Each event belongs to exactly one trace and can consist of several attributes. Every event must have a case id, an activity and must be ordered with respect to other events in the same case. The case id indicates to which case the event belongs and the activity represents the event's name. Apart from these two attributes, often occurring attributes are the following:

- **Event id**: A unique identifier for the event. This identifier is often a running number.
- **Timestamp**: A timestamp or date that represents the point in time when the event occurred. If a timestamp is given, the order of the events should reflect the temporal order of the timestamps.
- **Resource**: A resource the event is related to. This may be the name of a customer, an object that is processed, the person or machine producing this event etc.
- **Cost**: The amount of cost for the event.

Table 3.1 shows an excerpt of a process log for handling compensations. Every event has an event id, a timestamp, an activity, a resource and a cost value. The

case id is only shown for the first event of a case for simplicity. In a real log, every event would have a value in the field "Case id". The "Activity" represents the name of the action taken, the "Resource" the person performing the action and the "Cost" the cost value associated with the event. The first case represents a trace where a compensation request starts with the activity "Register request" performed by Pete with cost 50 at 30-12-2010:11.02 with event id 35654423. This event is followed by the activity "Examine thoroughly" performed by Sue with cost 400 at 31-12-2010:10.06 with event id 35654424. The first case is completed with the activity "Reject request" performed by Pete with cost 200 at 07-01-2011:14.24 with event id 35654427. Then the cases with case ids $2, 3$ and 4 follow where each case consists of different events.

Table 3.1: A fragment of an event log for handling compensations: each line corresponds to an event (according to [Aal11])

Case id	Event id	Timestamp	Activity	Resource	Cost
1	35654423	30-12-2010:11.02	Register request	Pete	50
	35654424	31-12-2010:10.06	Examine thoroughly	Sue	400
	35654425	05-01-2011:15.12	Check ticket	Mike	100
	35654426	06-01-2011:11.18	Decide	Sara	200
	35654427	07-01-2011:14.24	Reject request	Pete	200
2	35654483	30-12-2010:11.32	Register request	Mike	50
	35654485	30-12-2010:12.12	Check ticket	Mike	100
	35654487	30-12-2010:14.16	Examine casually	Pete	400
	35654488	05-01-2011:11.22	Decide	Sara	200
	35654489	08-01-2011:12.05	Pay compensation	Ellen	200
3	35654521	30-12-2010:14.32	Register request	Pete	50
	35654522	30-12-2010:15.06	Examine casually	Mike	400
	35654524	30-12-2010:16.34	Check ticket	Ellen	100
	35654525	06-01-2011:09.18	Decide	Sara	200
	35654526	06-01-2011:12.18	Reinitiate request	Sara	200
	35654527	06-01-2011:13.06	Examine thoroughly	Sean	400
	35654530	08-01-2011:11.43	Check ticket	Pete	100
	35654531	09-01-2011:09.55	Decide	Sara	200
	35654533	15-01-2011:10.45	Pay compensation	Ellen	200
4	35654641	06-01-2011:15.02	Register request	Pete	50
	35654643	07-01-2011:12.06	Check ticket	Mike	100
	35654644	08-01-2011:14.43	Examine thoroughly	Sean	400
	35654645	09-01-2011:12.02	Decide	Sara	200
	35654647	12-01-2011:15.44	Reject request	Ellen	200

As a lot of software and business produce log data, the process mining community[1] decided to form the IEEE Task Force on Process Mining[2]. This task force discusses important matters of process mining, e.g. a standard format for saving and exchanging

[1] `http://www.processmining.org/`
[2] `http://www.win.tue.nl/ieeetfpm`

process logs, i.e. log files that contain data extracted from processes. The current standard format for process logs is called *eXtensible Event Stream* (XES)[3]. Figure 3.1 shows a part of the log data from Table 3.1 in the XES format. The case with the *id* or *concept:name* "1" contains events. These events contain the attributes shown in Fig. 3.1 and their corresponding values.

```
<log xes.version="1.0" ...>
   ...
   <trace>
      <string key="concept:name" value="1"/>
      <event>
         <string key="concept:name" value="register request"/>
         <string key="org:resource" value="Pete"/>
         <date key="time:timestamp" value="2010-12-30T11:02:00.000+01:00"/>
         <string key="Event_ID" value="35654423"/>
         <string key="Costs" value="50"/>
      </event>
      <event>
         <string key="concept:name" value="examine thoroughly"/>
         <string key="org:resource" value="Sue"/>
         <date key="time:timestamp" value="2010-12-31T10:06:00.000+01:00"/>
         <string key="Event_ID" value="35654424"/>
         <string key="Costs" value="400"/>
      </event>
      <event>
         <string key="concept:name" value="check ticket"/>
         <string key="org:resource" value="Mike"/>
         <date key="time:timestamp" value="2011-01-05T15:12:00.000+01:00"/>
         <string key="Event_ID" value="35654425"/>
         <string key="Costs" value="100"/>
      </event>
      <event>
         <string key="concept:name" value="decide"/>
         <string key="org:resource" value="Sara"/>
         <date key="time:timestamp" value="2011-01-06T11:18:00.000+01:00"/>
         <string key="Event_ID" value="35654426"/>
         <string key="Costs" value="200"/>
      </event>
      <event>
         <string key="concept:name" value="reject request"/>
         <string key="org:resource" value="Pete"/>
         <date key="time:timestamp" value="2011-01-07T14:24:00.000+01:00"/>
         <string key="Event_ID" value="35654427"/>
         <string key="Costs" value="200"/>
      </event>
   </trace>
   <trace>
      <string key="concept:name" value="2"/>
      <event>
         <string key="concept:name" value="register request"/>
         <string key="org:resource" value="Mike"/>
         <date key="time:timestamp" value="2010-12-30T11:32:00.000+01:00"/>
         <string key="Event_ID" value="35654483"/>
         <string key="Costs" value="50"/>
      </event>
      ...
   </trace>
   ...
</log>
```

Figure 3.1: A fragment of an event log in XES format (according to [Aal11])

3.2.2 Anomaly Detection

In Process Mining the anomaly detection task is solved applying *conformance checking* approaches [AM05]. In conformance checking the conformance of a given trace (sequence) with respect to the model is checked. Therefore, the trace is replayed on the model and it is checked whether rules are violated during this replay (also called *log replay*). To the best of our knowledge conformance checking is only done

[3] `http://www.xes-standard.org/`

on an event level without considering the time values. In Section 3.2.3.2 we present algorithms that deal with conformance checking.

3.2.3 Models & Algorithms

Several models exist to represent a process and several mining algorithms can infer the model from given data. In most cases, Petri nets are used for modeling processes [Aal98; AS11]. The most commonly used algorithm for inferring a Petri net from process logs is the α-algorithm [AWM04]. In this section, we review the α-algorithm and the model extracted by it, and describe different approaches for conformance checking.

3.2.3.1 α-Algorithm (Petri net inference)

In [AWM04] the α-algorithm for the process discovery step is explained. The α-algorithm extracts a *Structured Workflow net (SWF-net)* (a special Petri net[4]) from a given event log and can then be used for conformance checking of the found model compared to new log data. Workflow nets (WF-nets) are used to describe the pattern of a process with transitions modeling events and also so called "AND"-splits/joins. Places contain marks (as usual) but also model "OR"-splits/joins. Figure 3.2 shows a WF-net for the process steps A, B, C, D and E. It also contains an "OR"-split after transition "A" and an "AND"-split that models parallelism.

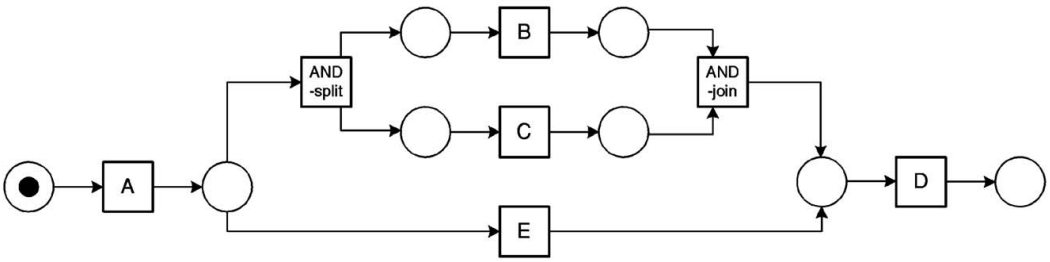


Figure 3.2: A workflow net (WF net) for the process steps A, B, C, D and E (cf. [AWM04]).

WF nets must satisfy the following three properties:

1. *Object creation*: The workflow net contains a starting place that has no ingoing transitions.
2. *Object completion*: The workflow net contains a stopping place that has no outgoing transitions.
3. *Connectedness*: The workflow net is connected if adding a transition which connects the starting and stopping place yields strong connectivity.

Structured Workflow nets (SWF-nets) additionally require that choices ("OR"-splits) should not be mixed with synchronizations ("AND"-joins) and that "OR"-joins

[4]For an introduction to Petri nets we refer to [Mur89].

should not precede "AND"-splits[5]. A *workflow log* of a WF net contains sequences that are accepted by the WF net (e.g. AED or ABCD for the WF net in Fig. 3.2). Furthermore, four relations (direct ordering ($>$), direct causality ($\rightarrow$), parallelism ($||$), no direct ordering ($\#$)) on pairs of events in a workflow log are defined. Given these definitions and making assumptions about the underlying model, the α-algorithm can infer the underlying model: Given a sound Structured Workflow net (SWF-net) N (i.e. N is safe, properly completes, can always complete, and contains no dead transitions) with no short circuits (circuits of length one or two) and a complete workflow log of this SWF-net N (the log contains all possible direct orderings $>$ of transitions and every transition occurs in the log) the α-algorithm provably reconstructs N from the complete workflow log. For non-structured WF nets the α-algorithm does not provably reconstruct the original workflow net.

The α-algorithm works as follows: All transitions are reconstructed from the direct ordering $>$. The initial and stopping place can be inferred from the sequences, as well. Then, the algorithm finds all causal links between all possible sets of transitions and discards all but the largest sets. Doing so yields those transitions that can safely be executed in parallel.

3.2.3.2 Conformance Checking

Given the α-algorithm for the process discovery step, conformance checking is applied in several approaches. In this section we review different approaches of conformance checking.

In [AM05] the authors apply the α-algorithm to analyze traces of user-system interactions (audit trails) to detect security violations. It is assumed that the workflow log only contains acceptable audit trails (only normal behavior). These audit trails are regarded as complete workflow log and the underlying WF net is sound. Then the so called "token game" can be played by moving tokens from one place to the other in the WF net as long as firing a transition is possible. If firing a transition is not possible and the stopping place has not been reached, an anomalous process execution is found at the current place. Additionally, the authors provide a means to check whether specific patterns are true for given log data. Therefore, a subnet is built from the pattern and the conformance is checked for the subnet.

[Aal05] investigates how to compare defined processes and user behavior for business processes. *Delta analysis* compares the predefined and the real processes and conformance checking is applied to measure the fit between the event logs and the predefined processes. For the delta analysis the authors apply the *Greatest Common Divisor* (GCD) and *Least Common Multiple* (LCM) on processes as defined in [AB01]. Applying the GCD (LCM) on processes yields common superclasses (subclasses) of the two process models. Hence, these metrics show similarities or differences in the predefined process model and the one extracted from the process log. For the conformance checking, a predefined model is directly compared with an event log. The authors distinguish between two possibilities of conformance violations: *underfitting* and *overfitting*. Underfitting (in terms of conformance violations) is described as behavior that exists in the process log but is not possible according to

[5]For the formal definition we refer to [AWM04].

the predefined model. Overfitting means that the log file is covered by the predefined process model but parts of the predefined process model are not addressed by the process log. According to the authors it is difficult to quantify overfitting, thus they only present different metrics for underfitting. If a sequence of events is checked for conformance, even after a failure (non-blocking semantics) it is possible to show which transition in the process model usually differs with respect to the log.

In [ASS11] process models are enriched with timing information to predict the completion time of running instances. Instead of using the α-algorithm to construct a Petri net a state-transition system similar to an automaton is constructed. In the first step an abstraction function is applied, e.g. to only take into account the set of events of a sequence instead of regarding the sequence and the ordering of events. As in the previous approach, the authors mention over- and underfitting, but do not present approaches to reduce these problems. For the prediction of remaining time until completion of a sequence, every sequence in the (training) log is replayed and for every (abstract) state that is visited during the replay, the remaining execution time is added to a bag of the (abstract) state. Then, general prediction functions are introduced that predict the remaining time of a sequence given, which part of the sequence has already been executed and how long this execution lasted. For specific prediction functions, the authors propose the average, minimum, maximum or standard deviation of the bag values of an abstract state.

Most of the models and mining algorithms are freely available in the process mining workbench *ProM* [Don+05]. ProM offers to load data in various formats (XES, csv, etc.), mine models from the data and store the mined models. Thus ProM fulfills the same tasks in the field of process mining as Weka [Hal+09] for machine learning.

The International Business Process Intelligence Challenge (BPIC) is a challenge that aims at comparing different approaches for solving process mining problems. It is held in conjunction with the International Workshop on Business Process Intelligence (BPI)[6] for the fifth time. At the challenge, a data set consisting of different process logs from a company or public authority is made available and questions related to the data set should be answered.

3.3 Grammatical Inference

Grammatical Inference studies the field of inferring (or learning) a language $\mathcal{L}$ from a set of words S where each word in S is contained in $\mathcal{L}$. Usually, an inference (or learning) algorithm constructs a model from S that reflects $\mathcal{L}$ as good as possible. Depending on the complexity of $\mathcal{L}$, different algorithms and models perform better than others. Usually, the complexity of $\mathcal{L}$ is given in advance such that the best learning algorithm and model can be chosen. Some Grammatical Inference algorithms and their models are presented in Section 2.3 and Section 2.4.

In most cases, $\mathcal{L}$ is a regular language and a Deterministic or Non-Deterministic Finite Automaton is used to construct (or sample) words from $\mathcal{L}$. Given a set of words S from this language $\mathcal{L}$, the goal is to construct a model that accepts only the words in $\mathcal{L}$ and rejects all other words. Sometimes, the set of words S is also called input sample and the words are called strings. Every word (or string) consists

[6] `http://www.win.tue.nl/bpi/`

of arbitrary many letters, symbols or characters. The word *abcab* is a sample word from the alphabet $\Sigma = \{a, b, c\}$.

After presenting the data format of the grammatical inference research area, we review different algorithms from this area and present results of a grammatical inference challenge—the PAutomaC challenge.

3.3.1 Data Format

The data format of the grammatical inference community was mainly developed during the PAutomaC challenge. This data format is defined in the following way: The alphabet Σ may only contain consecutive integers, so no characters are allowed. The first line of the input file must contain the number of words in the input sample followed by the alphabet size. Every following line contains one word, starting with the length of the word followed by its symbols (integers in this case). Figure 3.3 shows an input sample with five words and an alphabet size of eight ($\Sigma = 8$). The first word contains twelve symbols, the second seven symbols etc.

```
5 8
12 5 4 1 1 5 3 4 7 4 7 5 0
7 4 4 7 4 4 4 7
12 2 4 4 5 5 4 2 2 7 1 0 3
2 1 3
9 5 4 3 2 3 3 2 4 2
```

Figure 3.3: A PAutomaC input sample containing five words with $|\Sigma| = 8$

Even though most algorithms focus on (untimed) words, Verwer, de Weerdt, and Witteveen introduced the RTI+ algorithm for *timed* words that learns a (real-)time automaton. A timed word (or timed string) does not only contain letters from an alphabet Σ but also time values corresponding to every letter. Thus, a timed word is a sequence of letter-time value tuples $s_t = (a_1, t_1)(a_2, t_2) \dots (a_n, t_n)$ with $a_i \in \Sigma$ and $t_i \in \mathbb{N}$. Every time value t_i represents the time delay between an event a_i and its successor a_{i+1}.

For timed words Verwer introduced another input format which is related to the one presented in PAutomaC—apart from letters the timed words also contain the associated time values in form of tuples, separated by a double space. Figure 3.4 shows a timed input sample derived from Fig. 3.3 enriched with time values. The log still contains five sequences and an alphabet of size eight ($\Sigma = 8$). The first sequence contains twelve symbols. It starts with the symbol '5', then waits 20 time units, followed by symbol '4' waiting nine time units etc.

```
5 8
12 5 20  4 9  1 0  1 200  5 17  3 19  4 23  7 29  4 31  7 37  5 41  0 42
7 4 1  4 2  7 3  4 4  4 5  4 6  7 7
12 2 1  4 9  4 5  5 7  5 9  4 3  6 12  2 11  7 14  1 6  0 0  3 2
2 1 5  3 7
9 5 1  4 2  3 3  2 4  3 5  3 6  2 7  4 8  2 9
```

Figure 3.4: An input sample in Verwer's file format containing five *timed* words with $|\Sigma| = 8$

3.3.2 Anomaly Detection

Grammatical Inference deals with inferring language models from given data. Therefore, the task is to infer exactly the same model from which the data was sampled from. Most often, the difference between the sampling model and the inferred model is computed, instead of measuring how many anomalies are detected. When applying Grammatical Inference to detect anomalies, the problem reduces to the word acceptance problem. For a new word, it is checked whether the word is accepted by the inferred model. If the word is not accepted, it is marked as anomaly.

3.3.3 Models & Algorithms

In this section, we mainly review PDFA-inference algorithms with different theoretical bounds or in different scenarios. Then, we present the approaches and results of the PAutomaC challenge for which different models are inferred to reconstruct an unknown language. Last, we describe an approach that infers an automaton model to classify truck driver behavior in a semi-supervised fashion.

3.3.3.1 PDFA Inference

In this section, we review PDFA-inference algorithms that probably converge in the PAC framework or operate on data streams. Then, we review models and algorithms that were applied in the PAutomaC challenge, and briefly summarize the result of the challenge.

PAC-Learning PDFAs As described in Section 2.4.4, different algorithm analysis frameworks exist. ALERGIA and MDI converge in the limit, but not in the PAC framework, while the algorithms reviewed here do so. Before reviewing the algorithms we review a common definition of automata, namely the μ-distinguishability: A PDFA is μ-distinguishable if for any two states q_1, q_2 their suffix distribution (of strings) differs at least by μ.

In [CT04] Clark and Thollard present an algorithm that polynomially PAC learns a PDFA (cf. Section 2.4.4.2). The algorithm applies a state-merging technique and uses the Kullback-Leibler Divergence (KLD; Section 2.4.2.2) as error measure. To be able to compute a PDFA in the PAC framework, the algorithm requires data polynomial in the following three parameters:

- The number of states of the target automaton n.
- The distinguishability of the automaton μ.
- The maximum expected length of strings generated from any state of the target automaton.

For computing the target automaton the algorithm additionally requires the following parameters[7]:

- The confidence parameter δ.

[7] We do not present the algorithm here but instead refer to [CT04].

- The precision parameter ε.
- The alphabet Σ.

Hence, for PAC learning a PDFA the algorithm requires six parameters with some of them possibly being hard to specify. Sometimes, this algorithm is also called the C-T algorithm.

Castro and Gavaldà describe another algorithm that also PAC learns PDFAs in [CG08]. It is closely related to the C-T algorithm but PAC learns in a stronger sense. Additionally, the algorithm does not require the distinguishability parameter μ as input and the runtime neither depends on the precision parameter ε nor on the maximum expected length of strings. Opposed to the C-T algorithm their approach tries to extract as much information as possible from a given sample instead of demanding the calculated number of required samples. The authors perform preliminary experiments and provide calculations that the polynomial of the runtime of the C-T algorithm is in the order of 10^{24} for interesting parameters $(\delta, \varepsilon, \mu, \dots)$ whereas their algorithm's runtime is much lower. Finally, they believe (but cannot prove) that all but the confidence parameter δ can be removed while still satisfying the PAC requirements.

PDFA-Inference on Data Streams [BCG12] presents an approach where PDFAs are not learned in a batch manner but from a stream of sequences (or words). Therefore, the algorithm always keeps a structure hypothesis (DFA). Every state is marked as *safe*, *candidate* or *insignificant*, with safe states being fixed in the hypothesis, candidate states being potential safe states and insignificant states being statistical noise. As for batch PDFA learners, states can be merged, but in the stream scenario only candidate and safe states can be merged, and the similarity of two states is only tested after some time. Additionally, a parameter search algorithm is presented that finds the right hyperparameters for the PDFA inference algorithm because the hyperparameters are hard to estimate at the beginning of the stream. Finally, the authors present a change detection test that computes whether the current hypothesis (DFA) significantly differs from the recent content of the sequence stream.

PAutomaC Challenge The PAutomaC challenge [VEH14] was held in 2012 as a competition between various language learning algorithms. The organizers supplied (untimed) data sets of languages that were sampled from randomly constructed NFAs, PDFAs and Hidden Markow Models (HMMs; [Bau+70]). The aim of the challenge was to test algorithms on different types of problem domains and to figure out whether one algorithm is superior to others or whether the choice of the right algorithm is problem-specific. The participants had to submit an algorithm that learns a model on a training set S_{train} and computes a probability for every word of a test set S_{test}. After every algorithm submission, the organizers evaluated every algorithm on a test set of every language, that was known (sic!) to the participants, but the participants did not know the generating model. For every string in the test set, the probability computed by the respective algorithm was compared with the real probability for that string computed by the generating model. Hence, the

challenge was not limited to automaton models, but every algorithm that computes a probability for an unknown word was allowed to participate. For the PAutomaC challenge a simple version of ALERGIA was given as a baseline.

Five teams submitted algorithms that scored results by being among the top four algorithms for at least one data set. In the following we will shortly present the different approaches.

Team Shibata-Yoshinaka In [SY12] team Shibata-Yoshinaka describes their approach of *Collapsed Gipps Sampling* (CGS) [BJ06] applied to finite automata learning. Therefore, they assume a Dirichlet distribution of the transition probabilities such that the computation becomes easier. Using Gibbs Sampling for every state they iteratively fix all but one state (and their transition probabilities) and update the transition probabilities of that one state. Finally, they sample the states (and their transition probabilities) multiple times and use their average as result. Even though the PAutomaC challenge did not have any bounds on algorithm runtime, the organizers point out that CGS is computationally much more expensive than all other algorithms submitted to the challenge.

Team Hulden In [Hul12] team Hulden presents their software package Treba that contains algorithms that infer NFAs, PDFAs and HMMs. For the PAutomaC challenge they started with an n-gram approach that counts the subwords of length n and computes their probabilities. Later, they changed to an approach that starts with a random non-deterministic finite automaton of fixed size and then applies the Baum-Welch algorithm [Wel03] (an expectation maximization algorithm) to refine the automaton.

Team Llorens Team Llorens submitted a modified version of ALERGIA that does not directly perform a merge if two significantly similar states are found, but also computes the similarity of all other states. Then it performs the merge that is the most significant.

Team Bailly In [Bai11] team Bailly describes the quadratic weighted automaton (QWA) as a probabilistic model for estimating string probabilities. For finding the QWA they apply a spectral approach that uses a Hankel matrix that represents the counts of every possible prefix-suffix pair. The Hankel matrix factorization directly yields the parameters of the QWA.

Team Kepler In [KMB12] team Kepler presents their approach of n-grams of variable length for learning probabilistic automata. As n-grams of variable length result in a huge memory consumption, they are stored in a tree that is pruned using a threshold to reduce the memory footprint. Surprisingly, this approach performed worse than a 3-gram on almost all data sets.

PAutomaC Result Team Shibata-Yoshinaka clearly won the PAutomaC challenge, while team Hulden and team Llorens almost achieved the same result with team Hulden closely leading. Team Bailly's first submissions were among the best

ones but they were ranked fourth at the end. Team Kepler achieved a low score, resulting in the fifth place.

PDFA-Inference for Labeled Sequences Regular Positive and Negative Inference (RPNI) [OG92] is the two-class variant of ALERGIA and learns a PDFA from labeled untimed sequences. Hence, the event sequences either belong to the positive sequence set S_e^+ or to the negative one S_e^-. RPNI ensures that the resulting automaton does not accept negative sequences from the training data. It starts by building a prefix tree. In contrast to ALERGIA, RPNI does not apply a statistical significance test to check whether two states are similar. Instead, RPNI performs a merge operation and checks afterwards whether the resulting automaton accepts negative sequences. If so, the merge is rejected, and accepted otherwise. Furthermore, Oncina and García prove that RPNI converges to the true model in the limit and its runtime is cubic in the input data.

Real-Time Identification (RTI) [VWW11] is the two-class variant of RTI+ (cf. Section 2.4.3.2), and also applies the Evidence-Driven State Merging approach (EDSM). As RTI+, it learns a PDRTA from given data. The sequences in the training data have to be labeled, thus the training data contains positive and negative sequences $S_t^{+/-}$, and not only positive sequences S_t^+. If the data originates from a real-world system, the data is distinguished into correct sequences and sequences that are either faulty or do not originate from the system. The resulting PDRTA accepts all positive sequences from the training data and rejects all negative ones.

RTI operates similarly to RTI+. It also builds a prefix tree with the positive sequences and additionally holds a set of rejecting states that represents the negative sequences. As RTI+, RTI computes the evidence value for every possible split and merge for every pair of red and blue states (red-blue framework). Additionally, RTI checks for every split or merge whether this operation caused an inconsistency. An inconsistency is present if red states are rejecting states or if a transition from a red state is taken via a positive and negative sequence. Then RTI performs the action with the highest evidence score. Verwer, Weerdt, and Witteveen prove that RTI runs in polynomial time, is correct, complete and converges efficiently to the correct PDRTA (in the limit) [VWW11].

3.3.3.2 Classification of Truck Driver Behavior

In [VDW11] Verwer, De Weerdt, and Witteveen describe a semi-supervised approach of applying RTI+ to classify the driving behavior of truck drivers. Therefore, they first generate discrete events from continuous signals which are recorded while the truck is driving. Additionally, the time intervals between two succeeding events are stored. Using the extracted timed sequences, RTI+ is applied to infer a PDRTA that describes the possible driving behavior of a truck driver. To be able to distinguish between two classes of driving behavior (e.g. accelerating too quick or normally), a few labeled sequences are generated in a supervised fashion. With the help of these labeled sequences the learned PDRTA is augmented with accepting and rejecting states that represent the two classes. Using the augmented PDRTA, it is possible to correctly classify 79% of the test sequences.

3.4 Sequence-based Anomaly Detection

Sequence-based Anomaly Detection deals with the automatic model generation and anomaly detection in discrete event systems. While Process Mining and Grammatical Inference mostly deal with automatic model generation, Sequence-based Anomaly Detection also focuses on the anomaly detection. The model reflects the normal behavior of some system and the anomaly detection is used to detect anomalies while the system is running. These two parts can be considered separately, but both parts play an equally important role.

As Sequence-based Anomaly Detection deals with sequences, the input for the model generation is called *set of sequences*. This set consists of (timed) sequences with every sequence consisting of event-time value tuples.

The set of sequences can be acquired from most technical system. A (technical) system often writes events to a log file. This log file contains at least events and a timestamp for each event and may contain even more information, e.g. a payload. As this log file contains the essential items of timed sequences (events and time values) it can be interpreted as a set of timed sequences.

3.4.1 Data Format

We decided to introduce a third file format for timed sequences. On the one hand, the XML format for process logs is very verbose, needs a lot of storage space and contains more information than usually needed for sequence based model generation. On the other hand, the file format introduced by Verwer (cf. Fig. 3.4) is hard to read as it uses integers for both the events and the time values. Furthermore it uses spaces between events and the corresponding time value and double spaces as separator between event-time value tuples. Often, the difference between a single and a double space is not visible and thus, events and time values are mixed up when reading such a log. Nevertheless, it only contains information about the timed sequences.

Thus, our proposed file format combines the advantages of both previously presented file formats. It is closely related to the Verwer file format, omits some information, and also enhances readability. The new file format only contains the needed information for timed sequences, allows event qualifiers to be arbitrary strings, and encloses event-time value tuples with brackets. The entries on the size of the file, the size of the alphabet and the length of a sequence are omitted. Figure 3.5 shows the same data as in Fig. 3.4 but with a different alphabet. Every integer from the alphabet was replaced by a character (a to g) except the integer 7 that was replaced by the event qualifier "some_event". The brackets show the event-time value tuples such that a confusion between an event qualifier and a time value will be less likely.

```
(f,20) (e,9) (b,0) (b,200) (f,17) (d,19) (e,23) (some_event,29) (e,31) (some_event,37) (f,41) (a,42)
(e,1) (e,2) (some_event,3) (e,4) (e,5) (e,6) (some_event,7)
(c,1) (e,9) (e,5) (f,7) (f,9) (e,3) (g,12) (c,11) (some_event,14) (b,6) (a,0) (d,2)
(b,5) (d,7)
(f,1) (e,2) (d,3) (c,4) (d,5) (d,6) (c,7) (e,8) (c,9)
```

Figure 3.5: An input sample in the new file format containing five *timed* sequences, with the alphabet $\Sigma = \{a, b, c, d, e, f, g, some\_event\}$ and $|\Sigma| = 8$

3.4.2 Anomaly Detection

Sequence-based Anomaly Detection deals with detecting anomalies in data consisting of sequences of different length. The task is to label unknown sequences as abnormal or normal with the help of a model that has been learned before. As presented in Section 2.5.2.1 ANODA is the only algorithm that solves the anomaly detection for timed sequences given a timed automaton model (PDTA).

In the following section, we present approaches from the research area of sequence-based anomaly detection that operate on slightly different data, e.g. on untimed sequences or hybrid data (processing also continuous signals).

3.4.3 Models & Algorithms

Sequence-based anomaly detection has been applied using different approaches and in several different scenarios. In the following, we present a selection of approaches to give an overview of possible approaches and applications.

3.4.3.1 Untimed Sequence Data

Apart from PDFA-learning algorithms like ALERGIA, other approaches have been used to detect anomalies in untimed sequences. We review t-stide and a k-NN approach as representatives for frequency- and distance-based approaches.

t-stide (frequency-based) A well-known sequence-based anomaly detection method is the threshold-based sequence time delay embedding (t-stide) proposed by Warrender, Forrest, and Pearlmutter [WFP99]. For every (untimed) test sequence s_e, t-stide computes an anomaly score that is defined as the relative frequency of s_e in the training set. The idea is that rare sequences are likely to be anomalies. A test sequence is determined as anomaly if its anomaly score is smaller than a predefined threshold t. The t-stide method has been successfully applied to various application, e.g. in operating system intrusion detection based on sequences of system calls [WFP99; CLM01].

k-NN (distance-based) Chandola, Mithal, and Kumar [CMK08] propose a k-nearest neighbor (k-NN) approach for sequence-based anomaly detection. The anomaly score for a test sequence s_e is equal to the inverse distance of s_e to its k^{th} nearest neighbor in the training set. If its anomaly score is lower than a predefined threshold t, the test sequence is considered as anomaly. In spite of its simplicity, this approach has been shown to be quite effective, and it is even able to outperform a complex clustering-based technique [CMK08]. However, the runtime of the approach increases with the size of the training set because the distance between the test sequence s_e and every training sequence must be computed. Additionally, the performance quality of the approach depends on the chosen distance metric. The normalized length of the longest common subsequence has been shown promising for anomaly detection in sequence data [Bud+06]. Another possible distance metric for sequences is the Dynamic Time Warping (DTW) algorithm [SC78].

3.4.3.2 The Hybrid Bottom-Up Timed Learning Algorithm (HyBUTLA)

The Hybrid Bottom-Up Timed Learning Algorithm (HyBUTLA) [Nig+12] is an extension of BUTLA that infers a PDTA and additionally learns continuous signals for every state. The resulting model is called hybrid automaton which is a PDTA enriched with functions Θ that describe the continuous signals in every state with $\theta_q \in \Theta$. This hybrid automaton can model hybrid systems (Section 2.2) which are a mixture of discrete and continuous systems.

HyBUTLA requires a different input than BUTLA consisting of signal measurements at a fixed frequency. The signals have to be distinguished into discrete signals, continuous input and output signals, and the time itself. A change in a discrete signal is transformed into an event (or symbol). This event generation can also be done for BUTLA.

After the PTA generation (based on the events) continuous signals can be mapped to states because the signals between two events can be tracked. Then, all continuous signals in a state q are gathered and a function θ_q is learned e.g. with simple linear regression or more complex regressors. The time can also be incorporated into θ_q.

In the merge step, states are checked for compatibility as in BUTLA. Additionally, the similarity of the continuous signal functions has to be examined. For every regressor the similarity check is different and may introduce additional complexity [Vod13]. For linear regression, at least 50% of the coefficients had to be similar according to some predefined threshold to merge a state.

HyBUTLA can also be used for detecting anomalies in hybrid systems. Then, it does not only track event and time values (as ANODA does for BUTLA) but also keeps track of the continuous signals. An alarm is raised if in a state q the predicted continuous signals θ_q differ too much from the measured signals.

3.5 Other Anomaly Detection Approaches

In this section, we review approaches that tackle the anomaly detection problem on vectors of fixed length (Section 3.5.1), on data streams (Section 3.5.2) or that try to prevent fraud on ATMs (Section 3.5.3).

3.5.1 Vector-based Anomaly Detection

Vector-based anomaly detection deals with detecting anomalies in a data set that consists of vectors of fixed length. Various vector-based approaches are evaluated on a data set published by the DARPA in 1999.

DARPA Challenge

In 2000 the Defense Advanced Research Projects Agency (DARPA) organized the second offline intrusion detection challenge in which participants try to detect attacks in a data set containing network traffic. In [Lip+00] the results of the second DARPA challenge are presented. The organizers generate network data similar to a government site with hundreds of users and thousands of hosts. The data is split into three weeks of training and two weeks of test data. In this period 200 attacks

were launched consisting of 58 different types of attacks. The participants apply host- and network-based intrusion detection systems as well as forensic analysis on file system data. The best participant was not able to detect ten (out of 58) types of attacks, hence the detection rate for some attacks was very poor. After the DARPA challenge was organized, newly developed approaches were tested on the DARPA data set because the data set was considered as being "hard to solve".

PAYL [WS04] is a payload-based anomaly detector that computes a profile byte frequency distribution in the training phase and compares test data against the built profile. Due to the fact that different ports behave differently, for every port and length of the network packet a new byte frequency distribution of the packet payload is created during the training phase. A new byte distribution of the respective payload is computed for an arriving packet and compared against the distribution for the according port and packet length. For comparing the two distributions the authors apply Mahalanobis distance and use a threshold to detect outliers. On the overall DARPA data PAYL achieves 60% detection rate at a false positive rate of 1%. For port 80 PAYL achieves 100% detection rate with 0.1% false positive rate.

[HLV03] applies Robust SVMs (RSVMs; [SHX02]) on the data set of the DARPA challenge. The standard SVM is very sensitive to noise in the training data. Robust SVMs are less sensitive to noise because they are not penalized for misclassification in the training set. Instead, the aim is to obtain a smooth decision boundary with fewer support vectors. The authors compare RSVMs with standard SVMs and a k-nearest neighbor (kNN) approach. To show the superiority of RSVMs over SVMs a part of the DARPA data set is mixed with noise. Surprisingly, the RSVM outperformed the kNN approach and the SVM not only on the noisy data set, but also on the clean data set, achieving 80% detection rate at a false positive rate of 1%. Additionally, RSVMs require less support vectors to represent the training data. The authors conclude that RSVMs generalize better because of less support vectors and—as a consequence—a smoother decision boundary.

In [CMA03] two approaches are presented—namely Learning Rules for Anomaly Detection (LERAD) and Clustering for Anomaly Detection (CLAD). Usually, rule learners compute rules in the form $(A = x) \rightarrow (B = y)$ with A, B attributes of the vector and x, y arbitrary values. These rules demand that if attribute A has value x then attribute B has to have value y. For anomaly detection LERAD computes rules $(A = x) \rightarrow (\neg B = y)$ that demand that if attribute A has value x then attribute B must not have value y. If B has value y the given vector is a possible anomaly. For every rule a likelihood is computed and all violated rules define the anomaly score of a given vector $\mathbf{x}$.

CLAD assigns every vector $\mathbf{x}$ to one or more clusters. First, it computes a maximal cluster width W by sampling pair-wise point distances and using the average distance of the closest neighbors as W. CLAD checks for every vector $\mathbf{x}$ whether it is at most distance W away from a cluster centroid. If so, $\mathbf{x}$ is assigned to this cluster and else, $\mathbf{x}$ becomes a new cluster centroid itself. In a second phase, every vector $\mathbf{x}$ is also assigned to clusters that are within distance W of $\mathbf{x}$. After the clustering, every cluster that is sparse and distant from the other clusters is considered as anomalous. For the definition of sparsity and distance in the context of CLAD we refer to [CMA03].

LERAD and CLAD are evaluated in the DARPA 99 dataset [Lip+00]. The best original DARPA participant detected 85 anomalies out of 201 with a hand-built signature detector. LERAD is able to detect 117 anomalies, but CLAD only 76. However, LERAD assumes an anomaly-free training set while CLAD does not.

3.5.2 Anomaly Detection on Data Streams

Up to now we reviewed approaches that deal with the anomaly detection problem in a batch manner. In a batch manner the whole (training) data is available and can be used more than once. In the *data stream* domain new data is continually arriving and has to be processed without being able to store all arriving data points. Furthermore, for every data point some property (e.g. a class label) has to be computed before the next data point arrives. Thus, every data point can be processed only once and the property computation must be done fast and efficiently. Sometimes, data streams pose additional challenges when the nature of the data slowly changes (*concept drift*), abruptly changes (*concept change*), new classes arise (*concept evolution*) or new features occur (*feature evolution*).

As WEKA [Hal+09] exists for classical machine learning, Massive On-line Analysis (MOA) [Bif+10] is built for data stream mining. MOA contains a collection of stream mining algorithms in the fields of stream classification, regression, clustering, outlier detection and online recommender systems. In this section we present approaches that are not available in MOA, but cover an important aspect of data stream mining.

[Zho+03] presents an approach for KDEs on data streams. Even though KDEs are not directly related to anomaly detection, they play an important role in this thesis and hence, we also describe the streaming application. The idea of the approach is to merge kernels with the same value to have a single (M-)kernel for those values. In the offline scenario an M-kernel has three parameters: a weight, a mean and a bandwidth. For a possibly infinite stream only storing one M-kernel per distinct value would still result in an infinite amount of required memory. Hence, the authors introduce the approximate kernel merging that merges not only kernels with the same value but also similar ones according to a presented similarity measure. Thus, the number of kernels is fixed such that the KDE always uses the same amount of memory. In the experiments the authors show that their streaming approach and the standard KDE almost yield the same result.

In [Mas+13] a stream classification approach is presented that deals with concept-drift, concept evolution and feature evolution at once. The authors state that most approaches only cover the possibly infinite length of a stream and a concept drift. They propose an ensemble classification framework with majority voting that covers all data stream challenges. It processes the stream in chunks of fixed size and learns a classifier on one chunk after it has been labeled (by a human expert). Concept drift is covered by always learning new classifiers from the most recent stream data and removing the worst classifiers from the ensemble. Each classifier in the ensemble is equipped with a novel class detector that stores the recent outliers and if enough outliers were found, the detector checks whether a new class exists in those outliers. For handling feature evolution the authors propose a lossless feature set homogenization technique (cf. [Mas+10]). In the experiments the approach is found to be superior to existing data stream classification approaches.

3.5.3 ATM Fraud Detection

In the experiments of this thesis (cf. Section 6.5) we apply our approach on real-world ATM data to detect manipulations of ATMs. Therefore, we review related techniques that try to prevent ATM fraud, too.

In [DS06] an image-based fraud detection system for ATMs is proposed that tries to detect whether money is withdrawn by the owner of a credit card or by a criminal who stole the card. Most ATMs contain cameras that capture the face of the person withdrawing money. The authors assume that criminals would mask their face while withdrawing money with a stolen card. Hence, they propose a pipeline that takes the camera input and detects moving objects with a mixture of Gaussians, then detects the face region and finally extracts facial features, e.g. hiding the eyes with a cap or sunglasses, covering everything but the eyes with a mask, etc. Based on these features and manually defined thresholds a video sequence is classified as normal or fraud. The approach is only tested on few sample videos but no real-world evaluation is performed.

[RNM12] deals with the problem of detecting counterfeit cards usage by visualizing ATM usage patterns. The visualization is presented to a human to detect the fraud. Criminals usually follow patterns when committing credit card fraud. First, they steal or copy credit cards and capture the PIN (called *counterfeiting phase*). In the *distribution phase* cards are distributed among different persons as they are sold or bought on the black market. Finally, the criminals check whether a card is valid and how much cash is available on the card (called *reconnaissance phase*). Only then, money is withdrawn from the card (*cashing out phase*). For the reconnaissance and cashing out phase criminals use less frequented ATMs during the night that do not have camera surveillance. There, they first check the validity and balance of cards in a batch manner without withdrawing money, but only sorting out invalid cards. Second, they empty accounts or withdraw the maximum daily amount, again for a batch of cards. These patterns can be visualized and are easily detectable for humans. For future work the authors propose to automate the process of detecting suspicious behavior.

In [DE15] Demiriz and Ekizoglu proposes a rule-based approach based on spatio-temporal relations to prevent banking fraud. The authors gathered data from different ATMs in Turkey and found usage patterns that could separate customers based on their behavior. If a customer behaves differently from his or her normal behavior an alarm could be raised. The paper visually shows a lot of usage patterns and gives hints on what could be implemented, but it neither implements a rule-based system nor presents results on the efficiency of the approach.

Summary

In this chapter, we reviewed related work and related different research areas—namely Process Mining, Grammatical Inference and Sequence-based Anomaly Detection. For every research area, we described the data format, which models and algorithms are applied and to what extent anomaly detection is performed. In the field of process mining, most often Petri nets are mined from data using the α-algorithm. With the help of these Petri nets, *conformance checking* analyzes whether the current process deviates, e.g. because of an anomaly. The grammatical inference research area mainly deals with learning automaton models (and grammars) from predefined languages, that are hidden from the learner. Some of the researchers deal with timed languages and timed automata. These models can then be used to check whether a given word is accepted by a previously learned automaton. Sequence-based anomaly detection utilizes various different models and algorithms ranging from k nearest neighbor methods to hybrid automata learning algorithms. All approaches have in common that the learning of a model enables detecting anomalies in unknown data afterwards.

Table 3.2 summarizes the content of this chapter with the rows containing the notations of the respective research area for the terms *event*, *sequence* and *set of sequences*. For some entries more than one term exists, thus the table cell contains two (or three) terms. Even though the different research areas do not cooperate much, they partially solve a similar or even the same task.

Furthermore, we presented other anomaly detection approaches that do not originate from the reviewed research areas. These approaches tackle the anomaly detection problem on vectors of fixed length, on data streams and for detecting ATM fraud.

In the following chapter, we present a new automaton model for representing timed sequences (PDTTA), a learning algorithm for this model (Probabilistic Deterministic Timed Transition Automaton Learning (ProDTTAL)) and an anomaly detection algorithm for timed automaton models (AmAnDA).

Table 3.2: Overview of different notations for events, sequences and sets of sequences.

research area	**notation for event**	**notation for sequence**	**notation for set of sequences**
Process Mining	event	trace / case	(process) log
Grammatical Inference	symbol / letter / character	word / string	input sample
Sequence-based Anomaly Detection	event	(timed) sequence	set of (timed) sequences

4
Anomaly Detection with PDTTAs

In this chapter, we present three contributions of this thesis. First, we present a new automaton type called *Probabilistic Deterministic Timed Transition Automaton (PDTTA)* (cf. Section 4.2) that models time more accurately than existing automaton models. Second, we develop the Probabilistic Deterministic Timed Transition Automaton Learning (ProDTTAL) that learns PDTTAs from given positive timed sequence data, and analyze its runtime and convergence (cf. Section 4.3). Third, we present the Automata-based Anomaly Detection Algorithm (AmAnDA) that can detect sequence-based anomalies given a PDTTA or a similar automaton model that meets certain requirements (cf. Section 4.4), and analyze the runtime of AmAnDA. Finally, we describe how these algorithms could be applied in a two-/multi-class setting and which difficulties would emerge (cf. Section 4.5).

The PDTTA, a preliminary version of AmAnDA and a naive anomaly detection algorithm have already been published in [KAK14; Kle+14]. Apart from that the content of this chapter is developed in this thesis only and was never published before.

4.1 Motivation

Automata of various types are adequate models for detecting anomalies in many technical systems. The automata should be learned from data because defining an automaton model for a complex system is very time-consuming or impossible—even for a domain expert. Additionally, the automaton should be capable of detecting anomalies for a new event sequence. Regarding timed sequences in technical systems the state-of-the-art algorithms Real-Time Identification from Positive Data (RTI+) and Bottom-Up Timed Learning Algorithm (BUTLA) still face challenges.

As already presented in Section 2.4.3.2 the RTI+ algorithm has several drawbacks, e.g. the necessity to globally define histogram intervals for the time values and the poor execution time of the algorithm. On the other hand RTI+ learns a real Probabilistic Deterministic Real-Time Automaton (PDRTA) and takes time information into account when merging or splitting states.

BUTLA tries to circumvent some drawbacks with an event split based on the global time values for every event in a preprocessing step. After the event split a Probabilistic Deterministic Finite Automaton (PDFA) is learned via state merging and the time guards are added afterwards based on the global timing information of each event. A split of automaton states is not executed at all. The event split based on global timing information can lead to undesired behavior because huge deviations in a *local* state may not even be reflected in the *globally* executed event

split. Nevertheless, the execution time of BUTLA is lower compared to RTI+.

We introduce a new type of automaton called PDTTA that considers the timing for every transition in the automaton *locally* and is easy to learn in terms of computational complexity and evaluated runtime. After introducing the PDTTA (Section 4.2) we present the learning algorithm (called *ProDTTAL*) that learns a PDTTA given a positive set of timed sequences S_t^+ (Section 4.3). Then we show how we can decide for a test timed sequence s_t whether it is an anomaly given a PDTTA (Section 4.4).

4.2 Probabilistic Deterministic Timed Transition Automaton (PDTTA)

In this section we present the Probabilistic Deterministic Timed Transition Automaton (PDTTA) with which we model discrete event systems. We first described PDTTA in [KAK14] and refined it in [Kle+14]. The PDTTA is an extension of a Probabilistic Deterministic Finite Automaton (PDFA) (cf. Section 2.3.2.2) with an additional function τ that assigns a plausibility function to every transition. Every plausibility function models the plausibility of a time value when the corresponding transition is taken. The definition of a PDTTA (with the elements of a PDFA) is as follows:

Definition 12 (PDTTA)**.** *A Probabilistic Deterministic Timed Transition Automaton is a six-tuple* $(\Sigma, Q, q_0, \delta, \pi, \tau)$ *with*

- Σ *is a finite alphabet comprising all relevant symbols.*
- Q *is a finite set of states.*
- $q_0 \in Q$ *is the start state.*
- $\delta : Q \times \Sigma \to Q$*: The state-transition function assigning a state and symbol the successor state (for some state-symbol combinations it may be undefined).*
- π *twofold:*
  - $\pi_1 \colon \delta \to [0, 1]$*: The transition probability function assigning a probability to every transition .*
  - $\pi_2 \colon Q \to [0, 1]$*: The final state probability function assigning a probability to every state .*
- $\tau \colon \delta \to \Theta$ *is a transition time plausibility function, which assigns a plausibility distribution* $\theta \in \Theta$ *to each transition, with* Θ *being the set of all possible plausibility distributions. Every* $\theta \in \Theta$ *has the signature* $\theta \colon \mathbb{N}_0 \to [0, 1]$*, with* $\mathbb{N}_0$ *representing all possible time values.*

As for the PDFA we also write π instead of π_1 or π_2 if it is clear from the context. Figure 4.1 shows a PDTTA with the underlying PDFA from Fig. 2.6 in Section 2.3.2.2. In addition to the states, symbols and event probabilities, the function τ is added to represent the time value plausibility functions. Every plausibility function expresses how plausible a time value is with 1 being the most plausible outcome and 0 being

not plausible. In contrast to probability functions, the sum of all plausibility values may be greater than one. As we utilize the PDTTA for anomaly detection, modeling time values using plausibility functions is a reasonable choice because different time values may be equally plausible (with their plausibility value being one) which would not be possible with probability functions.

The time plausibility function for the transition (q_0, e_1, q_1) contains two modes with high plausibility for the time values around these two modes and zero plausibility for the rest of the time values. The other time plausibility functions only contain one mode with the function of transition (q_1, e_2, q_0) being less spiky than the one for transition (q_1, e_3, q_2).

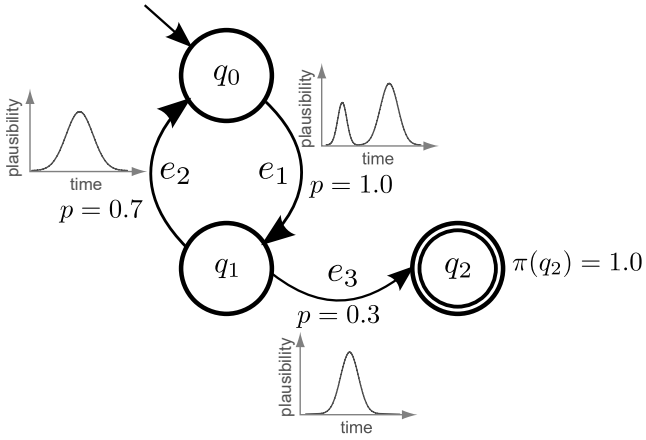


Figure 4.1: PDTTA with three states (q_0, q_1, q_2), three symbols (e_1, e_2, e_3), event probabilities and time plausibilitiy functions.

The idea of adding τ to a PDFA is to model the waiting time for every transition. Opposed to Probabilistic Deterministic Real-Time Automata (PDRTAs) we do not model the time as a strict decision criterion that may be true or false. In a PDRTA, the time value 4.0 with event e_1 may be a valid input in some state q, but the time value 4.000001 with the same event e_1 in the same state q may be invalid[1] In the PDTTA, we model the time values more smoothly and assign a plausibility to every time value. Note, that we limit the model not being able to have more than one successor node for a state-event pair, or in other words, the transition to take depends on the current state and the symbol, but not on the time value. This limitation is still consistent with the definition of discrete event systems as state changes are caused by events.

4.3 Learning PDTTAs

In this section we present how to learn a PDTTA from a positive set of timed sequences S_t^+. The learning of a PDTTA was first described in [KAK14] and refined in [Kle+14]. In this thesis we analyze its runtime and show how Monte Carlo sampling is applied for approximating plausibility values of new sequences.

[1] In a Probabilistic Deterministic Real-Time Automaton (PDRTA) the time value is strictly in the interval bounds or outside, but nothing in between.

4.3.1 PDTTA Learning Algorithm ProDTTAL

In this section we present an algorithm for learning PDTTAs called Probabilistic Deterministic Timed Transition Automaton Learning (ProDTTAL). For obtaining a PDTTA from a positive set of timed sequences S_t^+ ProDTTAL performs two steps. In the first step, a PDFA A' is learned with a state-of-the-art inference algorithm by leaving out the time information in S_t^+. Thus, we transform the set of *timed* sequences S_t^+ into a set of *untimed* sequences S_e^+ by transforming every timed sequence $s_t = (e_1, v_1), (e_2, v_2), \ldots, (e_n, v_n)$ into an untimed sequence $s_e = e_1, e_2, \ldots, e_n$. Every algorithm that infers a PDFA from S_e^+ can be used in this step. We will focus on Minimal Divergence Inference (MDI) and ALERGIA because they are well known, simple and thus easy to understand.

In the second step, we enrich the PDFA A' from the first step with time information, thus adding the transition time plausibility function τ (cf. Definition 12) to A' and obtaining a PDTTA. We call this second step *time learning*. In the following we will use the expression to traverse a PDFA with a *timed* sequence, i.e. we traverse the PDFA with the untimed part of the given sequence. In general a PDFA can only process *untimed* sequences.

To obtain a PDTTA from a PDFA A', we adapt a technique from process mining called *event log replay*. For every timed sequence $s_t = (e_1, v_1), (e_2, v_2), \ldots, (e_n, v_n) \in S_t$, we traverse A' with s_t and for every transition that is taken because of an event e_i, we store its time value v_i in a bucket that belongs to this transition. After all sequences are replayed, we fit a probability density function (PDF) for every bucket using a Kernel Density Estimation (KDE). Then, we apply a probability-possibility transformation to compute the plausibility function from the PDF. More formally, the second step proceeds as follows (cf. Algorithm 5: ProDTTAL-time-learning):

Given the PDFA A' from step one, for every transition $(q_i, e_i, q_j) \in \delta$ for a state q_i, a symbol e_i and the successor state q_j we initially create a bucket $B_{(q_i,e_i,q_j)}$ (line 2). As the transition is already uniquely defined by the state and a symbol, we use a short notation B_{q_i,e_i} instead of $B_{(q_i,e_i,q_j)}$. For every timed sequence $s_t \in S_t$ we traverse A' starting with the current state being the initial state q_0 (line 4). Then we process s_t and for every symbol e_i in s_t and the current state q we store the associated time value v_i in the bucket B_{q,e_i} (line 6), traverse the transition $(q, e_i, q_j) \in \delta$ (line 7) and set the current state to q_j (line 8). Finally, we compute τ by approximating a time plausibility function $\theta_{q,e}$ for every transition $(q_i, e_i, q_j) \in \delta$ (line 10). The details of this approximation are explained in the following section.

4.3.1.1 Time Plausibility Approximation

In this section, we describe how we approximate the time plausibility functions. Even though we compute the plausibility values for the anomaly detection step, the approximation influences the runtime of the model learning algorithm ProDTTAL because we apply a preprocessing step.

For the approximation of the time plausibility functions we first fit probability density functions for the gathered time values at every transition. Then, we apply a probability-possibility transformation [Dub+04] to obtain plausibility functions.

We use a KDE to approximate the density functions of the time values because KDE

ALGORITHM 5: ProDTTAL-time-learning

Data: Positive set of timed event sequences S_t^+, PDFA $A' = (\Sigma, Q, q_0, \delta, \pi)$
Result: transition time plausibility function τ

```
foreach transition (q_i, e, q_j) ∈ δ do
    initialize(B_{q_i,e});
foreach sequence s_t ∈ S_t^+ do
    q ⟵ q_0;
    foreach tuple (e_i, v_i) ∈ s_t do
        B_{q,e_i}.add(v_i);
        t ⟵ getTransition(q, e_i);
        q ⟵getTargetState(t);
foreach transition (q_i, e, q_j) ∈ δ do
    τ(q_i, e, q_j) ⟵approximateTheta(B_{q_i,e});
return τ
```

is a non-parametric estimator that does not assume anything about the underlying distribution but use the data to model approximate the density. In the case of ProDTTAL we do not know anything about the underlying distribution, either[2].

Given a density function $\hat{f}_{q,e}$ for a bucket $B_{q,e}$ which is associated with the transition from state q and event e, we can compute the plausibility $\theta_{q,e}(v)$ when observing the time value v. Therefore, we compute the area under the density function where the density value $\hat{f}_{q,e}(x)$ is lesser or equal than the density at point v. Hence, the plausibilty function $\theta_{q,e}$ for every bucket with $[\![\cdot]\!]$ the indicator function is computed as follows:

$$\theta_{q,e}(v) = \int [\![\hat{f}_{q,e}(x) \leq \hat{f}_{q,e}(v)]\!] \hat{f}_{q,e}(x) \mathrm{d}x \tag{4.1}$$

Computing or approximating the integral may become a hard task. We will first present the density estimation in detail and then explain how we compute the integral.

The fitting of a probability density function (PDF) depends on the algorithm used to fit the PDF. We found that KDE perform well in most applications[3]. It is possible to set the kernel and bandwidth the same for every transition or apply some heuristic to obtain a good kernel and a good bandwidth. As the values obtained for the different transitions may vary in magnitude, it may be poor to choose the same bandwidth for all KDEs. Hand-tuning the bandwidth for every transition is a hard task as well and requires a lot of expert knowledge. Hence, we use the rule of thumb from [Sil86] as an approximation for the bandwidth h of every transition with $\hat{\sigma}$ the standard deviation of the sample and n the number of samples[4]:

$$h = 1.06 \hat{\sigma} n^{-\frac{1}{5}} \tag{4.2}$$

[2]The described probability-possibility transformation is also applicable to parametric distributions.
[3]We are aware that kernel density estimators and PDFs generally deal with continuous data. We use them to approximate integer data that stems from an underlying continuous process with a fixed (integer) resolution.
[4]The whole formula (including the constant 1.06) is deduced in [Sil86].

We will now explain how we approximate the time plausibility functions Θ when given density estimators $\hat{f}_{q,e}$ for any state q and event e. Instead of trying to compute the exact integral for the already approximated density function $\hat{f}_{q,e}$, we approximate this function with Monte Carlo integration applying uniform sampling [Pre+96]. In case of bad approximations this may lead to poor results but it is a reasonable tradeoff between accuracy and computation time. Recall, that we want to find the area under $\hat{f}_{q,e}$ where the density value $\hat{f}_{q,e}(x)$ is lesser or equal than the density at point v. For simplicity we will assume that $\hat{f}_{q,e}$ is a one-dimensional function[5].

Figure 4.2 shows the steps of the Monte Carlo sampling: We approximate the area under $\hat{f}_{q,e}$ by uniformly sampling a fixed number m of two-dimensional points (x, y) and choosing only those (blue) points with $y \leq \hat{f}_{q,e}(x)$. Let $M_{q,e}$ be this sample with $|M_{q,e}| \leq m$. If we know the minimum and maximum of $\hat{f}_{q,e}$ and the minimum and maximum x-values it is feasible to sample only in this range. Hence, $|M_{q,e}|$ and m differ by a constant factor, only, so $m = \mathcal{O}(|M_{q,e}|)$.

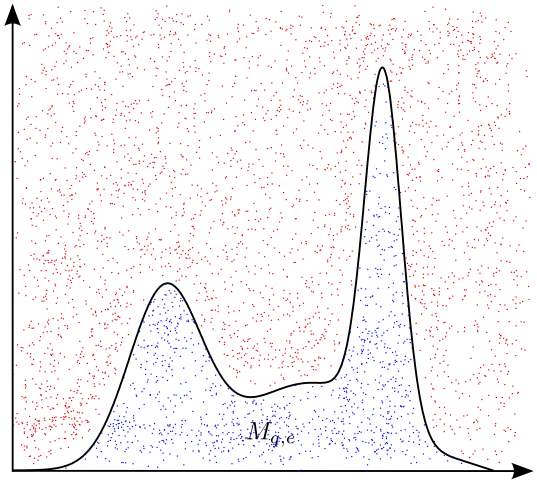


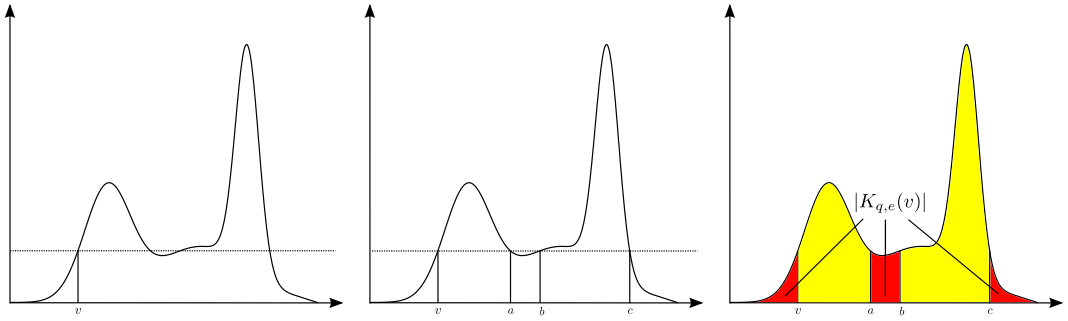
Figure 4.2: Monte Carlo sampling for given KDE.

If we want to compute the plausibility $\theta_{q,e}(v)$ of a point v, we do the following (cf. Fig. 4.3 for the illustration): First, we compute the density value $\hat{f}_{q,e}(v)$ at point v (Fig. 4.3a) and second, we select all points $(x, y) \in M_{q,e}$ where the density value of x is lesser or equal than the density of v (Fig. 4.3b). Formally, let $K_{q,e}(v)$ be this set with $K_{q,e}(v) = \{(x, y) | (x, y) \in M_{q,e} \wedge \hat{f}_{q,e}(x) \leq \hat{f}_{q,e}(v)\}$ (Fig. 4.3c). Third, we compare the number of points in $K_{q,e}(v)$ and to the number of points in $M_{q,e}$. The fraction of the cardinality of the two sets is the approximated plausibility of the time value v, hence $\theta_{q,e}(v) \approx \frac{|K_{q,e}(v)|}{|M_{q,e}|}$.

To compute $K_{q,e}(v)$ quickly, we sort the points in $M_{q,e}$ ascending according to their density value of the x-coordinate ($\hat{f}_{q,e}(x)$). If two points have the same density value we use the y-coordinate as a tiebreaker.

Algorithm 6 shows the Probabilistic Deterministic Timed Transition Automaton Learning (ProDTTAL) algorithm in pseudo code. Given a positive set of timed event

[5]Nevertheless, Monte Carlo integration can be computed for arbitrary functions.



(a) Density for time value v and intersection.

(b) Intervals with lower and higher density.

(c) Comparison of density areas.

Figure 4.3: Plausibility approximation of a time value t.

sequences S_t^+ and a PDFA learner L, ProDTTAL first infers the PDFA A' using L and then computes the time plausibility functions using ProDTTAL-time-learning (Algorithm 5).

ALGORITHM 6: Probabilistic Deterministic Timed Transition Automaton Learning (ProDTTAL)

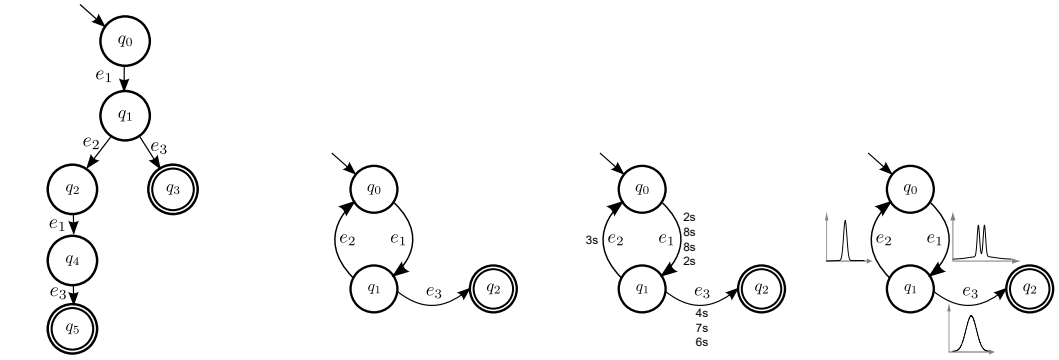
Data: Positive set of timed event sequences S_t^+, PDFA learner L
Result: PDTTA A
1 $A' \longleftarrow L$.`learnPDFA` (S_t^+);
2 $\tau \longleftarrow$ProDTTAL-`time-learning`(S_t^+, A');
3 $A \longleftarrow$PDTTA(A', τ);
4 **return** A

Figure 4.4 shows the (simplified) behavior of the two steps for a small sample consisting of the following sequences:

- $s_{t_1} = (e_1, 2s), (e_3, 4s)$
- $s_{t_2} = (e_1, 8s), (e_2, 3s), (e_1, 8s), (e_3, 7s)$
- $s_{t_3} = (e_1, 2s), (e_3, 6s)$

In the first step a Prefix Tree Acceptor (PTA) is created from the input sample S_t^+. This PTA is merged with a PDFA learning algorithm (e.g. ALERGIA) to obtain a PDFA.

In the second step (time learning) the sequences are replayed and the time values are stored for every transition. After all sequences have been processed, a probability density function is approximated from the values in the respective bucket for every transition using kernel density estimators. Then, we sample points for every density function and store them. These points are needed later to compute the plausibility of an unknown time value.



(a) Build the PTA. (b) Learn a PDFA. (c) Replay sequences. (d) Fit KDEs.

Figure 4.4: Learning a PDTTA from timed sequences.

4.3.2 Runtime Complexity

In this section we will analyze the runtime complexity of ProDTTAL. As our algorithm uses a PDFA inference algorithm and a KDE, its complexity also depends on the respective algorithms. Moreover, the complexity of the inference algorithm and the KDE is based on different criteria: Given an input sample S_t^+ in most cases the complexity of the PDFA algorithm depends on the size of the PTA that is constructed from S_t^+ (cf. Sections 2.4.2.1 and 2.4.2.2). The complexity of the KDE depends on the number of samples that the estimator has to take into account. Because of these different factors, we introduce the following notation:

- $|S_t^+|$: The number of sequences in the input sample S_t^+.
- l_{max}: The longest sequence in S_t^+ (in terms of sequence length).
- B_{max}: The bucket containing the most time values ($B_{max} = \arg\max_{B_{q,e}}\{|B_{q,e}|\}$).
- $|PTA(S_t^+)|$: The number of states in the PTA that is constructed from S_t^+.

If we apply ALERGIA or MDI for learning the PDFA we need to take into account the respective runtime, which depends on the PTA creation ($\mathcal{O}(|S_t^+| \cdot l_{max})$; cf. Section 2.4.1.2) and the merge step ($\mathcal{O}(|PTA(S_t^+)|^3)$ for ALERGIA and $\mathcal{O}(|PTA(S_t^+)|^3 \cdot |\Sigma|)$ for MDI (with $|\Sigma|$ the size of the alphabet; cf. Section 2.4.2.1). For other PDFA-inference algorithms the runtime may not depend on the PTA built from S_t^+. In the following we will denote the complexity to infer the PDFA from the input S_t^+ with $PDFA(S_t^+)$.

For the event log replay the runtime complexity does neither depend on the size of the PTA nor the size of the PDFA but only depends on the number of time values occurring in the input S_t^+. If we assume that all sequences have equal length l and thus equally many time values, the number of time values in the input is $|S_t^+| \cdot l$. If we do not assume anything about the sequence length, we can upperbound the number of time values by using the maximal sequence length l_{max}. The upper bound for the number of time values TV in the input S_t^+ for the bucket filling is:

$$TV\left(S_t^+\right) = \mathcal{O}\left(|S_t^+| \cdot l_{max}\right) \tag{4.3}$$

We will now describe the runtime complexity of the steps performed to approximate Θ with Monte Carlo integration. In this section we will only describe the preprocessing that is done once for every bucket while inferring the PDTTA. Every computation of a plausibility value $\theta_{q,e}(v)$ for a time value v in a state q and event e requires additional effort that is described in Section 4.4.2.3.

At first we will focus on the preprocessing for *one* single bucket $B_{q,e}$. At the end we will calculate the complexity for all buckets. For the Monte Carlo integration we need to compute the PDF of the KDE a couple of times based on the desired resolution. Recall, that we sampled two-dimensional points and stored those points that are under the PDF in the sample set $M_{q,e}$ with $m = \mathcal{O}(|M_{q,e}|)$. This sample set defines the resolution of the Monte Carlo approximation.

Proposition 1. *The runtime complexity of the preprocessing step for a single bucket with two-dimensional Monte Carlo integration is*

$$\mathcal{O}\left(m \cdot (|B_{q,e}| + \log m)\right)$$

with m being an upper bound on the number of points sampled and $|B_{q,e}|$ the size of the bucket for state q and event e.

Proof. For every sampled point we have to check whether it is contained in $M_{q,e}$ so we have to compute the density for every sampled point. Computing the density k times for a sample of n points requires $\mathcal{O}(k \cdot n)$ many evaluations of the kernel function. Transferred to our notation with $n = |B_{q,e}|$ and $k = m$ we end up with $\mathcal{O}(|B_{q,e}| \cdot m)$ many evaluations of the kernel function for one bucket $B_{q,e}$. Additionally, we sort $M_{q,e}$ according to the density values of the points. If we save the density value for every sampled point after checking whether the point is contained in $M_{q,e}$, we must only sort $M_{q,e}$, which requires $\mathcal{O}(|M_{q,e}| \cdot \log |M_{q,e}|) = \mathcal{O}(m \cdot \log m)$. Hence, the runtime complexity for the Monte Carlo preprocessing for one bucket $B_{q,e}$ is:

$$\begin{aligned} \text{Single bucket Monte Carlo preprocessing} \hat{=} & \mathcal{O}\left(|B_{q,e}| \cdot m + m \cdot \log m\right) \\ = & \mathcal{O}\left(m \cdot (|B_{q,e}| + \log m)\right) \end{aligned} \tag{4.4}$$

■

The calculation from Proposition 1 has to be done for every bucket. We can now analyze the runtime for the Monte Carlo preprocessing for the whole PDTTA:

Proposition 2. *The runtime complexity of the preprocessing step for all buckets of a PDTTA with two-dimensional Monte Carlo integration is*

$$\mathcal{O}\left(m \cdot \left(TV\left(S_t^+\right) + |\delta| \cdot \log m\right)\right)$$

with m an upper bound on the number of points sampled, $|\delta|$ the number of transitions in the PDFA, and $TV(S_t^+)$ the number of time values in the input S_t^+.

Proof. As the preprocessing for the Monte Carlo integration (Proposition 1) has to

be done for every bucket, we sum up the costs for all buckets[6]:

$$\begin{aligned}\text{Monte Carlo preprocessing} \hat{=} &\mathcal{O}\left(\sum_{(q,e,*)\in\delta}(m\cdot(|B_{q,e}|+\log m))\right)\\ =&\mathcal{O}\left(m\cdot\sum_{(q,e,*)\in\delta}(|B_{q,e}|+\log m)\right)\\ =&\mathcal{O}\left(m\cdot\left(\sum_{(q,e,*)\in\delta}\log m+\sum_{(q,e,*)\in\delta}|B_{q,e}|\right)\right)\\ =&\mathcal{O}\left(m\cdot\left(|\delta|\cdot\log m+\sum_{(q,e,*)\in\delta}|B_{q,e}|\right)\right)\end{aligned}\tag{4.5}$$

As the sum over the size of all buckets $\left(\sum_{(q,e,*)\in\delta}|B_{q,e}|\right)$ is the same as the number of time values in the input, we use the upper bound from Eq. (4.3) and rewrite Eq. (4.5):

$$\text{Monte Carlo preprocessing} = \mathcal{O}\left(m\cdot\left(TV\left(S_t^+\right)+|\delta|\cdot\log m\right)\right)\tag{4.6}$$

■

Now, the runtime of the whole θ-preprocessing step can be easily formulated:

Lemma 3. *The runtime complexity of the θ-preprocessing step with two-dimensional Monte Carlo integration is*

$$\mathcal{O}\left(m\cdot\left(TV\left(S_t^+\right)+|\delta|\cdot\log m\right)\right)$$

with m an upper bound on the number of points sampled, $|\delta|$ the number of transitions in the PDFA, and $TV(S_t^+)$ the number of time values in the input S_t^+.

Proof. The runtime directly follows from adding the costs for replaying the time values (filling the buckets), which is exactly the number of time values in the input S_t^+ and the runtime from Proposition 2:

$$\begin{aligned}\theta\text{-preprocessing} \hat{=}\ &\underbrace{TV\left(S_t^+\right)}_{\text{bucket filling}}+\underbrace{\mathcal{O}\left(m\cdot\left(TV(S_t^+)+|\delta|\cdot\log m\right)\right)}_{\text{Monte Carlo preprocessing (Proposition 2)}}\\ =&\mathcal{O}\left(m\cdot\left(TV\left(S_t^+\right)+|\delta|\cdot\log m\right)\right)\end{aligned}\tag{4.7}$$

■

Note that the runtime of Lemma 3 is an upper bound on the number of evaluations of the kernel function of the KDE. Depending on the chosen kernel the runtime complexity may be constant (uniform kernel) or include the computation of a point

[6]There is one bucket for every transition.

on an exponential function (Gaussian kernel). Thus, we will ignore the complexity of the kernel function.

Now we can formulate the runtime for the inference of a PDTTA:

Theorem 4. *The runtime complexity of inferring a PDTTA (ProDTTAL) from a positive timed sequence set S_t^+ in combination with two-dimensional Monte Carlo integration for approximating the time value plausibilities is*

$$\mathcal{O}\left(PDFA\left(S_t^+\right) + m \cdot \left(TV\left(S_t^+\right) + |\delta| \cdot \log m\right)\right)$$

with $PDFA(S_t^+)$ the runtime for inferring the PDFA, m an upper bound on the number of points sampled, $|\delta|$ the number of transitions in the PDFA, and $TV(S_t^+)$ the number of time values in the input S_t^+.

Proof. The runtime follows from the runtime complexity of the PDFA-inference algorithm and Lemma 3 (Eq. (4.7)):

$$\mathcal{O}\Big(PDFA\left(S_t^+\right) + \underbrace{m \cdot \left(TV\left(S_t^+\right) + |\delta| \cdot \log m\right)}_{\theta-\text{preprocessing (Lemma 3)}}\Big) \tag{4.8}$$

■

If we use e.g. ALERGIA as inference algorithm, we can specify the time needed for the PDFA inference $PDFA(S_t^+)$. This inference requires to create and merge a PTA. The runtime of the θ-approximation is independet of the PDFA-inference runtime.

$$\begin{aligned}&\mathcal{O}\Big(\underbrace{|S_t^+| \cdot l_{max}}_{\text{PTA creation}} + \underbrace{|PTA\left(S_t^+\right)|^3}_{\text{PTA merge}} + \underbrace{m \cdot \left(TV\left(S_t^+\right) + |\delta| \cdot \log m\right)}_{\theta-\text{preprocessing (Lemma 3)}}\Big)\\ =&\mathcal{O}\Big(|PTA\left(S_t^+\right)|^3 + m \cdot \left(TV\left(S_t^+\right) + |\delta| \cdot \log m\right)\Big)\end{aligned} \tag{4.9}$$

For the PDTTA inference and using ALERGIA for the PDFA inference we end up with a runtime complexity that depends on the number of states of the PTA ($|PTA\left(S_t^+\right)|$), the size of the sample drawn for the Monte Carlo integration (m; larger value leads to better approximation), the number of time values in the input ($TV\left(S_t^+\right)$) and the number of transitions in the PDTTA ($|\delta|$).

4.3.3 Convergence of ProDTTAL

In this section, we discuss the convergence of the resulting PDTTA inferred by ProDTTAL. The convergence of ProDTTAL depends on various factors—namely on the underlying PDFA-inference algorithm, on the KDE and the Monte Carlo integration.

As the Monte Carlo approximation can become arbitrarily poor with respect to the original function, it does not provably converge (neither in the limit nor in the PAC framework). In the remainder of this section, we analyze the convergence without taking into account the Monte Carlo approximation.

First, we discuss the convergence of the underlying PDFA-inference algorithm. Depending on the underlying algorithm different statements can be made. ALERGIA converges in the limit to the true PDFA from which the data was sampled [CO94; DT00]. As presented in Section 3.3.3.1 it has been shown in [CT04] that PDFAs are also polynomially PAC-learnable with the (sometimes called) C-T algorithm. The algorithm presented in [CG08] does not require the distinguishability μ as input but still polynomially PAC learns a μ-distinguishable PDFA. Hence, the underlying PDFA can be learned in the limit and in the PAC framework.

The other factor is the convergence of the KDE. To the best of our knowledge, PAC learnability of KDEs has not been shown in the literature because KDEs are mostly studied in the field of statistics (and not in the field of machine learning). Instead, the research focuses on *consistency*. An estimator $\hat{f}_n$ (dependent on the number of samples n) of an unknown distribution f is *(weak) consistent* if the probability tends to zero that the difference between f and $\hat{f}_n$ is greater than some $\varepsilon > 0$ in the limit:

$$\lim_{n\to\infty} \Pr\left(|\hat{f}_n - f| > \varepsilon\right) = 0 \tag{4.10}$$

Strong consistency demands that the estimator $\hat{f}_n$ convergences to the true distribution f in the limit with probability one ($\Pr(\lim_{n\to\infty} \hat{f}_n = f) = 1$). If we assume that the underlying function f is uniformly continuous, it can be shown that KDEs are strongly consistent if we choose a suitable kernel function and bandwidth [WW12]. Hence, KDEs converge in the limit. For further details, we refer to [WW12].

Summarizing, ProDTTAL learns the underlying PDTTA in the limit if we omit the Monte Carlo approximation and assume a uniformly continuous density function of the time values.

4.3.4 Properties of Timed Automata Inference Algorithms

After we introduced the PDTTA and ProDTTAL, we can compare properties of the inference algorithms ProDTTAL, RTI+ and BUTLA. Some properties directly follow from the model they infer (PDTTA, PDRTA and Probabilistic Deterministic Timed Automaton (PDTA)) and some are caused by the algorithm itself. Table 4.1 shows properties of the respective algorithms.

While RTI+ and BUTLA process time values globally, ProDTTAL applies time processing locally in every state. RTI+ requires the user to specify histograms, whereas BUTLA assumes a Gaussian distribution of the time values and splits events based on this assumption into subevents. All algorithms merge states based on events but only RTI+ can also split states (because of events or time)[7]. RTI+ is also the only algorithm capable of merging states because of their time behavior. ProDTTAL cannot identify time-based substructures because the automaton structure is inferred only from the event data (without regarding the corresponding time data). BUTLA can identify time-based substructures using its global and time-based event split while RTI+ identifies these substructures with its Evidence-Driven State Merging approach (EDSM). The probabilities of events and time values solely depend on the current state for RTI+ (because of the PDRTA) and the time values are assumed to

[7]The BUTLA event split is a simplified global state split inducing new symbols.

Table 4.1: Properties of ProDTTAL, RTI+ and BUTLA.

algorithm model	ProDTTAL PDTTA	RTI+ PDRTA	BUTLA PDTA
time processing	local	global (histograms; symbol independent)	global (subevents; symbol dependent)
event-based merge	yes	yes	yes
event-based split	no	yes	no
time-based merge	no	yes	no
time-based split	no	yes	no
time-based substructures	no	yes	yes (preprocessing)
event probabilities	transition	state	transition
time probabilities/ distribution	transition / KDE / plausibilities	state / uniform	transition / not defined

be uniformly distributed. For ProDTTAL and BUTLA the event probabilities depend on the transition taken because both build on some PDFA inference algorithm. The time probabilities can have arbitrary shape for ProDTTAL because they are modeled with KDEs and later transformed into plausibilities (but with the same shape). For BUTLA they are not defined, but a Gaussian distribution is assumed during the event split.

Based on these properties we infer the following strengths(+)/weaknesses(-):

ProDTTAL

+ More accurate modeling of time values with plausibilities.
+ Local time model for every transition.
- Different substructures based on time values cannot be identified.
- Event model (PTA) can only be merged (no split).

RTI+

+ Different substructures based on time values can be identified.
+ Event model (PTA) can be refined via split.
- Global, user-defined time histograms.
- Difficult to distinguish time values of different magnitude with histograms.
- Probabilities based on current state (instead of transition).

BUTLA

+ Event split based on time values is performed once via preprocessing.

- Global event split.

- Gaussian distributions of time values should not overlap.

In Chapter 5 we present PDRTAs that contain weaknesses which are hard to detect for RTI+ and ProDTTAL, respectively. We experimentally show that these weaknesses influence the anomaly detection of automata inferred with RTI+ and ProDTTAL.

4.4 Anomaly Detection

In this section we present how to detect anomalies given a PDTTA from Section 4.3. First we will formulate the anomaly detection problem as a one-class problem, present the anomaly detection algorithm and then analyze its runtime complexity. In [KAK14; Kle+14] we only gave a naive solution for the anomaly detection using two thresholds. In this thesis we additionally present more elaborate techniques how to apply vector-based one-class algorithms to the sequence based anomaly detection.

4.4.1 Anomaly Detection Problem

We formulate the anomaly detection problem as a one-class problem:

Definition 13 (Anomaly Detection)**.** *Given a positive set of timed sequences S_t^+ and a timed test sequence s_t, determine whether s_t is an anomaly with respect to S_t^+.*

This definition does not make any assumption about the algorithms used. It only states that we deal with timed sequences and a positive set of timed sequences S_t^+. In terms of one-class classification, we are given samples from one class (the normal sequences) and we want to decide whether a given sample belongs to the one class (is normal) or does not belong to this class (is anomalous). In [And+14] we evaluated various one-class algorithms in the context of *untimed* sequences. We are not limited to an approach inferring PDTTAs when we refer to Definition 13. Hence, we refine the anomaly detection definition to be model-based:

Definition 14 (Model-based Anomaly Detection)**.** *Given a model A inferred from a positive set of timed sequences S_t^+ and a timed test sequence s_t, determine whether s_t is an anomaly with respect to A.*

This definition still captures the one-class case but better suits our needs as we first infer a model (e.g. a PDTTA; cf. Section 4.2) and second decide for a timed test sequence whether it is an anomaly. With Definition 14 we can present the anomaly detection algorithm to solve the problem described at hand.

4.4.2 Automata-based Anomaly Detection Algorithm (AmAnDA)

In this section we develop the Automata-based Anomaly Detection Algorithm (AmAnDA). AmAnDA decides whether a timed test sequence s_t is anomalous with respect to an PDTTA A. Recall, that in a state q in a PDTTA A we can obtain an event probability and a plausibility of a time value for a given event-time tuple (e_i, v_i):

- $\pi(q, e_i)$ assigns a probability for observing the event e in the current state q.
- $\tau(q, e_i)$ assigns a transition time plausibility function θ that models the plausibility of observing any time value. Given θ we obtain the plausibility of observing the time value v_i when in state q and observing event e_i.

The idea of the algorithm is to traverse the given PDTTA A, compute the respective probabilities/plausibilities and store them. As the probabilities of events (π) and plausibilities of time values (τ) are of different nature they are stored separately. Hence, we obtain two lists: One list containing event probabilities ($P_{\mathbb{E}}$) and one containing time plausibilities ($P_{\mathbb{T}}$).

In the following we present the anomaly detection algorithm with a PDTTA A as input. Nevertheless, the automaton does not need to be a PDTTA. With small changes we can also apply our anomaly detection algorithm to PDRTAs (learned with RTI+) and PDTAs (learned with BUTLA). We only require to obtain two lists containing the event probabilities $P_{\mathbb{E}}$ and the time plausibilities $P_{\mathbb{T}}$.

Algorithm 7 shows the pseudo code of the Automata-based Anomaly Detection Algorithm (AmAnDA). As input we require a timed event sequence s_t, a PDTTA A (or a similar timed automaton), a feature extractor FE and a classifier C. The feature extractor FE and the classifier C will be described in Sections 4.4.2.1 and 4.4.2.2. The feature extractor extracts some features from the two lists and returns one vector on which the classifier C decides whether the vector is an anomaly.

We start in the start state q_0 of the automaton A (line 1). For every tuple (e_i, v_i) in the input sequence s_t (line 2) and the current state we traverse the transition t indicated by state q and event e_i (line 3) and store the probability of this event $\pi(q, e_i)$ in the event probability list $P_{\mathbb{E}}$ (line 4)[8]. For the time value v_i we analogously store its plausibility $\tau(q, e_i)(v_i)$ in the time plausibility list $P_{\mathbb{T}}$ (line 5)[8]. Then we continue with the following state (line 6). At the end of the timed sequence s_t we add the final state probability to the event probability list (line 8). We extract features from the event and time lists ($P_{\mathbb{E}}/P_{\mathbb{T}}$) using the feature extractor FE (line 9) and finally return whether the input sequence s_t is an anomaly using the classifier C with the extracted feature vector $\mathbf{x}$ as input (line 9).

4.4.2.1 Feature Extraction

Our feature extraction method extracts features from the lists containing event probabilities ($P_{\mathbb{E}}$) and time plausibilities ($P_{\mathbb{T}}$). We do so because most classifiers need a vector of fixed length as input but the extracted lists may have arbitrary length

[8] As every event-time tuple defines a unique transition, we may also write $\pi(t)$ or $\tau(t)$ instead of $\pi(q, e_i)$ or $\tau(q, e_i)$ for a transition $t = (q, e_i, q_j)$.

ALGORITHM 7: Automata-based Anomaly Detection Algorithm (AmAnDA)

Data: timed event sequence s_t, PDTTA $A = (\Sigma, Q, q_0, \delta, \pi, \tau)$, feature extractor FE, classifier C
Result: true/false

```
1 q ⟵ q_0;
2 foreach tuple (e_i, v_i) ∈ s_t do
3 |   t ⟵ getTransition(q, e_i);
4 |   P_𝔼.append(π(t));
5 |   P_𝕋.append(τ(t)(v_i));
6 |   q ⟵getTargetState(t);
7 P_𝔼.append(π(q));
8 x ⟵ FE.extractFeatures(P_𝔼,P_𝕋);
9 return C.isAnomaly(x);
```

depending on the input sequence. Hence, we apply a fixed set of transformations which we call feature extraction. As long as we extract the same features for every sequence, we can add or remove any features. We will now give examples for simple features and then describe one complex feature—the length-normalized sequence aggregation—in detail. Every feature is extracted twice: Once from the event probability list $P_{\mathbb{E}}$ and once from the time plausibility list $P_{\mathbb{T}}$.

Simple Features The following list shows an example of features that can be extracted from a list of real-valued entries $(p_1, \ldots, p_l)$. We use these aggregation functions as a starting point provided by most programming languages, but we are aware that other features may improve the performance of AmAnDA. We are aware that we deal with probabilities for the events and plausibilities for the time values. However, these aggregations are only a starting point that can be computed for probabilities and plausibilities. We only have to make sure that probabilities and plausibilities are not mixed.

- Length: The length of the sequence (l).
- Minimum: The minimal value of the respective list ($\min\{p_i\}$).
- Maximum: The maximal value of the respective list ($\max\{p_i\}$).
- Mean: The arithmetic mean of the respective list ($\mu = \frac{1}{l}\sum_{i=1}^{l} p_i$).
- Variance: The arithmetic variance of the respective list ($\frac{1}{l}\sum_{i=1}^{l}(p_i - \mu)^2$).
- Min-difference: The minimum difference between two succeeding entries ($\min\{|p_i - p_{i+1}|\}$).
- Max-difference: The maximal difference between two succeeding entries ($\max\{|p_i - p_{i+1}|\}$).

Length-Normalized Sequence Aggregation This feature is inspired by the average probability of the event list $(p_1, \ldots, p_l)$, and works similarly for plausibilities because their maximal value is also 1. We already introduced it in [KAK14; Kle+14]. To aggregate of the whole sequence a naive approach would be to multiply the entries with each other $(\prod_{i=1}^{l} p_i)$. This naive approach has the drawback that we deal with sequences of different length, so the resulting feature value would decrease for longer sequences. Hence, for a long sequence the sequence's aggregation would always be smaller than for a short sequence. As we compare sequences of different length and want the features to be comparable, we normalize the sequence aggregation according to the length. We do so by taking the l^{th} root of the product of the entries:

$$\text{length-normalized sequence aggregation} = \sqrt[l]{\prod_{i=1}^{l} p_i} \tag{4.11}$$

As this length-normalized sequence aggregation leads to very small values because of multiplying many values lesser or equal than one, it may happen that these small numbers cannot be represented in usual programming languages any more. Hence, we compute the length-normalized sequence aggregation in the logarithmic space for numeric stability, but without a change in semantics:

$$\begin{aligned}
\text{log-length-normalized sequence aggregation} &= \ln\left(\sqrt[l]{\prod_{i=1}^{l} p_i}\right) \\
&= \ln\left(\left(\prod_{i=1}^{l} p_i\right)^{\frac{1}{l}}\right) \\
&= \frac{1}{l} \cdot \ln\left(\prod_{i=1}^{l} p_i\right) \\
&= \frac{1}{l} \cdot \sum_{i=1}^{l} \ln p_i
\end{aligned} \tag{4.12}$$

Runtime The runtime of the feature extraction algorithm is polynomial in the number of extracted features and length of the given list.

Proposition 5. *Given a list $P_{\mathbb{X}} = (p_1, \ldots, p_l)$ extracting k of the presented features requires $\mathcal{O}(l \cdot k)$ steps.*

Proof. Every presented feature requires at most one run over the list $P_{\mathbb{X}}$. As this is done for k features, the proposition directly follows. ■

In this section we presented features that can be extracted for a feature vector. Given this feature vector we describe the algorithms and models we apply for detecting anomalies in the next section.

4.4.2.2 Anomaly Classification

Anomaly classification deals with the challenge of classifying a given vector as normal or abnormal. Most classification algorithms require a training set that contains samples from an underlying unknown process. The aim of the algorithm or the model produced by the classification algorithm is to predict the class given a test sample. As described in Section 2.5.1 the classification algorithm may work in a supervised or unsupervised manner. If we consider two-class problems (one class the positive and one the negative class) the training set may contain samples for both classes (common binary classification) or just samples from the positive class (one-class classification).

In our setting we consider one-class problems where the training set only contains normal sequences as samples. In the test case a sequence may be normal or abnormal. As most one-class classifiers need vectors of fixed length in the training and test set we transform the respective sets containing timed sequences to sets containing vectors of fixed length using the feature extractors described earlier in this section. Now we can define the anomaly classification problem:

Definition 15 (Anomaly classification)**.** *Given a set of normal vectors obtained from a PDTTA A and a training set S_t^+ via a feature extractor, decide whether a given timed test sequence s_t is normal with respect to A and S_t^+.*

Training the Classifier Most classifiers require a training phase in which the classification model is built. The training works similar to AmAnDA (Algorithm 7) but instead of calling `isAnomaly` in line 9 we train the classifier with the vector x obtained in line 8. As input we require the input S_t^+, the learned PDTTA A and a feature extractor FE. For every $s_t \in S_t^+$ we calculate the probability/plausibility lists $P_{\mathbb{E}}$ and $P_{\mathbb{T}}$ using the PDTTA A and then compute a feature vector v with the feature extractor FE. This feature vector is handed to the one-class classifier for training[9].

Choosing the Classifier We can utilize different one-class classifiers for the anomaly classification. The classifier must be able to handle real-valued vectors of fixed length. Depending on the classifier, we also normalize the vector entries to $[0; 1]$. In the following we present potential classification algorithms and their requirements for our anomaly classification setting. These algorithms were explained in Section 2.5.1.

One-Class SVM The one-class support vector machine (SVM) (cf. Section 2.5.1.1) is a special kind of SVM that can handle training with samples from only one class. Therefore, an additional parameter ν is introduced that defines the target rejection rate. By adjusting ν we can vary the percentage of training data that is considered as outlier. Hence, ν can be seen as some threshold. Like every SVM, also the one-class SVM requires training and tuning of various hyper parameters.

[9] This training can be easily transformed to a batch training by first extracting and storing the feature vector for every timed sequence s_t and then training the classifier with all stored feature vectors.

Clustering Although not a real classification algorithm, we can expand clustering to become a one-class classification algorithm as described in Section 2.5.1.2. There are two possibilities to apply clustering for one-class classification[10]:

1. We cluster the data in the training phase, store all clusters and remove noise points. In the test phase we compute the distance of the test vector to k nearest neighbor points and determine whether the distances are smaller than a given threshold.

2. We store all points in the training phase. In the test phase we add the test point to the points from the training phase, cluster all points and determine whether the test point belongs to a cluster or is classified as a noise point.

For both approaches we must specify at least a distance metric and depending on the clustering algorithm even more hyperparameters. For the first approach we also need a threshold on the distance. The second approach may not be applicable to all clustering algorithms as some clustering algorithms do not classify outliers as noise points but instead assign every point to a cluster.

Threshold Classifier As most of the above described approaches require the input of a threshold, the naive approach is to specify a threshold on every vector entry. Depending on the feature extractor the amount of thresholds to specify may become large and thus inconvenient. The classifier labels a test vector as anomaly if one or all of its entries are below or above the respective threshold. Opposed to the other classifiers mentioned above, the threshold classifier does not need any training.

Outlier Robustness In this paragraph we briefly describe how outlier-robust the PDTTA in combination with ProDTTAL and AmAnDA is. The one-class setting requires that the training data only contains samples from one class. Usually, this requirement is not fully satisfied but few samples in the training data are also from another class that should be labeled as outliers. As already described, ANOmaly Detection Algorithm (ANODA) (cf. Section 2.5.2.1) is not outlier-robust. Consider a sequence s^- that is an outlier but part of the training data. If a path in the automaton model exists for such a sequence (as it usually does when learning a model with BUTLA), ANODA labels s^- as normal.

If we learn a PDTTA with ProDTTAL, this sequence s^- would also be part of the resulting PDTTA. However, if we then traverse the automaton, the probability for one transition is presumably low because no other sequences in the training data traverses this transition. Depending on the feature extraction, this low probability is an entry in the feature vector that probably separates this wrong training sample from the rest of the training samples. As the applied one-class classifiers are usually outlier-robust to some extent, s^- will be labeled as outlier if it is presented to AmAnDA in the test data. Model-based approaches can also be made outlier-robust by not incorporating those samples into the learned model. For example, we would have to remove rarely occurring sequences from the learned PDTTA by removing

[10]Some clustering algorithms may not be applicable to both methods.

transitions with a low event probability but we did not implement this because AmAnDA should take care of the outliers.

In [KAK14; Kle+14] we introduce the anomaly detection for timed sequences with PDTTAs. We extracted one feature (length-normalized sequence aggregation) and applied the threshold detector with two thresholds for the aggregation of the $P_{\mathbb{E}}$ and $P_{\mathbb{T}}$, respectively.

4.4.2.3 Runtime Complexity

In this section we will analyze the runtime complexity for the anomaly detection algorithm given one timed test sequence s_t. We only analyze the runtime complexity for the anomaly detection algorithm without taking into account the complexity of the classifier in detail. Then we will roughly describe the impact on the runtime if we employ a classifier that needs training.

The calculation of the time value plausibility $\theta(v)$ (line 5 in Algorithm 7: AmAnDA) has the main impact on the runtime of the anomaly detection because it involves the Monte Carlo integration. Except lines 5, 8 and 9 every other line requires constant time. As we will not analyze the runtime of the latter two lines because they involve the feature extractor and the classifier, we only analyze the runtime complexity of the Monte Carlo integration in line 5.

If we applied the preprocessing described in Section 4.3 the computation of $\theta(v)$ is easier. In the following runtime analysis we assume that the preprocessing was applied. We will first describe the computation of $\theta(v)$ for one time value $timeValue$ and then add the costs for computing $\theta(\cdot)$ for a whole sequence.

Lemma 6. *The runtime complexity of the computation of a single plausibility $\theta_{q,e}(v)$ for state q, event e and time value v is*

$$\mathcal{O}\left(|B_{q,e}| + \log m\right)$$

with m an upper bound on the number of points sampled and $|B_{q,e}|$ the size of the respective bucket.

Proof. Assuming that we compute $\theta_{q,e}(v)$ for transition $t = (q, e, *)$ with bucket $B_{q,e}$ we must compute the density $\hat{f}_{q,e}(v)$ at point v. As we applied kernel density estimators this computation requires $\mathcal{O}(|B_{q,e}|)$ evaluations of the kernel function. Then we compute the subset of the sampled points $M_{q,e}$ where the density of the x-value is higher than $\hat{f}_{q,e}(v)$: $K_{q,e} = \{(x, y)|(x, y) \in M_{q,e} \wedge \hat{f}_{q,e}(x) \geq \hat{f}_{q,e}(v)\}$. Finally, we divide $|K_{q,e}|$ by $|M_{q,e}|$. In the θ-preprocessing we sorted the points in $M_{q,e}$ according to their density value. Now we exploit this sorting when we compute the cardinality of $K_{q,e}$. We apply binary search to find the first point with the density higher than $\hat{f}_{q,e}(v)$, which can be found in $\mathcal{O}(\log |M_{q,e}|) = \mathcal{O}(\log m)$. Knowing the index of this point and the cardinality of $M_{q,e}$ we also know the number of points that are contained in $K_{q,e}$, hence we know $|K_{q,e}|$.

Summing up, the computation costs of $\theta_{q,e}(v)$ for a single time value v are:

$$\theta_{q,e}(v) \hat{=} \mathcal{O}(|B_{q,e}| + \log m) \tag{4.13}$$

■

Now, we can formulate the runtime of the probability/plausibility computation for a whole sequence.

Theorem 7. *The runtime complexity of the probability/plausibility computation for a timed sequence* $s_t = (e_1, v_1), \ldots, (e_l, v_l)$ *is*

$$\mathcal{O}\left(l \cdot (|B_{max}| + \log m)\right)$$

with m *an upper bound on the number of points sampled and* $|B_{max}|$ *an upper bound on the number of points in one bucket.*

Proof. Assuming that $s_t = (e_1, v_1), \ldots, (e_l, v_l)$ leads to state visits $(q_1, \ldots, q_l)$ we sum up the runtime for every single time value plausibility computation $\theta_{q_i,e_i}(v_i)$ from Lemma 6 (the event probability can be accessed in constant time from the underlying PDFA):

$$\begin{aligned}
\text{probability/plausibility computation} \hat{=} &\mathcal{O}\Big(\sum_{i=1}^{l} \underbrace{\theta_{q_i,e_i}(v_i)}_{\text{Lemma 6}}\Big) \\
= &\mathcal{O}\Big(\sum_{i=1}^{l} \left(|B_{q_i,e_i}| + \log m\right)\Big) \\
= &\mathcal{O}\Big(\sum_{i=1}^{l} \left(|B_{max}| + \log m\right)\Big) \\
= &\mathcal{O}\Big(l \cdot \left(|B_{max}| + \log m\right)\Big) \qquad (4.14)
\end{aligned}$$

■

Given the runtime from Theorem 7 for a timed sequence s_t, extracting k features using the feature extractor presented in Section 4.4.2.1 and expressing the runtime of the classification as variable C, the runtime complexity of the anomaly detection algorithm AmAnDA (Algorithm 7) is:

$$\text{anomaly detection} \hat{=} \mathcal{O}\Big(\underbrace{l \cdot (|B_{max}| + \log m)}_{\text{prob./plaus. comp. (Theorem 7)}} + \underbrace{(l \cdot k)}_{\text{feature extr. (Proposition 5)}} + C\Big) \qquad (4.15)$$

Classifier Training Runtime Now we will shortly describe the impact on the runtime of the chosen classifier. If we choose a classifier that requires training, we have to insert an additional step after the ProDTTAL-time-learning (Algorithm 5). As described earlier, after applying ProDTTAL we transform and extract features from the input S_t^+ with a feature extractor and gain a set of vectors $\mathbf{X}^+$ with $|\mathbf{X}^+| = |S_t^+|$. This transformation is similar to the anomaly detection, except that every vector $\mathbf{x} \in \mathbf{X}^+$ is not classified but instead used as input to the classifier. Hence, we run a slightly adapted version of the anomaly detection for the input S_t^+ with the following runtime:

Theorem 8. *The runtime complexity of transforming a set of positive timed sequences S_t^+ into a set $\mathbf{X}^+$ containing vectors of length k is*

$$\mathcal{O}\left(|S_t^+| \cdot l_{max} \cdot (|B_{max}| + \log m + k)\right)$$

with l_{max} the length of the longest sequence in S_t^+, m an upper bound on the number of points sampled, and $|B_{max}|$ an upper bound on the number of points in one bucket.

Proof. We have to compute the lists of probabilities/plausibilities $P_{\mathbb{E}}$ and $P_{\mathbb{T}}$ (cf. Theorem 7) for every sequence in the input and extracted k features from the two lists (cf. Proposition 5). Hence, we obtain the following runtime (with the sequence length l upperbounded by l_{max}):

$$\begin{aligned}&\mathcal{O}\Big(|S_t^+| \cdot \Big(\underbrace{l_{max} \cdot (|B_{max}| + \log m)}_{\text{prob./plaus. comp. (Theorem 7)}} + \underbrace{(l_{max} \cdot k)}_{\text{feature extr. (Proposition 5)}} \Big)\Big)\\ =&\mathcal{O}\left(|S_t^+| \cdot l_{max} \cdot (|B_{max}| + \log m + k)\right) \qquad (4.16)\end{aligned}$$

■

Theorem 8 solely represents the runtime complexity for the transformation from S_t^+ to a vector set $\mathbf{X}^+$. Depending on the chosen classifier the runtime complexity for training may e.g. be $\mathcal{O}(|\mathbf{X}^+|^2)$ (linear SVM) or constant (Parzen window). For testing whether a vector is an anomaly, for the linear SVM the runtime depends on the support vectors chosen in the training phase, and the Parzen window requires $\mathcal{O}(|\mathbf{X}^+|)$ many evaluations of the chosen kernel function.

4.5 Anomaly Detection in a Two-/Multi-class Setting

In this section we will briefly discuss whether and how the presented one-class anomaly detection algorithm can be applied in a two- or multi-class setting. In the two-class (or binary) setting positive and negative samples (normal and abnormal) samples are presented to the classifier in the training and the test phase. The multi-class setting extends this setting to more than two classes.

4.5.1 Two-class Setting

Usually, for feature vectors a two-class problem is easier to solve than a one-class problem because the two-class classifier can learn a separation between the two classes. In the one-class setting the classifier can only try to fit a boundary for the positive class, maybe neglect some of the training samples because of noise in the training data, and then hope that the boundary is not too tight or too loose.

As already presented in Section 3.3.3.1 the RPNI algorithm can infer a PDFA from positive and negative (untimed) sequences. For timed sequences, the RTI algorithm can even infer a PDRTA from positive and negative timed sequences[11]. Nevertheless, an anomaly detection algorithm for PDRTAs was not presented, so only a naive

[11] RTI+ is an adaptation of RTI from the two- to the one-class problem.

classification approach of checking whether a path exists for a given test sequence is feasible.

Our approach of learning PDTTAs from positive data and using feature extractors and one-class classifiers cannot be easily adapted to the two-class setting. At first sight, it seems straight forward to exchange the one-class algorithms by a two-class approach. Obviously, for learning the PDFA (the event structure) we could exchange e.g. ALERGIA or MDI with RPNI. The feature extraction could stay the same and instead of using a one-class classifier we could apply a two-class vector-based classification algorithm like an SVM, a neural network, etc.

But when doing so, we face a problem in the training phase: For an abnormal sequence we do not know whether it is abnormal because of its event structure or because of its time values. If we omit the time values and do not change the labels, we may present a sequence s_e to RPNI that is abnormal because of timing behavior but normal regarding its event structure. We may also present the same event sequence s_e as normal because its timing behavior is normal, too. Hence, RPNI is confused because the same sequence s_e may occur twice: once as normal and once as abnormal.

We could circumvent this issue if we introduce additional information whether an abnormal timed sequence is abnormal because of its event structure or its timing behavior. If it was abnormal because of the event structure we would present it to RPNI as a negative sample and would not present it to the vector-based classifier. If it was abnormal because of the timing behavior (and not because of the event structure) we would present it to RPNI as a normal sample and would present it as an abnormal sample to the vector-based classifier. In this way we would feed the respective algorithms only with those samples which are really distinguishable based on the information presented to them.

4.5.2 Multi-class Setting

Adapting our automaton-based approach of classifying length-varying timed sequences to a multi-class setting is not straightforward because we do not deal with anomaly detection or binary classification any more but with labeling a timed sequence with a multi-class label. An automaton model can accept or reject a sequence by traversing transitions starting in the initial state. Acceptance and rejection of sequences can be regarded as the two classes from the binary classification. For the multi-class setting one would need a timed automaton model that does not only accept or reject sequences but assigns multi-class labels to the sequences. To the best of our knowledge such a multi-class automaton model for length-varying timed sequences does not exist. If a multi-class automaton would exist, we still lack an efficient algorithm that can infer a timed multi-class automaton model.

On the other hand if we could solve the binary classification problem for length-varying timed sequences, we could use k different binary classifiers in a one-vs-all fashion. Every classifier would predict whether a given timed sequence belongs to a class $i \in \{1, \ldots, k\}$ or not.

In [ISS00] Ishiguro, Sawada, and Sakano propose an algorithm for assigning multi-class labels to untimed fixed-length sequences using an adapted version of an AdaBoost classifier [FS95]. It may be possible to adapt this approach to deal

with timed sequences but adapting to the length variation of the sequences is not straightforward. As this thesis focuses on one-class classification the mentioned ideas are beyond the scope of this thesis, but may be used as a possible starting point to adapt our approach to the two-/multi-class setting.

Summary

In this chapter, we presented the new automaton model for timed sequences Probabilistic Deterministic Timed Transition Automaton (PDTTA) (Section 4.2), the corresponding algorithm that learns PDTTAs from positive data (Probabilistic Deterministic Timed Transition Automaton Learning (ProDTTAL); Section 4.3) and the Automata-based Anomaly Detection Algorithm (AmAnDA) for detecting anomalies with timed automata (Section 4.4). We concluded with an outlook how the approaches could be transferred to a two-/multi-class setting (Section 4.5).

The PDTTA models sequences enriched with time information, but opposed to current automaton models, the time is modeled more granularly from a local point of view. For modeling the event data, we apply state-of-the-art Probabilistic Deterministic Finite Automaton learning algorithms, e.g. ALERGIA or MDI. The time values are modeled using Kernel Density Estimations (KDEs) as a good approximation of unknown time value distributions. These KDEs are transformed into plausibility functions using Monte Carlo integration. The plausibility functions express how plausible the time values of unknown sequences are.

The learning algorithm ProDTTAL learns a PDTTA from positive sequence data (Section 4.3.1). Its runtime complexity is polynomial in the size of the training set (and some other factors; Section 4.3.2) and under some assumptions it converges in the limit to the underlying model (Section 4.3.3). In Section 4.3.4 ProDTTAL is compared to other state-of-the-art timed automata learning algorithms and strengths and weaknesses of the respective approaches are pointed out.

We continue with the anomaly detection using timed automata (Section 4.4), present the Automata-based Anomaly Detection Algorithm (AmAnDA) (Section 4.4.2) that can detect anomalies given various timed automata and analyze its runtime complexity (Section 4.4.2.3). For an unknown sequence, AmAnDA traverses the automaton model and stores all probabilities and plausibilities in two separate lists. Then, it uses a feature extractor to extract features from these two lists. These features are handed over to a one-class classifier that decides whether the unknown sequence is an anomaly.

Finally, we argue why it is not straightforward to transfer our approach to a two-/multi-class setting and describe hurdles that would arise (Section 4.5). In the two-class setting, we only know that a sequence is normal or abnormal, but we do not know whether it is an anomaly because of its event or time value. However, this information would be needed, e.g. to build the correct automaton structure because a time-abnormal sequence still contains a valid event sequence. The transformation into a multi-class approach would be even harder, but could be implemented in a one-versus-all manner.

In the next chapter, we describe how sequence-based anomaly detection approaches can be evaluated systematically given an anomaly-free data set.

5 Anomaly Detection Evaluation

In this chapter, we deal with the challenge of how to evaluate approaches that try to solve the sequence-based anomaly detection problem in Discrete Event System (DES). First, we review possible performance metrics (Section 5.1). In Section 5.2 we identify five types of anomalies that can occur in DES (Section 5.2.1). Then, we introduce a random anomaly insertion (Section 5.2.2) and a model-based anomaly insertion approach using a Timed Prefix Tree Acceptor (TPTA) (Section 5.2.3). Both approaches can insert the identified anomaly types into an anomaly-free data set to obtain a data set that contains anomalies and, hence, can be used for the experimental evaluation. Furthermore, we discuss the influence of the anomaly rate on the anomaly detection performance. Finally, we propose an approach how to evaluate the scaling of algorithms that solve the timed sequence anomaly detection problem (Section 5.3).

Four types of anomalies and the random anomaly insertion have already been published in [Kle+14]. The rest of this chapter has not been published and was developed during this thesis.

5.1 Performance Metrics

In this section, we explain how to measure the performance of the different approaches. In the field of grammatical inference, where only the inference algorithm is of interest, performance is measured by sampling sequences from a generative model, inferring another model from these sequences and comparing the resulting model and the generating model. This comparison does not state anything about the quality of the model in anomaly detection. In anomaly detection the quality of an approach depends on the quality of the model as well as on the anomaly detection algorithm. As this is a one-class classification problem, we will present different classification metrics.

Typical measures for performance evaluation of classifiers in the two-class setting are *true positives (tp), false positives (fp), true negatives (tn)* and *false negatives (fn)*. True positives are the number of times the classifier predicted the positive class when the presented sample was indeed positive. False positives are the number of times the classifier predicted the positive class while the presented sample was labeled with the negative class. True negatives and false negatives are defined analogously. We let anomalies be the positive class and normal samples the negative class.

Based on these numbers other metrics like precision, recall, F-Measure (or F_1 score), accuracy and many more are defined in literature. In this thesis we mainly focus on F-Measure and the Matthews Correlation Coefficient (MCC) (Eq. (5.6)).

The *precision* is the proportion of samples that are correctly labeled as positive out of all samples that are labeled as positive.

$$\text{precision} = \frac{tp}{tp + fp} \tag{5.1}$$

It represents how far the method can be trusted if it labels a sample as positive (anomaly). A precision of one indicates that every anomaly that was labeled as such was indeed an anomaly. The precision does not state anything about how many anomalies were actually detected.

Therefore, the *recall* (also *sensitivity*) is defined as the ratio between correctly labeled positives and all positives:

$$\text{recall} = \frac{tp}{tp + fn} \tag{5.2}$$

It indicates how often a method misses an anomaly (positive) and falsely labels it as normal (negative). A high recall states that many anomalies were detected and thus labeled as anomaly.

A high recall can always be achieved by classifying every sample as anomaly but this strategy results in a low precision. The optimal approach would detect every anomaly (recall = 1) and label every normal sample as normal (precision = 1). The *F-Measure* (also F_1 score) is a suitable metric because it combines precision and recall. It enables to optimize an approach while tuning only a single metric. The F-Measure is the harmonic mean of recall and precision:

$$\text{F-Measure} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{5.3}$$

An F-Measure of one can only be achieved with precision and recall being one. For any other case the F-Measure weights precision and recall equally. For weighting precision over recall or vice versa the F_β score was designed. As we do neither prefer precision nor recall we only employ the F_1. The F-Measure can also be expressed in terms of true and false positives or negatives:

$$\begin{aligned}
\text{F-Measure} =& 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = 2 \cdot \frac{\frac{tp}{tp+fp} \cdot \frac{tp}{tp+fn}}{\frac{tp}{tp+fp} + \frac{tp}{tp+fn}} \\
=& \frac{\frac{2 \cdot (tp)^2}{(tp+fp) \cdot (tp+fn)}}{\frac{tp \cdot (tp+fn)}{(tp+fp) \cdot (tp+fn)} + \frac{tp \cdot (tp+fp)}{(tp+fn) \cdot (tp+fp)}} = \frac{\frac{2 \cdot (tp)^2}{(tp+fp) \cdot (tp+fn)}}{\frac{tp \cdot ((tp+fn)+(tp+fp))}{(tp+fn) \cdot (tp+fp)}} \\
=& \frac{2 \cdot (tp)^2}{(tp + fp) \cdot (tp + fn)} \quad \cdot \quad \frac{(tp + fn) \cdot (tp + fp)}{tp \cdot (2 \cdot tp + fn + fp)} \\
=& \frac{2 \cdot (tp)^2}{tp \cdot (2 \cdot tp + fn + fp)} \\
=& \frac{2 \cdot tp}{2 \cdot tp + fn + fp}
\end{aligned} \tag{5.4}$$

Equation (5.4) shows that the F-Measure is biased by the true positives (tp) and

the true negatives (tn) do not even influence the result. Moreover, the definition of which class is positive or negative heavily alters the outcome. As a rule of thumb one should always define that class as the positive one for which less samples exist.

The *accuracy* does not incorporate the difference between the class definition. It is defined as the fraction of correctly classified instances among all instances:

$$\text{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn} \tag{5.5}$$

However, the accuracy is biased if the two classes (positives and negatives) are of different size. Hence, the *Matthews correlation coefficient* (MCC; also *Phi coefficient*) can be used. It is related to the χ^2-test [Pla83] for two classes and does not depend on the definition of the classes or their sizes [Pow11]. The Matthews correlation coefficient is defined as:

$$\text{MCC} = \frac{tp \cdot tn - fp \cdot fn}{\sqrt{(tp + fp) \cdot (tp + fn) \cdot (tn + fp) \cdot (tn + fn)}} \tag{5.6}$$

While the F-Measure ranges from zero to one where one means perfect classification and zero means no true positive, the Matthews correlation coefficient ranges from -1 to $+1$. The result is -1 if there are no correctly classified samples, 0 if the classifier is not better than a random predictor and 1 if the classifier is perfect.

The *area under the receiver operating characteristic curve* (also (ROC-)AUC) is often used to measure the performance of a binary classifier. It describes the classifiers performance as a function of a single decision threshold. The F-Measure, accuracy and MCC can only be computed for one particular value of the decision threshold. But in our scenario we may be faced with more than one decision threshold or at least with more than one parameter that influences the decision. Then, the computation of the AUC becomes non-trivial. Moreover, the AUC weighs misclassification for different models differently [Han09].

So far, we have presented a selection of commonly used performance metrics with their strengths and weaknesses. Table 5.1 shows the presented performance metrics for the same experiment result for an imbalanced data set depending on which class is defined as the positive one. Even though the result is not very good (for one of the classes only 50% were classified correctly) the accuracy is large and the F-Measure is large in one case while low in the other one. The MCC is quite small for both cases which better reflects the result. In the remainder of the evaluation we will mostly focus on the Matthews correlation coefficient (MCC) because it does not depend on the definition which class is positive and negative and is not biased towards one of the classes.

Apart from the metrics that measure the quality of the classification performance other metrics are important as well. The best algorithm in terms of classification performance is useless if the training may take years and the class prediction of one sample a whole day. Moreover, the algorithm should run on commodity hardware and should not need a server farm to operate.

Hence, we will also measure the execution time for training and prediction as well as the number of states in the model and—related to the number of states—the memory consumption. The worst case runtime complexity was already analyzed

Table 5.1: Performance values for same experiment result with different definitions of the positive/negative class.

tp	tn	fp	fn	F-Measure	Accuracy	MCC
900	5	100	5	0.94	0.90	0.13
5	900	5	100	0.09	0.90	0.13

(cf. Sections 4.3.2 and 4.4.2.3), but sometimes the worst case runtime complexity heavily differs from the experimental runtime. We will investigate whether such a difference exists for the particular algorithms. We are aware that the experimental metrics depend on the computer, the operating system, the programming language etc. Nevertheless, if all algorithms are written in the same programming language and the evaluation is performed on the same computer, the metrics are comparable.

5.2 The Curse of One-class Evaluation

In the usual classification setting the training and the test data contain samples from all classes that shall be classified. If only one data set exists, this data set is e.g. split into a training and a test set. In the one-class setting, the training data only contains samples from the positive class. The training of the classifier can be performed with samples from the positive class, but when it comes to testing (or evaluation) negative samples must be at hand to measure the performance of the classifier. Without negative samples it is not possible to decide whether the classifier correctly labeled a sample as positive or negative. Usually, there do not exist any negative samples[1]. Therefore, it is harder to measure the performance of one-class classifiers. Assuming that a real-world data set containing only positive samples from a technical system exists, several approaches to produce negative samples (anomalies) are possible:

(a) *Real-world anomalies.* Examples are derived from monitoring the technical system for a long time, hoping to capture an anomaly. This anomaly has to be labeled as anomaly by an expert.

(b) *Physically simulated anomalies.* Examples are derived from the reproduction of known anomalies under laboratory conditions using a real-world copy of the technical system.

(c) *Model-based simulated anomalies.* Examples are derived from the reproduction of known anomalies by manipulating a (learned) behavior model. This manipulation has to be performed by an expert, too.

(d) *Artificially generated anomalies.* Examples are generated by some probabilistic or random process.

[1] If there were many negative samples, the problem would not be a one-class problem.

In [Kle+14] we state that it is most often impossible to acquire real-world or physically simulated anomalies. The first ones are hard to find and the latter ones are very expensive (if possible at all) because a copy of the technical system has to be built and then modified or even damaged. Before we describe a systematical approach of generating random- and model-based anomalies, we first describe which types of anomalies can occur in discrete event systems in the next section.

5.2.1 Anomalies in Discrete Event Systems

In [Kle+14] we presented four types of anomalies that can occur in discrete event systems. During our subsequent research we identified a fifth type of anomaly—the *Anomalous ending*. In this section we present the already known and the new type of anomaly. Let a sample path be one possible pass through a model.

1. *Anomalous event.* Given a sample path, a single event is anomalous if it causes a transition to an irregular system state. In a stochastic setting, an event can also be anomalous if its probability of occurrence is lower than some anomaly threshold. An anomalous event could be caused by an action that is not allowed in the current state.

2. *Anomalous sample path.* A sample path is anomalous if every single event is normal but the aggregation of the events' individual occurrence probability is lower than some anomaly threshold. This anomaly type only applies to a stochastic setting. In a real-world system an anomalous sample path may indicate a manipulation which forces the system to use allowed but rarely occurring events.

3. *Anomalous event timing.* Given a sample path, a single event e has an anomalous timing if the time gap between e and its directly preceding event is outside the regular range. If a single component works faster or more slowly, this change may result in an anomalous event timing.

4. *Anomalous sample path timing.* The timing of a sample path is anomalous if every single event has a normal timing but the aggregation of the events' individual timing is lower than some anomaly threshold. An anomalous sample path timing may be caused by a slowdown of the whole system whereas the slowdown of every component is still in the acceptable range.

5. *Anomalous ending* (new). Given a sample path, the ending is anomalous if the sample path ends in a state that is not a final state. In a stochastic setting, an ending can also be anomalous if sample paths ended in the final state very rarely. If the logging module in a system does not operate correctly, sequences may not contain every event and thus end in the wrong state.

In the next section we describe how we insert anomalies artificially (cf. [Kle+14]), followed by a new model-based anomaly insertion approach (Section 5.2.3).

5.2.2 Random anomalies

For inserting anomalies randomly, we split the data into a training, validation and test set and for every type of anomaly (cf. Section 5.2.1) we modify normal sequences in the validation and test set only. The training set is not modified because we operate in a one-class setting in which the training set may only contain positive sequences. The modifications are performed in the following way[2]:

1. *Anomalous event*: Replace a randomly chosen event e from the timed event sequence by another event e' from the alphabet.

2. *Anomalous sample path*: Replace the timed event sequence by another sequence that occurs relatively rarely in the validation and test set respectively.

3. *Anomalous event timing*: Modify a single time value in the timed event sequence.

4. *Anomalous sample path timing*: Modify every time value in the timed event sequence.

5. *Anomalous ending*: Cut the sequence at some point and discard the right part.

As these anomalies are very artificial it is hard to interpret the results an anomaly detection algorithm achieves. Hence, we also apply model-based simulations with anomalies inserted into the model to obtain more interpretable results. In the next section we describe this procedure.

5.2.3 Model-based simulated anomalies

Instead of randomly interspersing anomalies into normal sequences, we infer a Prefix Tree Acceptor (PTA) with timing information as model and modify the resulting tree. We call the Prefix Tree Acceptor (PTA) with timing information *Timed Prefix Tree Acceptor (TPTA)* because it is a PTA (containing event probabilities) enriched with the a time density function (similar to τ from the Probabilistic Deterministic Timed Transition Automaton (PDTTA) (cf. Section 4.2)). We chose to employ the TPTA because it models the event data perfectly. The time data itself is not modeled optimally because a TPTA cannot model time-dependent substructures (no transition split based on time values). To circumvent this drawback we apply the preprocessing of the Bottom-Up Timed Learning Algorithm (BUTLA)[3] before inferring the TPTA. Like in the PDTTA we use kernel density estimators (KDEs) for modeling the time values in the TPTA as a reasonable approximation of the time value density function.

Opposed to a PDTTA, a TPTA does not generalize or merge any states and the number of states may become very large. This largeness and missing generalization may become a problem for anomaly detection, but for the use case of simulating

[2]We described only four anomaly types in [Kle+14], but now added the fifth one (anomalous ending).

[3]For the KDE in the preprocessing we utilize the original KDE formula (Eq. (2.9)) instead of the specialized BUTLA variant because we experimentally found that the specialized BUTLA variant (Eq. (2.24)) does not result in a good event split.

anomalies the size of the model does not matter[4]. In the following we describe how a TPTA is inferred from data.

5.2.3.1 TPTA Inference

Given a positive data set S_t^+ containing only positive timed sequences, we want to build a model from which we can sample positive and negative timed sequences obtaining validation and test sets $S_t^{+/-}$. The inference of a TPTA from S_t^+ is similar to inferring a PDTTA with BUTLA preprocessing except that we omit the merge step:

1. Preprocess S_t^+ to split events based on time behavior.
2. Build a PTA A' from S_t^+ omitting the time values (cf. Section 2.4.1.2).
3. Apply ProDTTAL-time-learning (Algorithm 5) to A'.

ProDTTAL-time-learning (Algorithm 5) can be applied because it accepts every Probabilistic Deterministic Finite Automaton (PDFA) as input and a PTA is a special kind of PDFA. Figure 5.1a shows an excerpt of a TPTA with four states and three events. The state q_3 is the only visible accepting state. The transition probabilities as well as an event split are left out for simplicity.

Now we will describe how we modify the obtained TPTA to generate anomalies.

5.2.3.2 Anomaly Insertion

Given S_t^+ we infer a TPTA, copy it and insert one type of anomaly into every copy as described in the remainder of this section. Figure 5.1 shows an excerpt of this process. As we elaborated five types of anomalies we obtain five different TPTAs to ensure that anomalies do not overlap in any way. Hence, an abnormal sequence is always abnormal because of a single manipulation in the respective TPTA and not because of a mixture of different anomalies. We employ the modified TPTAs only for sampling abnormal sequences and sample the normal sequences from the initial (unmodified) TPTA (Fig. 5.1a). For every type of anomaly the initial TPTA is modified as follows:

1. *Anomalous event*: On every tree level of the TPTA we choose a state uniformly at random and modify the symbol of one outgoing transition to another symbol that does not occur in the selected state (Fig. 5.1b). If there is only one outgoing transition on the current level for all states we do not modify anything because we want normal and abnormal transitions on every level.
2. *Anomalous sample path*: We compute the likelihood of all event paths that occur in the TPTA and select the k least probable ones. These paths are labeled as anomalous and are also removed from the TPTA that we apply to

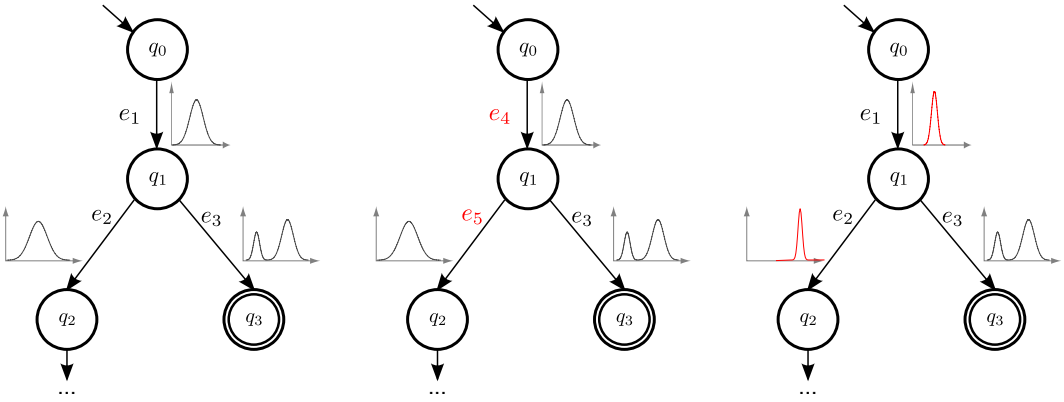
[4]Compact and general models are needed for the anomaly detection on embedded monitoring devices where memory and computational power may be very limited. For the purpose of simulating anomalies devices with a huge amount of memory and computational power can be used.

sample normal sequences. If we do not remove these paths they may be labeled as normal when sampled for the training data set.

3. *Anomalous event timing*: Similar to the anomalous event we choose an outgoing transition uniformly at random from the states on every tree level and heavily increase or decrease the outcome of its time distribution when sampling time values from it (Fig. 5.1b).

4. *Anomalous sample path timing*: Similar to the anomalous sample path we compute the likelihood of all event paths that occur in the TPTA. To avoid overlapping of the respective sample path anomalies we select the k most probable sample paths. On these paths every outcome of the time distribution at every transition is slightly increased or decreased. Opposed to the sample path anomaly we do not remove these paths from the initial TPTA as they contain different time values.

5. *Anomalous ending*: On every tree level we select all states that are not final states, i.e. states with zero final state probability. We pick one of those states uniformly at random and add a small, but varying final state probability. Then we adjust the probabilities of outgoing transitions such that they sum up to one again.



(a) Normal TPTA.

(b) Event e_1 (e_2) has been exchanged with e_4 (e_5) (red).

(c) Two density functions were replaced with the red ones.

Figure 5.1: Excerpt of normal and modified TPTAs with anomalies of type one and three.

Given these modifications we can sample the five types of anomalies. We start at the root node of the respective TPTA and repeatedly choose a transition according to the transition probabilities. Eventually, we choose a final transition and stop. A sequence from an abnormal TPTA is labeled abnormal if an abnormal transition on the sample path from the root node to the final state was chosen. With a sequence labeled as normal or abnormal we can evaluate the performance of a model by computing different metrics on the decisions made by the model.

5.2.4 Anomaly Rate

Given ways to insert anomalies we now discuss how many anomalies shall be inserted into the test set. If we train a classifier on the same (anomaly-free) training set, the resulting classifier is always the same. Hence, we assume that the classifier labels the same percentage of anomalies correctly or incorrectly, i.e. the true/false positive and true/false negative rates are constant. As usual, we define anomalies as positive class. We exemplarily show the respective metrics for two different classifiers and different anomaly insertion rates. Classifier A predicts anomalies with a true positive rate ($\frac{tp}{tp+fn}$) of 70% and a true negative rate ($\frac{tn}{tn+fp}$) of 90%, whereas classifier B predicts anomalies with a true positive rate of 60% and a true negative rate of 80%. Thus, classifier A is better than B in terms of true positive and true negative rate and should always achieve higher performance metrics. Table 5.2 shows how the performance metrics F-Measure, Accuracy and MCC (Section 5.1) would change for a test set containing 1000 test samples, classifiers A and B and different anomaly insertion rates. Classifier A is always better than classifier B for all metrics. With increasing anomaly rate the F-Measure increases whereas the accuracy drops. The MCC first rises and drops again with the highest MCC at 50% of inserted anomalies. As the ordering of the classifiers is not influenced by the anomaly insertion rate, we fix the anomaly insertion rate for the remainder of this thesis to 50%.

Table 5.2: Performance values for same experiment result with different definitions of the positive/negative class.

% anomalies	# anomalies	classifier	tp	tn	fp	fn	F1	Acc.	MCC
1 %	10	A	7	891	99	3	0.12	0.90	0.19
1 %	10	B	6	792	198	4	0.06	0.80	0.10
5 %	50	A	35	855	95	15	0.39	0.89	0.39
5 %	50	B	30	760	190	20	0.22	0.79	0.21
10 %	100	A	70	810	90	30	0.54	0.88	0.49
10 %	100	B	60	720	180	40	0.35	0.78	0.28
50 %	500	A	350	450	50	150	0.78	0.8	0.61
50 %	500	B	300	400	100	200	0.67	0.7	0.41
75 %	750	A	525	225	25	225	0.81	0.75	0.52
75 %	750	B	456	200	50	294	0.73	0.66	0.35

In the next section we describe how the developed anomaly insertion approaches can be used to experimentally evaluate the scaling of timed sequence-based model learners.

5.3 Experiment Design / Scaling of Experiments

In this section we will focus on how to design fair and comparable experiments while increasing the size of the data or its complexity. As already stated in the description of ALERGIA (cf. Section 2.4.2.1), the runtime experiments for ALERGIA [CO94] are biased because the size of the training set $|S_e^+|$ was increased by sampling more sequences from an automaton although the runtime of ALERGIA does not directly depend on the training set size $|S_e^+|$ but on the size of the resulting PTA ($|PTA(S_e^+)|$). Thus, we may observe heavily differing results for the same size of the training set $|S_e^+|$. If an algorithm does not create a PTA or if the algorithm computes how well the model fits the training data, the runtime depends on the size of the training set $|S_e^+|$. We must take into account that sampling more sequences from a single model increases the runtime of such an algorithm.

Hence, we must not only care about the size of the training set $|S_e^+|$ when comparing algorithms and their runtime, but we must also care about how the increase of $|S_e^+|$ is accomplished, e.g. by only sampling more sequences from a single model or by sampling sequences from a small model and a bigger model. Different *scaling* approaches of changing an existing generative model are possible to increase the runtime of the algorithms:

- *Add event-based transition*: We add new event-based transitions to the existing model without increasing the number of states or changing the alphabet. In an automaton we could add transitions to a state if it does not contain outgoing transitions with all symbols. Event-based means that we add a transition with a new event.

- *Add time-based transition*: We add a transition similar to the event-based case. Instead of adding a new symbol we add new timing behavior. This is only possible if the model we sample from can distinguish between transitions with the same symbol but different time behavior.

- *Add states*: We add new states and for states that have more than one outgoing transition we set the target state of one transition to the newly created state. In this way we do not increase the number of transitions because we only bend existing transitions, but we must make sure that every state is still reachable.

- *Enlarge the alphabet*: We introduce new symbols without changing the number of states or transitions of the existing model. For some transitions we exchange the existing symbol with a new one and simultaneously ensure that every symbol is used at least for one transition.

- *Bigger training set*: We sample more sequences from the same generative model without any changes to the model.

Every of the presented approaches increases only one aspect that may increase the runtime but a combination of the approaches is possible. Such a combination may increase the runtime but it will not be clear which change is responsible for the runtime increase.

When evaluating the scaling of the algorithms (cf. Section 6.6) we will apply every of the presented scaling approaches. After describing how we implemented the scaling approaches, we will present the expected runtime increase and compare it with the observed runtime for every investigated algorithm .

Summary

In this chapter, we reviewed classification performance metrics and their strengths and weaknesses with the Matthews Correlation Coefficient (MCC) being a good performance metric for anomaly detection. Then, we presented five types of anomalies that can occur in Discrete Event System (DES), followed by the random and the model-based anomaly insertion approach to insert anomalies into anomaly-free timed sequence data. The random anomaly insertion approach picks normal sequences, modifies these sequences according to the anomaly type and changes the label to abnormal. The model-based approach builds a Timed Prefix Tree Acceptor (TPTA) from the anomaly-free data and modifies the resulting tree with respect to the anomaly type. After discussing the influence of the anomaly rate, we presented a systematic approach how to investigate the scalability of timed sequence-based model learners in the last section. Therefore, we proposed to sample sequences from an automaton, systematically increase the number of transitions, states, sampled sequences or the alphabet, and investigate how different learners behave under these circumstances.

In the next chapter, we implement the proposed anomaly insertion approaches to experimentally evaluate different algorithms (mainly ProDTTAL,BUTLA and RTI+) on synthetic and real-world data.

6 Experimental Evaluation

In this chapter, we present the experimental evaluation we conducted on synthetic data and real-world data gathered on a public Automated Teller Machine (ATM). For every type of data, we ran the algorithms Probabilistic Deterministic Timed Transition Automaton Learning (ProDTTAL), Real-Time Identification from Positive Data (RTI+) and Bottom-Up Timed Learning Algorithm (BUTLA) (and the two baselines ALERGIA and the Timed Prefix Tree Acceptor (TPTA)) on the respective data. To the best of our knowledge, neither of the three algorithms have been compared with each other up to now. First, we present how we tuned the hyperparameters of the respective method (Section 6.1), followed by the default experiment setup (Section 6.2). Then, we present preliminary experiments (Section 6.3) and more thorough experiments (Section 6.4)—both on synthetic data. After evaluating the approaches on real-world data (Section 6.5), we conclude the chapter with implementing the scaling approach from the previous chapter and giving some insights on the runtime of the analyzed approaches (Section 6.6). All in all, we evaluated approximately 4.5 million different algorithm configurations (parameter settings), resulting in a duration of about 3 CPU years.

6.1 Hyperparameter Tuning

All used algorithms require the specification of various hyperparameters. The right configuration of these hyperparameters is crucial for the algorithms to work well. Finding the best algorithm configuration can be very difficult. Instead of defining a lot of default values and applying a grid search over the remaining hyperparameter space, we use a state-of-the-art hyperparameter configuration algorithm that we present in the following section.

SMAC

Sequential Model-based Algorithm Configuration (SMAC) [1] [HHL11] is an algorithm to automatically find a good configuration for a given parametric algorithm $\mathcal{A}$ like RTI+, BUTLA or ProDTTAL. We will only present the idea of SMAC. For further details, we refer to [HHL11].

SMAC requires the following inputs:

- An algorithm $\mathcal{A}$.
- A set of data sets to train and test $\mathcal{A}$ on.

[1] Freely available at `http://www.cs.ubc.ca/labs/beta/Projects/SMAC/` under AGPLv3.

- A metric m that measures the performance of $\mathcal{A}$ on a data set.
- A description of the hyperparameters of $\mathcal{A}$, their possible values and the domain they stem from (e.g. natural number, categorical, ordinal, ...).
- A default configuration for the hyperparameters.

SMAC builds a model to predict the performance of $\mathcal{A}$. It builds a model with the default configuration, then iteratively chooses and executes one of the most promising configurations in terms of performance improvement and refines the model based on the result of the executed configuration.

As model, SMAC applies a random forest [Bre01]. The idea of this random forest is to train k regression trees on a random subsample of the training data and during training of each tree, at every inner node, only a subset of the features is considered eligible to split the tree. Given a random forest, for predicting a performance value for an unknown algorithm configuration the mean over the outputs from all trees is returned. The variance is also computed, as it is required in the next step.

To choose a promising configuration, the *expected improvement* (EI) [JSW98] is applied. It is a metric that utilizes the mean and variance (as uncertainty measure) to express the expected improvement for a given configuration. SMAC computes the EI for all previous configurations and for the highest of them it performs a local search in the hyperparameter space starting at the respective configuration. In the local search the EI is calculated with the predicted mean and variance from the random forest for every candidate configuration. If no configuration with larger EI can be found in the neighborhood of a given configuration, the search is terminated. The found configuration is the candidate to be executed next.

6.2 Experiment Setup

In this section we describe the default setup of our experiments. If not mentioned otherwise every experiment is run as described here.

6.2.1 SMAC Setup

As mentioned in the last section (Section 6.1) all of our experiments are performed with SMAC. In this way me make sure to find a good hyperparameter setting for every algorithm. Usually, SMAC operates in the following way: At first SMAC tries to find the optimal hyperparameter setting on train sets by always computing the performance (in terms of Matthews Correlation Coefficient (MCC)) of the algorithm on the train sets. After that SMAC validates the found setting on validation sets. As we deal with a one-class problem we cannot compute the performance of the found hyperparameters on the training data. Hence, we put the training and validation data into single files and use these files for the training phase of SMAC. In this way we already perform the validation phase during the SMAC train runs. For the SMAC validation, we also build the model on train sets but evaluate its performance on test sets. Therefore, every data file contains normal sequences for training and a mixture of normal and abnormal sequences for validation/testing. For the anomaly creation we usually apply the random and the model-based anomaly insertion approach

(cf. Section 5.2). If we can sample anomalies directly from a modified automaton model, we also do this. As the percentage of anomalies in the test set may be arbitrarily set as long it is the same for all approaches when using the MCC as performance metric we always fix it to (an arbitrarily chosen value of) 50%[2]. We provide ten files for the training phase and ten files for the validation phase of SMAC. Doing so gives SMAC the opportunity to reevaluate a hyperparameter setting on different files. Depending on the size of the data set, we run SMAC for a different amount of time and present this time in the respective section. The final performance value is the mean of the algorithm's performance in terms of MCC on all data files.

To tune the hyperparameters the respective hyperparameter values must be specified before. As every algorithm has a lot of hyperparameters that can be set the values can be found in Appendix A.1.

6.2.2 Default Environment

Some parameters in the experiments are always set to a fixed value if not mentioned otherwise. In this section we will describe which parameters are fixed and why.

6.2.2.1 Hardware

We ran our experiments on computers that all had the same hardware configuration. Every job we ran was able to run on a single dedicated CPU core which it did not share with any other job. Additionally, we limited the amount of memory for every job to 4 GB RAM (by specifying a Java Virtual Machine (JVM) argument). As the computers contained 8 GB RAM and an Intel Core i7-2600 CPU, we usually ran two jobs in parallel on one computer.

6.2.2.2 Feature Extractors

In Section 4.4.2.1 we described how to extract features from a list of probabilities / plausibilities. The features we extract are the following:

1. The length-normalized sequence aggregation.
2. The minimal value of the list.
3. The maximal value of the list.
4. The arithmetic mean of the list.
5. The arithmetic variance of the list.
6. The minimum difference between two succeeding entries.
7. The maximal difference between two succeeding entries.
8. The length of the sequence.

[2]Please note that 50% is an arbitrary value. As we focus on comparing different algorithms the percentage of anomalies should not influence the ranking of the algorithms, but if anything the MCC value (Section 5.2.4).

Usually we evaluate four different feature extractors ranging from a single feature to all features. Ideally, we would have tried all combinations of features, but that would have blown the experiment duration. Thus, we decided to create four feature creator settings as a good trade-off between experiment duration and parameter space covering[3]. The first feature creator only contains the first feature, the second feature creator contains features 1–3, the third contains the features 1–4, and the fourth feature creator contains all features.

6.2.2.3 One-class Classification Algorithms

As our approach for anomaly detection presented in this thesis is modular with respect to the one-class classification algorithm we could evaluate our approach with every possible one-class classification algorithm. We chose some algorithms that we already presented in Section 2.5.1. The following algorithms are used as one-class classification algorithms:

- Threshold on every entry of the feature extractor
- DBSCAN with a single threshold on the distance to the next core point
- k-Means/X-Means/G-Means with a single threshold on the distance to the next cluster center
- One-class SVM (ν-SVM) with ν as the percentage of rejections of samples from the training set (target rejection rate)

6.2.2.4 Anomaly Insertion

We evaluate every approach on different data sets. For the experiments on synthetic data, we insert anomalies directly into the data-generating model. Additionally, we apply our random and model-based anomaly insertion approach for every experiment to insert anomalies into anomaly-free data. All anomaly insertion approaches can insert different types of anomalies that range from easily detectable to very hard. We evaluate every type of inserted anomaly on separately and additionally provide a mixed data set that contains all types of anomalies at the same proportion.

6.2.3 Algorithm Improvements and Implementation Details

In this section we will present which problems can occur with BUTLA and RTI+ and also present improvements for those problems.

6.2.3.1 RTI+

The source code of RTI+ is freely available[4], but the implementation crashes even for simple examples without any error message. Hence, we reimplemented the algorithm. Additionally, we added some minor bug fixes and a small modification, that we also

[3] We chose four feature creators arbitrarily. Any other value that does not blow the experiment duration could have also been chosen.

[4] RTI+ source code available under GPLv3 at `http://www.cs.ru.nl/~sicco/software.htm`.

take into account the probabilities that a sequence ends in a state even although a Probabilistic Deterministic Real-Time Automaton (PDRTA) does not contain final state probabilities. Furthermore, we added the possibility to obtain a list of likelihoods of a given timed sequence for a given PDRTA. To generate the histograms we split the time values into k quantiles with k as a hyperparameter (originally, the histograms have to be user-defined, but Verwer implemented the same functionality). Apart from that we had to interpret how some implementation details are supposed to work (following the description in [VdW10]). Nevertheless, the choices we had to make do not have a big impact in the algorithm according to preliminary experiments.

6.2.3.2 BUTLA

For BUTLA the situation is different compared to RTI+ because we did not find a sound and clear description of the algorithm. In this section we present different problems we encountered when applying BUTLA [Pap16].

KDE Formula As presented earlier (cf. Section 2.4.3.1) the Kernel Density Estimation (KDE) function in [Mai14] is not defined correctly. Hence, we tried to find out the meaning of the KDE while still keeping the variable smoothing factor of 0.05. Figure 6.1 shows the comparison of a KDE with Gaussian kernel compared to all possible meanings of the KDE function in [Mai14] (for more details we refer to [Pap16]). The data represents time values that occur for the same event. The time values were sampled from overlapping Normal distributions with the mean marked as $E(a_i)$. As BUTLA assumes a Gaussian distribution of subevents, it should locate splits (local minima) only between two succeeding subevents. The blue line shows the KDE with Gaussian kernel and bandwidth $h = 10$. It represents the data very well and finds exactly the local minima where the event should be split into subevents (between the mean values of two subevents).

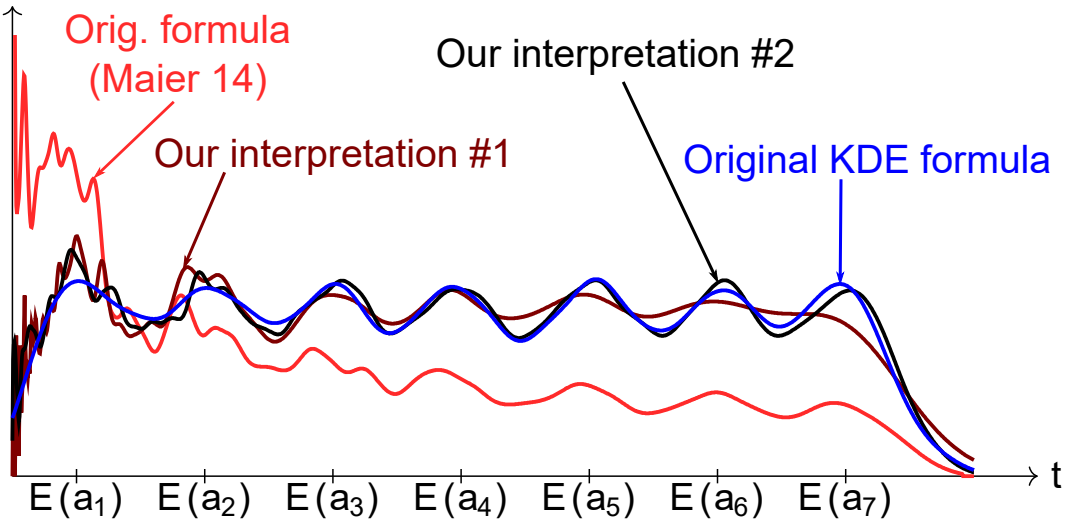


Figure 6.1: Original KDE and the possible interpretations of the KDE described in [Mai14].

The light red line shows the KDE as presented in [Mai14]. The approximation heavily oscillates, especially for low values, and does not correctly model the distribution for higher values. The dark red and black line show the approximation of our interpretation of the correct KDE in BUTLA (cf. Eq. (2.24)). Both approximations heavily oscillate for low values, but model the distributions better than the light red line. Nevertheless, compared to the original KDE (blue line), all possible interpretations of the KDE formula for BUTLA result in too many local minima and thus split the data into too many subevents.

During preliminary experiments with large time values BUTLA crashed for some runs because it split an alphabet with 17 events ($|\Sigma| = 17$) into more than 850 subevents. An alphabet of that size leads to a big prefix tree consuming a lot of memory and CPU time for merging those subevents. Hence, BUTLAs original KDE formula is not feasible, especially when dealing with large time values.

Overlapping Subevents As described in Section 2.4.3.1 BUTLA performs the event split for generating subevents globally. Then every tuple of symbol and time value is mapped to a subevent. As it is assumed that the time values of the subevents follow a Normal distribution these Normal distributions may overlap.

Figure 6.2 shows such an overlap for events a_1 and a_2. We call the overlap the *critical area*. The dotted line indicates the time value at which BUTLA splits the events into two subevents.

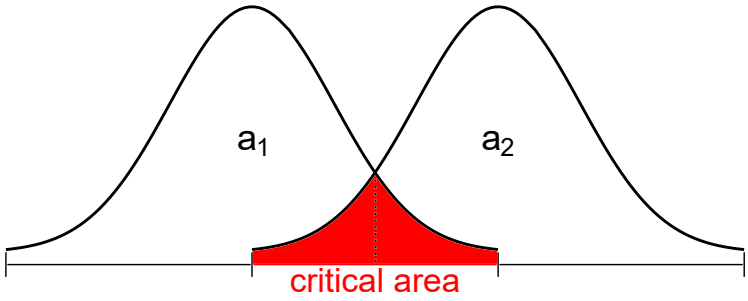


Figure 6.2: Critical area for two overlapping subevents a_1 and a_2.

When encountering (globally produced) critical areas it may happen that events with time values in the critical area may be mapped to the wrong subevent. In Fig. 6.2 the time value around the dotted line may belong to event a_1 or a_2. Hence, some events that are subevents of type a_1 are transformed to a_2 and vice versa. This incorrect mapping induces incorrect subtrees in the prefix tree. In [Pap16] we proposed two possible solutions. The first approach globally isolates critical areas and introduces a separate event (e.g. $a_{1.5}$ between a_1 and a_2) for every critical area. The second approach locally identifies states in which only one of the two subevents occurs. For every such state all time values are mapped to the occurring subevent by locally extending the subevents interval. Figure 6.3 shows two distributions with a critical area. No time value would be definitely mapped to a_2 locally. Hence, the interval of a_1 is locally extended to include the critical area because all time values belong to a_1 or to the critical area.

Additionally, we also implemented BUTLA in a top-down manner and can compare

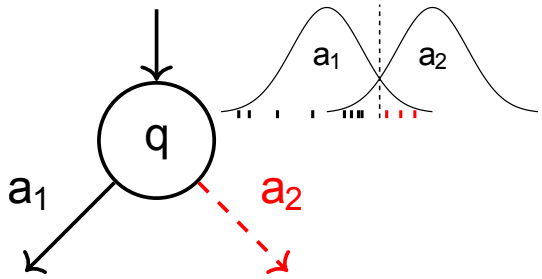


Figure 6.3: Local interval extension based on unused critical area (from [Pap16]).

the numbers of outgoing instead of incoming transitions. In the remainder of this chapter we will evaluate and most often use the improvements of BUTLA. We call these improvements Improved Bottom-Up Timed Learning Algorithm (BUTLA-i). The source of all algorithms we evaluate is implemented in Java and freely available at `https://github.com/TKlerx/SADL` under GPLv3.

6.3 Preliminary Synthetic Experiments

In this section we present preliminary experiments on a small model. We exploit the properties of ProDTTAL and RTI+ in Section 4.3.4 to build special PDRTAs from which we sample anomalies. Additionally, we investigate the performance of BUTLA and compare it to its improved version BUTLA-i (cf. Section 6.2.3.2).

6.3.1 PDRTA Data Generation

In this experiment we sample normal and abnormal sequences directly from a PDRTA. We inserted anomalies into the PDRTA in two distinct ways—one that cannot be detected by a Probabilistic Deterministic Timed Transition Automaton (PDTTA), but can be detected by a PDRTA and vice versa. As the PDTTA does not create more than one transition for a single symbol and different time values it cannot detect anomalies that occur in substructures that are reached because of the same symbol but different time values (called *substructural*). The PDRTA can distinguish between different timings and create more than one transition for a single symbol. On the other hand the PDRTA handles time values and symbols independently, hence it cannot detect an abnormal time value v in conjunction with symbol a if v is normal in conjunction with symbol b (called *interdependent*).

Figure 6.4 shows the normal PDRTA with the two modifications we performed (marked red). The initial PDRTA is shown in Fig. 6.4a, Fig. 6.4b shows the modification that cannot be detected by a RTI+ (interdependent) and Fig. 6.4c the modification that cannot be detected by a ProDTTAL (substructural). In Fig. 6.4b we swapped the time guards from q_0 for symbols a and b. In Fig. 6.4c we swapped the time guards from q_1 for the symbol c.

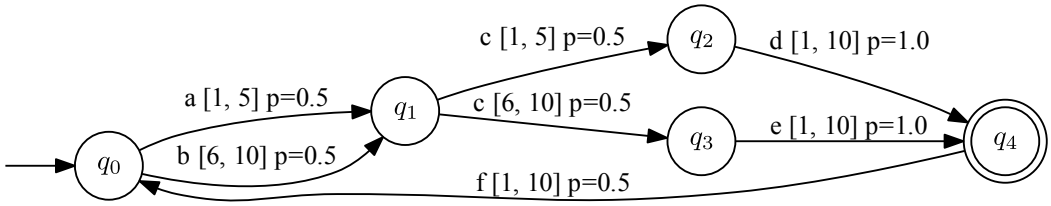
The normal sequences are always sampled from the initial automaton while the anomalies are sampled from the automatons in Figs. 6.4b and 6.4c. We sample 700 sequences for training as this is enough to learn such a small automaton. For testing we sample 300 sequences with an anomaly rate of 50% (as mentioned in Section 6.2). Every configuration is evaluated with SMAC for four hours. These four hours are enough since the automaton is simple and the data set is small. We expect RTI+ to perform poorly on anomalies sampled from Fig. 6.4b, and ProDTTAL to perform poorly on anomalies from Fig. 6.4c because these anomalies were designed to not be detectable by the respective algorithm.

6.3.2 Results

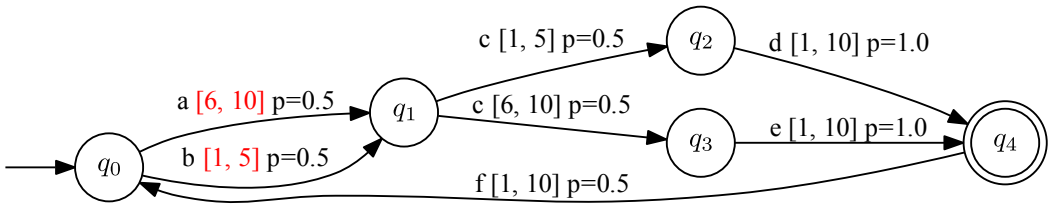
For comparison we evaluate the other algorithms (BUTLA, ALERGIA, TPTA) on the same data sets as well.

Table 6.1 shows the results of this experiment in terms of Matthews Correlation Coefficient (MCC) without any improvements of BUTLA (cf. Section 6.2.3) or a preprocessing step for the other algorithms. The best MCC is printed in bold and the number of states of the final model is given in round brackets (lowest in bold). ALERGIA is given as a baseline, but as the inserted anomalies are only visible if the time values are taken into account, the result is not surprising[5]. We expected

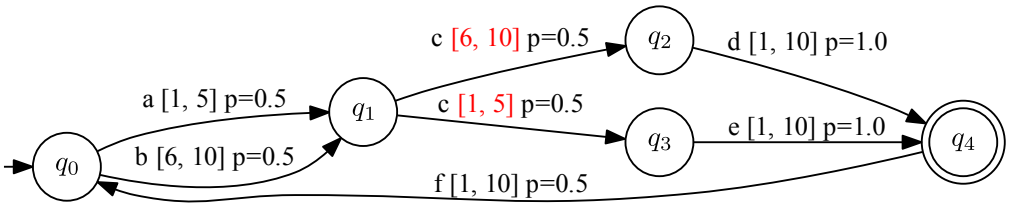
[5]Recall that an MCC of zero is equivalent to random guessing.



(a) Normal PDRTA that is used for sampling normal sequences.



(b) Interdependent anomaly in q_0 for symbols a and b.



(c) Substructural anomaly in q_1 for symbol c.

Figure 6.4: PDRTA modifications for tailored anomaly creation.

the TPTA and ProDTTAL to perform similarly on both data sets as the TPTA is a PDTTA without merges but both represent similar (timed) languages. While this assumption holds for the substructural data, ProDTTAL even outperforms the TPTA in the interdependent data with an optimal result ($MCC = 1$). We believe that the TPTA overfits the training data whereas ProDTTAL generalizes well. As expected, ProDTTAL performs poorly on the substructural anomalies (0.31) and optimal on the interdependent ones, while RTI+ performs poorly on the interdependent anomalies (0.38) and optimal on the substructural ones. BUTLA (without improvements) performs poorly on the substructural anomalies (0.08) as well as on the interdependent anomalies (0.46).

The number of states is almost equal for both anomaly types and the algorithms ALERGIA, ProDTTAL and RTI+ (5), whereas BUTLA requires more states (19 and 11). As the TPTA does not merge any states, it contains a huge amount of states (685 and 870).

In addition to performing the original algorithms we also evaluate the improved version of BUTLA (BUTLA-i; cf. Section 6.2.3). For a ALERGIA, TPTA and ProDTTAL we first perform the BUTLA-preprocessing to overcome the problem of not detecting substructural anomalies.

Table 6.2 shows the result of the experiments with the improved algorithms. We expected that the BUTLA-preprocessing (time-based event split) enables ALERGIA, TPTA and ProDTTAL to detect substructures based on different time behavior, and

Table 6.1: MCC for data sampled from PDRTAs with substructural and interdependent anomalies.

anomaly	ALERGIA	TPTA	ProDTTAL	RTI+	BUTLA
substructural	0.01 (5)	0.31 (685)	0.31 (**4**)	**1.00** (5)	0.08 (19)
interdependent	0.12 (**5**)	0.83 (870)	**1.00** (**5**)	0.38 (**5**)	0.46 (11)

thus, increase the respective MCC. Surprisingly, the BUTLA-preprocessing does not improve the MCC for those algorithms, but sometimes even decreases the algorithms' performance. On the other hand, the improvements in BUTLA (BUTLA-i) boost its performance to an (almost) optimal MCC of 1.00 (0.97).

The number of states remains (almost) the same for the algorithms ALERGIA, TPTA and ProDTTAL. However, BUTLA requires more states compared to the not optimized version for substructural anomalies (49 instead of 19), but less states for interdependent anomalies (6 instead of 11).

Table 6.2: MCC for data sampled from PDRTAs with different types of anomalies and algorithm improvements.

	with BUTLA-preprocessing			
anomaly	ALERGIA	TPTA	ProDTTAL	BUTLA-i
substructural	0.00 (6)	0.27 (685)	0.29 (**4**)	**0.97** (49)
interdependent	0.03 (**5**)	0.83 (870)	**1.00** (**5**)	**1.00** (6)

To give an impression of how many hyperparameter settings were evaluated, Table 6.3 shows the number of different hyperparameter settings for the respective algorithm (without improvements). SMAC stopped evaluating after 4 hours. Depending on the runtime of the algorithm, SMAC evaluated a different number of hyperparameter settings. Please note that SMAC spawns a new process for every new hyperparameter setting. Hence, the overhead of spawning Java processes may be even bigger than the actual runtime of the algorithm. Nevertheless, the table gives an indication on the runtime of the algorithms. The TPTA performs the least evaluations because it has to build a KDE and approximate the integral for every transition. As the TPTA is a tree, it contains roughly as many transitions as states and has to approximate a lot of integrals. ProDTTAL first runs ALERGIA and then approximates the KDEs via Monte Carlo integration. It is obvious that ProDTTAL runs slower than ALERGIA. Surprisingly, RTI+ runs quite fast. This fast runtime may yield from the small data set and the simple automaton. For bigger data sets the table may look different. BUTLA runs slightly faster than RTI+ and ALERGIA is the fastest algorithm because ALERGIA only operates on events and discards the timing information.

Table 6.3: Number of hyperparameter settings evaluated by SMAC in 4 hours for the algorithms without improvements.

anomaly	ALERGIA	TPTA	ProDTTAL	RTI+	BUTLA
substructural	**7492**	2452	4583	5136	5782
interdependent	**8411**	2736	4727	5739	6237

Additionally, we evaluated Minimal Divergence Inference (MDI) as a Probabilistic Deterministic Finite Automaton (PDFA)-learning algorithm for ProDTTAL. Unfortunately, MDI did not increase the classification performance as standalone or in conjunction with ProDTTAL. The results in terms of MCC and number of states are the same as the ones with ALERGIA in Table 6.1. Maybe, the automaton we sample the data from is too simple such that ALERGIA and MDI can infer an (almost) optimal PDFA. Hence, we investigate MDI again on real data in Section 6.5.

6.4 Synthetic Data Evaluation

In this section we evaluate the algorithms RTI+, BUTLA and ProDTTAL on more synthetic data. Therefore, we construct different automata and sample data from these automata. As we control the model (automaton) that generates the positive data, we have two possibilities to sample positive and negative data:

1. Insert anomalies into the automaton and sample positive and negative sequences directly from the automaton.

2. Sample only positive sequences from the automaton and create negative sequences artificially or model based (with a TPTA) as described in Section 5.2.3.

For both approaches no currently existing automaton model can represent the behaviors of PDRTA, PDTA and PDTTA at the same time. The PDRTA cannot model probability distributions other than a uniform distribution for every time bin. For the PDTA no distributions are defined for the time intervals. The PDTTA cannot model different substructures based on time values because it does not contain any time guards.

If we chose one of the models to sample the sequences from, the performance may not be comparable because the results may be biased (probably towards the learning algorithm for the type of model the sequences were sampled from).

Still, we could sample sequences from a PDRTA and a PDTTA[6] being aware of possibly biased results or try to create a more expressive model. For the latter option we propose to build a non-deterministic PDTTA (called Probabilistic Non-Deterministic Timed Transition Automaton (PNTTA)) where the same event e may occur for more than one transition from a single state q.

Definition 16 (PNTTA)**.** *A Probabilistic Non-Deterministic Timed Transition Automaton is a six-tuple* $(\Sigma, Q, q_0, \delta, \pi, \tau)$ *with*

- Σ *is a finite alphabet comprising all relevant events.*
- Q *is a finite set of states.*
- $q_0 \in Q$ *is the start state.*
- $\delta \colon Q \times \Sigma \to \mathcal{P}(Q)$*: The state-transition function assigning a state and symbol arbitrarily many successor states.*
- π *twofold:*
  - $\pi_1 \colon \delta \to [0, 1]$*: The transition probability function assigning a probability to every transition .*
  - $\pi_2 \colon Q \to [0, 1]$*: The final state probability function assigning a probability to every state .*

[6] For a PDTA it is not possible to sample sequences as no probability distribution over the time values is defined.

- $\tau\colon \delta \to \Theta$ *is a transition time plausibility function, which assigns a plausibility distribution* $\theta \in \Theta$ *to each transition, with* Θ *being the set of all possible plausibility distributions. Every* $\theta \in \Theta$ *has the signature* $\theta\colon \mathbb{N}_0 \to [0, 1]$*, with* $\mathbb{N}_0$ *representing all possible time values.*

The resulting PNTTA can represent different substructures based on the time value observed in combination with an event.

Hence, we have two options to sample artificial sequences from:

1. Sample from a PDRTA or PDTTA being aware of possibly biased results.
2. Sample from a PNTTA.

We end up with nine possibilities how to generate positive and negative sequences as shown in Table 6.4.

Table 6.4: Nine possibilities how to generate synthetic data sets.

anomaly generation	generative model
sampled directly	PDRTA
	PDTTA
	PNTTA
artificially	PDRTA
	PDTTA
	PNTTA
model-based	PDRTA
	PDTTA
	PNTTA

In the following we will present the respective automaton instantiations of the PDRTA, PDTTA and PNTTA and how we modified them to directly produce anomalies. For the artificial and model-based anomaly creation we refer to Section 5.2.

6.4.1 Direct Anomaly Insertion

Figure 6.5 shows the initial automaton from which we sample the normal sequences. It is a very simple automaton with four states consisting only of one branch and join and one loop.

The automaton model, their symbols and probabilities, and the respective time distributions for the transitions t_1 to t_5 are shown in Table 6.5. The transition probabilities are the same for all automaton models. The symbol for transition t_2 differs between the PDTTA and the other models (a or b) because the PDTTA cannot contain two transitions from a single state with the same symbol (in this case q_0). The time distributions for the PDTTA and the PNTTA are the same. The PDRTA can only represent uniform time distributions and thus we adapted them to produce a (to some extent) similar time behavior. We use $\mathcal{N}$ ($\mathcal{U}$) as symbol for a normal (uniform) distribution. Overlapping distributions are indicated by the set

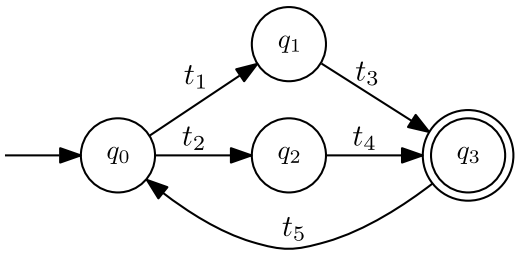


Figure 6.5: Automaton for sampling normal and abnormal sequences.

union '∪'. The final state probability for state q_3 is initially set to an arbitrarily fixed value greater than zero. We fix $\pi(q_3) = 0.5$ such that we do not obtain too long sequences.

Table 6.5: Overview on the symbol and time distributions for the different automaton models.

automaton	**property**	$\mathbf{t_1}$	$\mathbf{t_2}$	$\mathbf{t_3}$	$\mathbf{t_4}$	$\mathbf{t_5}$
all	probability	0.5	0.5	1	1	0.5
PDTTA	symbol	a	b	c	d	b
PNTTA PDRTA	symbol	a	a	c	d	b
PNTTA PDTTA	time	$\mathcal{N}(2.5, 1)$	$\mathcal{N}(10, 2)$ $\cup\mathcal{N}(15, 2)$	$\mathcal{N}(50, 10)$	$\mathcal{N}(5, 1)$	$\mathcal{U}(1, 20)$
PDRTA	time	$[0; 5]$	$[6; 15]$	$[0; 100]$	$[0; 10]$	$[1; 20]$

Table 6.6 summarizes how we insert the anomalies $A_1, \ldots, A_7$ directly into the respective automaton models. The anomalies can be defined independently of the specific automaton model because we only change specific aspects of the transitions that are common in all automaton models. These anomalies should not be confounded with the anomaly types 1–5 that we identified for Discrete Event System (DES). Instead, they change symbols or time behavior in the data-generating automaton that should be hard to detect based on the properties of the respective algorithms. The anomalies A_1 and A_2 change symbols but do not touch time behavior while the anomalies A_3 and A_4 only change the time behavior. Anomaly A_5 changes the final state of the automaton and the anomalies A_6 and A_7 change both, symbols and time behavior, in the respective automaton model.

In addition to the direct anomaly creation, we sample data from the normal automaton and create the anomalies artificially and model-based (type 1–5; cf. Section 5.2). Furthermore, we create a mixed data set that contains anomalies of all types in equal size. As in the preliminary experiments (cf. Section 6.3) every configuration is evaluated with SMAC for four hours. These four hours are enough since the automaton

Table 6.6: Overview of ways of directly inserting anomalies.

anomaly type	description
A_1	Exchange symbol of t_3 and t_4.
A_2	Exchange symbols of t_1 and t_2.
A_3	Exchange time distributions of t_1 and t_2.
A_4	Exchange time distributions of t_3 and t_4.
A_5	Make q_0 the final state with the same probability (0.5) and remove q_3 as final state (adapt the transition probabilities accordingly).
A_6	Exchange t_3 and t_4.
A_7	Exchange t_1 and t_2.

is simple and the data set is small. As before we sample 700 sequences for training as this is enough to learn such a small automaton. For testing we again sample 300 sequences with an anomaly rate of 50% (as mentioned in Section 6.2). For BUTLA we added all improvements as optional parameters to SMAC such that they can be used if SMAC finds them useful (cf. Section 6.2.3). As performance metric we used the MCC as in the previous experiments. Even though the BUTLA-preprocessing did not lead to good results in the preliminary experiments for ALERGIA, TPTA and ProDTTAL we added the BUTLA-preprocessing as optional hyperparameter. The preliminary results may be data dependent and hence, the preprocessing may yield better results for different data. Depending on whether the preprocessing is enabled for ALERGIA, its anomaly detection performance may increase for particular anomalies.

In the following we will present all results of the experiments with synthetic data. We start with data sampled from the same automaton as in the preliminary experiments, but with a different setup, and then present the anomalies we sampled from the automaton shown in Fig. 6.5. As we can implement this automaton as a PDRTA, PDTTA or PNTTA we present results for every such implementation. For all experiments we do not show the standard error because it is very small (or not existent at all), most probably because the all test data sets were sampled from the same data-generating process.

The experiments have to be read with care because the random or model-based anomaly insertion is arbitrary and inserting anomalies in any other way may yield different results. Still, they give a good overview how the approaches perform in different setups. In general, the anomalies of type 1 (abnormal single event) and 5 (abnormal ending) can be detected very easily by all approaches. The approaches ProDTTAL, RTI+ and BUTLA-i (which we will call *relevant*) usually outperform ALERGIA and TPTA because ALERGIA cannot represent time and the TPTA does not generalize. Sometimes, (almost) all approaches achieve the same result which is an indication that no better than the achieved performance can be achieved for the respective data set.

6.4.2 Additional Results for the Initial PDRTA

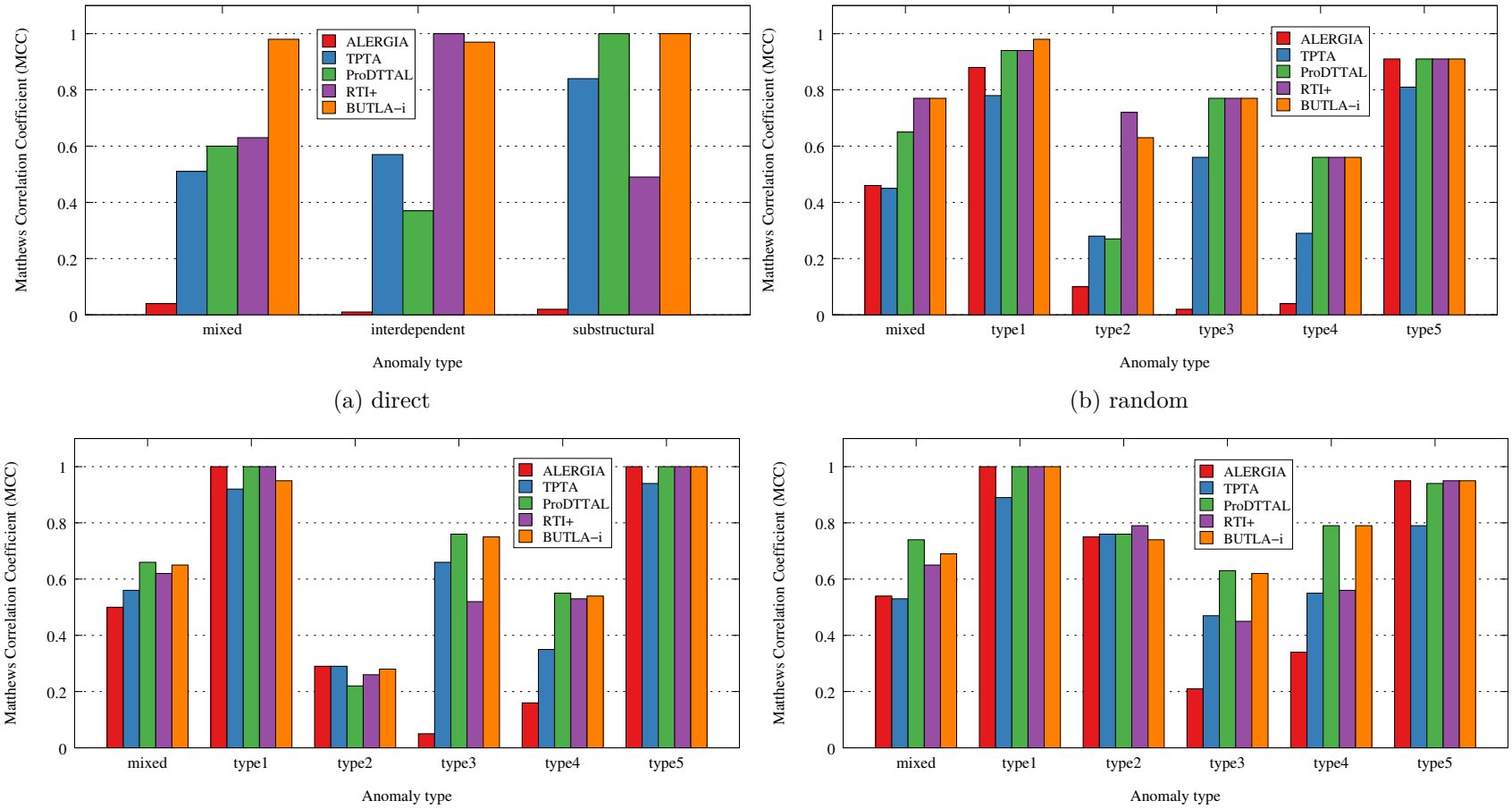
Figure 6.6 shows the experiment similar to the preliminary experiments, except we create a data set in which we equally mix all anomaly types and apply our random and model-based anomaly insertion approach. The directly inserted anomalies (Fig. 6.6a) are a recap of the preliminary experiments (including algorithm improvements). Additionally, we evaluate the performance on a mixture of both anomalies. ALERGIA (red) performs slightly better in the mixed case, but still very poorly compared to the other approaches. The TPTA (blue) performs almost as good as ProDTTAL (green) and RTI+ (purple) while BUTLA-i (orange) performs almost optimal. The interdependent and substructural anomaly types were already presented in the preliminary synthetic experiments (Table 6.1 in Section 6.3).

For the random anomalies (Fig. 6.6b) ALERGIA and TPTA perform equally for the mixed case, but show differences for the non-mixed case: ALERGIA performs way worse than the TPTA on anomalies of types 2–4, but slightly better on types 1 and 5. Therefore, the almost equal performance on the (equal) mixture of anomalies surprises. ProDTTAL performs as good as RTI+ and BUTLA-i on all anomaly types except for type 2. In this case, the inferiority of ProDTTAL for type 2 explains the slightly worse performance on the mixture of anomalies. RTI+ and BUTLA-i perform best and equally well on all anomaly types and hence, also equally on the mixture of anomalies.

For the model-based anomalies without preprocessing, all approaches perform optimal on anomalies of type 1 and 5 (TPTA only almost optimal).

The difference between the model-based anomalies with and without preprocessing is quite small. The ranking of the approaches is mostly the same for both approaches but the results strongly differ for type 2 (performance of all algorithms is higher with preprocessing) and type 4 (ProDTTAL and BUTLA-i perform better with preprocessing).

Summarizing, BUTLA-i optimally classifies on direct anomalies (cf. preliminary experiments in Section 6.3), ProDTTAL performs worse than RTI+ and BUTLA-i on random anomalies, and for model-based anomalies the three relevant approaches perform similarly.



(a) direct

(b) random

(c) model-based without preprocessing

(d) model-based with preprocessing

Figure 6.6: MCC for different types of anomalies sampled from the PDRTA of the preliminary experiments.

6.4.3 PDRTA Data Generation

Figure 6.7 shows the results of the experiments with data sampled from a PDRTA based on the automaton model in Fig. 6.5 with the direct anomaly insertion as described in Table 6.6.

For the directly inserted anomalies, ALERGIA (red) performs very poorly except for anomaly A_5 (for which every approach performs optimal). Hence, ALERGIA's overall performance is quite low. The other approaches perform quite well except for anomaly A_2, for which we cannot explain this very poor performance. The performance of the TPTA (blue) is better than ALERGIA's, but always behind the relevant approaches. For all anomalies RTI+ (purple) performs best with an optimal performance for A_1, A_3, A_5, A_6, A_7. This performance is not very surprising because the data-generating model is a PDRTA. ProDTTAL (green) is slightly worse than RTI+ in all cases with a bigger discrepancy for A_3. In almost all cases, BUTLA-i (orange) performs almost as good as ProDTTAL.

For the random anomalies (Fig. 6.7b) the result is similar to the direct anomalies. RTI+ performs best on all types of anomalies followed by ProDTTAL and BUTLA-i. Surprisingly, ALERGIA performs very poor on anomalies of type 2–4, but almost as good as the TPTA for the mixture of anomalies. It seems that the anomalies from type 1 to type 4 become harder to detect for all approaches (decaying with every type) whilst the anomalies of type 5 are easy to detect.

The results for the model-based anomaly insertion are similar for the mixture of anomalies but differ when looking at every single type of anomaly. Looking at the model-based anomalies without preprocessing (Fig. 6.7c) the relevant approaches do not differ much in terms of MCC. Even ALERGIA and the TPTA deliver comparable results except for anomaly types 3 and 4 and ALERGIA can even detect time-based anomalies of type 3. As usual, the anomaly types 1 and 5 are very easy to detect. Type 2 is still well detectable with an MCC of 0.8. For types 3 and 4 the performance of all approaches is quite low (≈ 0.4) with the baselines a bit worse (type 3) or quite bad (type 4).

The model-based anomalies with preprocessing (Fig. 6.7d) only yields a different result for the anomaly types 2 and 3. For type 2, the performance is lower than without preprocessing (0.8 vs. 0.5) and ALERGIA and the TPTA slightly beat RTI+ (difference to ProDTTAL and BUTLA-i is approx. 0.1 in MCC). On the other hand, the performance for type 3 improves to an MCC of almost 0.8 (0.4 without preprocessing) with only one change in the ranking (TPTA performs better than RTI+).

All in all, RTI+ detected the best when sampling data and anomalies from a PDRTA, slightly followed by the other relevant approaches ProDTTAL and BUTLA-i. Some anomaly types are very easy to detect while for others the anomaly detection performance is far from being optimal.

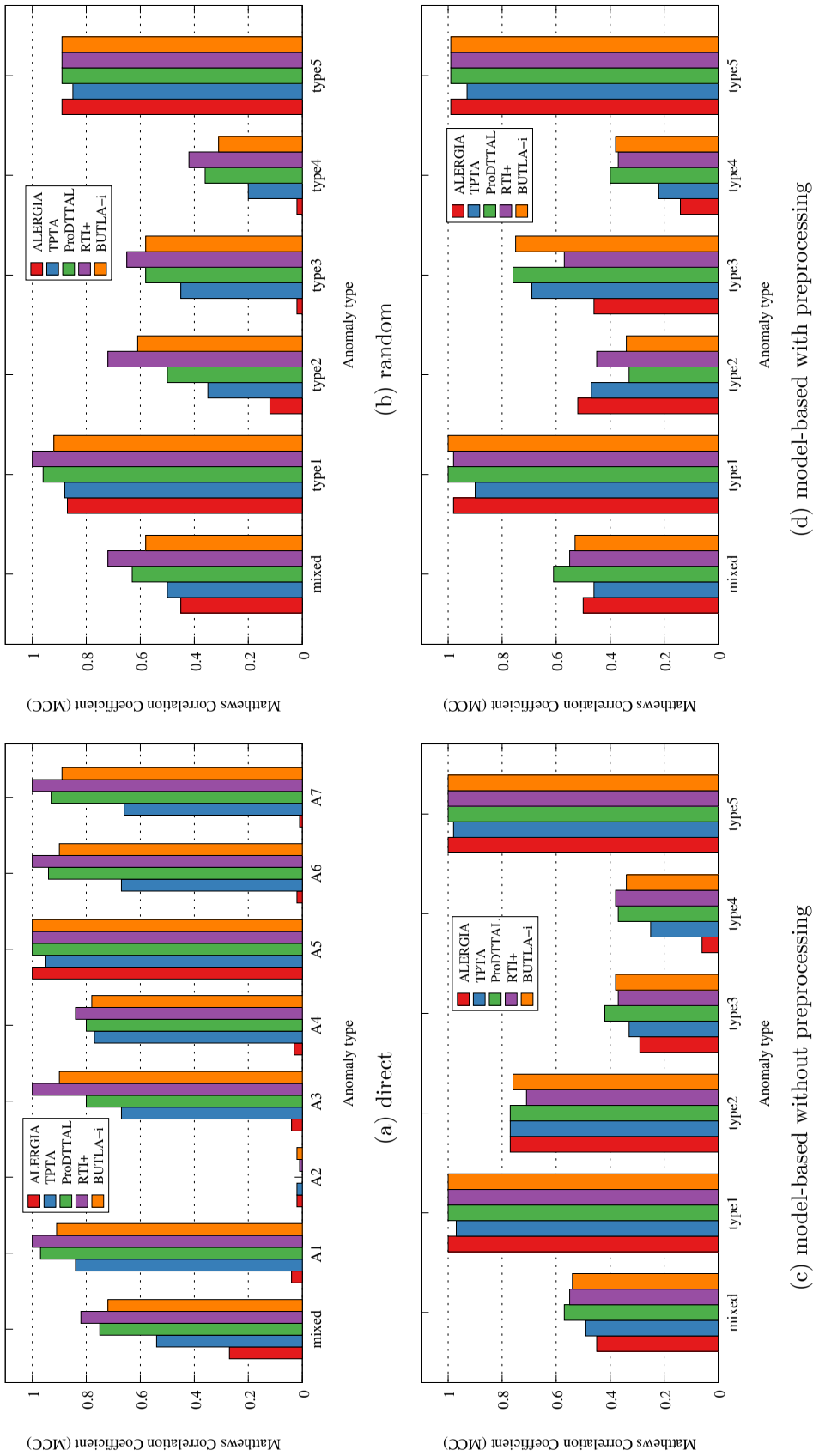


Figure 6.7: MCC for different types of anomalies sampled from a PDRTA.

6.4.4 PDTTA Data Generation

Figure 6.8 shows the results of the experiments with data sampled from a PDTTA based on the automaton model in Fig. 6.5. For the direct anomalies (Fig. 6.8a) all approaches except ALERGIA (red) perform very well with ProDTTAL (green) performing optimal, followed by BUTLA-i (orange), RTI+ (yellow) and TPTA (blue). ProDTTAL and BUTLA-i usually perform optimal for all inserted anomalies $(A_1, \ldots, A_7)$. The TPTA performs slightly worse than optimal for all anomalies. RTI+ performs better than the TPTA except for anomaly A_3 for which RTI+'s performance drops below 0.8. ALERGIA performs optimal for anomalies A_1, A_2, A_5, A_6 and A_7 but not better than random for anomalies A_3 and A_4. The mixture of anomalies summarizes the single anomalies. ProDTTAL and BUTLA-i perform (almost) optimal, followed by the equally well performing RTI+ and TPTA. ALERGIA shows the worst, but still acceptable performance.

Figure 6.8b shows the performance for the random anomalies. The anomalies of type 1 and type 5 are detected very well by all approaches, whilst the anomalies of type 2 and 4 are detected very poorly. As expected, ALERGIA is unable to detect the time-based anomalies of type 3 and 4. Apart from that, all approaches (even ALERGIA) perform almost equal, and the performance on the mixture of anomalies is also comparable.

For the model-based anomaly insertion the trend is comparable to the model-based anomalies for the PDRTA (Fig. 6.7). Comparing the the performance with and without preprocessing, we observe a similar performance for the mixture of anomalies but differences for the anomaly types 2–4. The MCC for the mixed anomalies is also around 0.6 for both model-based approaches.

For the model-based approach without preprocessing (Fig. 6.8c), all relevant approaches perform similar on all anomalies (and their mixture) and also the TPTA performs comparable. As usual, ALERGIA cannot detect the time-based anomalies of type 3 and 4.

Figure 6.8d shows the comparison for the model-based anomaly insertion with preprocessing. While the performance for the anomaly types 1 and 5 is (almost) optimal for all approaches, no approach dominates any other approach for the anomaly types 2–4. Unusually, ALERGIA can detect time-based anomalies of type 3. For the mixture of anomalies, ProDTTAL is the only algorithm that achieves an MCC greater 0.6. While BUTLA-i and RTI+ closely follow ProDTTAL, the TPTA and ALERGIA achieve an MCC of approx. 0.5.

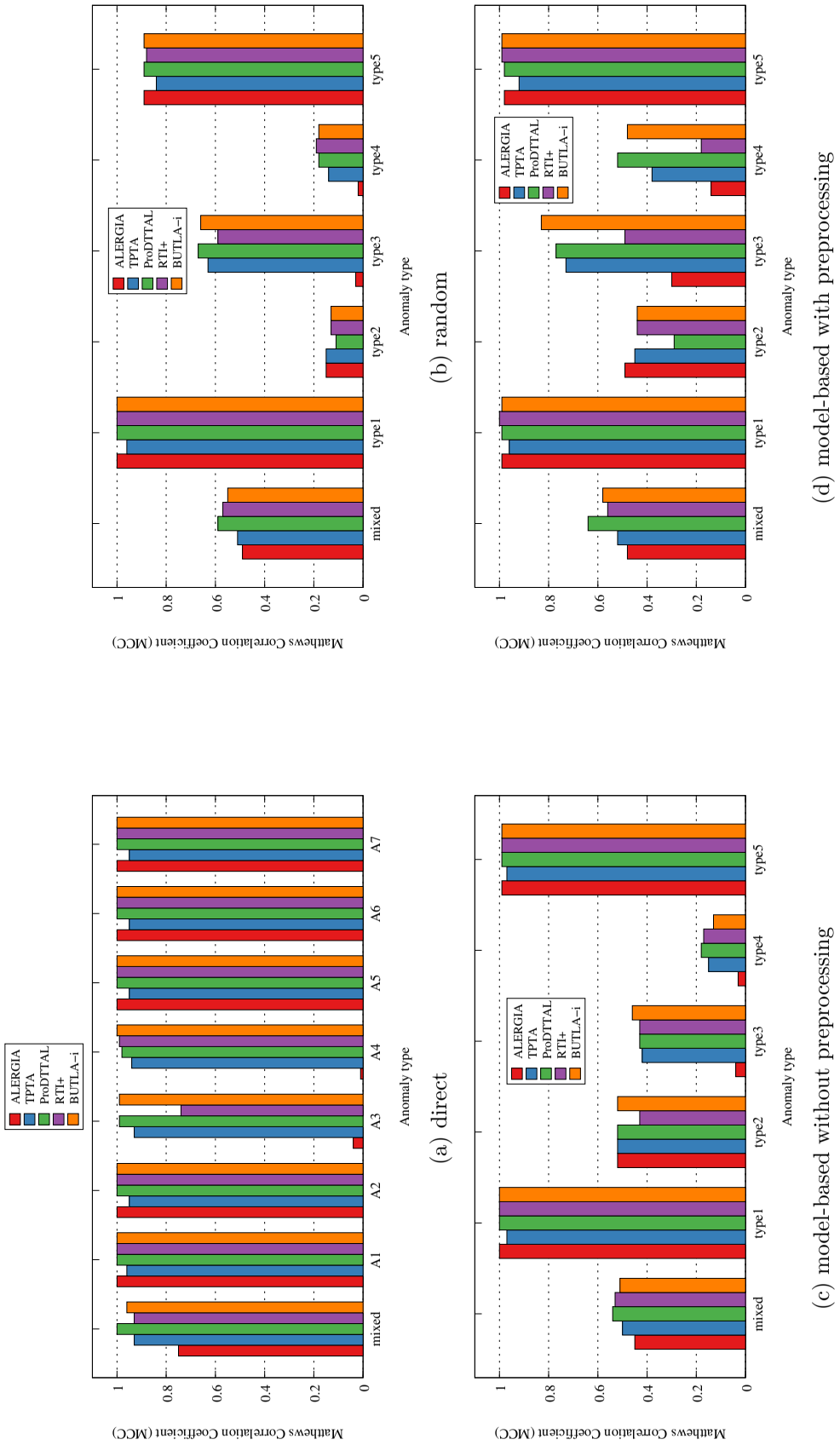


Figure 6.8: MCC for different types of anomalies sampled from a PDTTA.

6.4.5 PNTTA Data Generation

Figure 6.9 shows the results of the experiments with data sampled from a PNTTA based on the automaton model in Fig. 6.5. Compared to Fig. 6.8 (sampling from a PDTTA) the performance of ProDTTAL (green) for mixed anomalies is as good as the TPTA (blue) and worse than RTI+ (purple) or BUTLA-i (orange) for directly inserted anomalies (Fig. 6.9a). ALERGIA (red) achieves a poor performance on the mixture of anomalies because it performs not better than random guessing except for anomaly A_5 (optimal detection). Not further considering ALERGIA, all approaches perform (almost) optimally for anomalies A_1, A_4 and A_5, and very poor for A_2. The differences for anomalies A_3, A_6 and A_7 show the same trend as the mixture of anomalies: TPTA and ProDTTAL perform almost equally, but much worse than RTI+ and BUTLA-i which also perform almost equally.

This trend is similar, but not as strong for the random anomalies (Fig. 6.9b) for type 2 and 3, and the mixture of anomalies. As usual, anomalies of type 1 and 5 are very well whereas type 4 is hardly detectable. Hence, RTI+ and BUTLA-i outperform TPTA and ProDTTAL for all cases, but the difference is smaller than for the direct anomalies (Fig. 6.9a). ALERGIA performs very well for type 1 and 5, and very badly for type 2–4.

For the model-based anomaly insertion with and without preprocessing (Figs. 6.9c and 6.9d) all relevant approaches perform similarly with an MCC between 0.5 and 0.6. Again, ALERGIA can detect time-based anomalies (type 3) for the model-based approach without preprocessing (Fig. 6.9c).

Summary

The experiments on synthetic data showed that depending on the type of anomaly and from which model type we sample the anomalies, one of the relevant approaches (ProDTTAL, RTI+ or BUTLA-i) achieves the highest performance in terms of MCC, and every relevant approach always beats the baselines ALERGIA and TPTA. For some anomaly types the anomaly classification is optimal (e.g. anomaly type 1 in Fig. 6.8a with MCC = 1) while for others it is no better than randomly guessing (e.g. A2 in Fig. 6.7a with MCC = 0). On the one hand, the results of the model-based anomaly insertion have to be read with care because approaches achieved a good performance for anomalies that should not be detectable for those approaches (e.g. ALERGIA sometimes detects time-based anomalies of type 3). We conclude that the TPTA is not a suitable approach for inserting anomalies into an anomaly-free data set. On the other hand, the random anomaly insertion seems consistent with the capabilities of the applied approaches. Considering only the direct and random anomaly insertion, RTI+ performs best if we generated the data from a PDRTA or PNTTA. ProDTTAL performs best if the data-generating model is a PDTTA. In the next section we will evaluate the classification performance of the respective approaches on real-world ATM data.

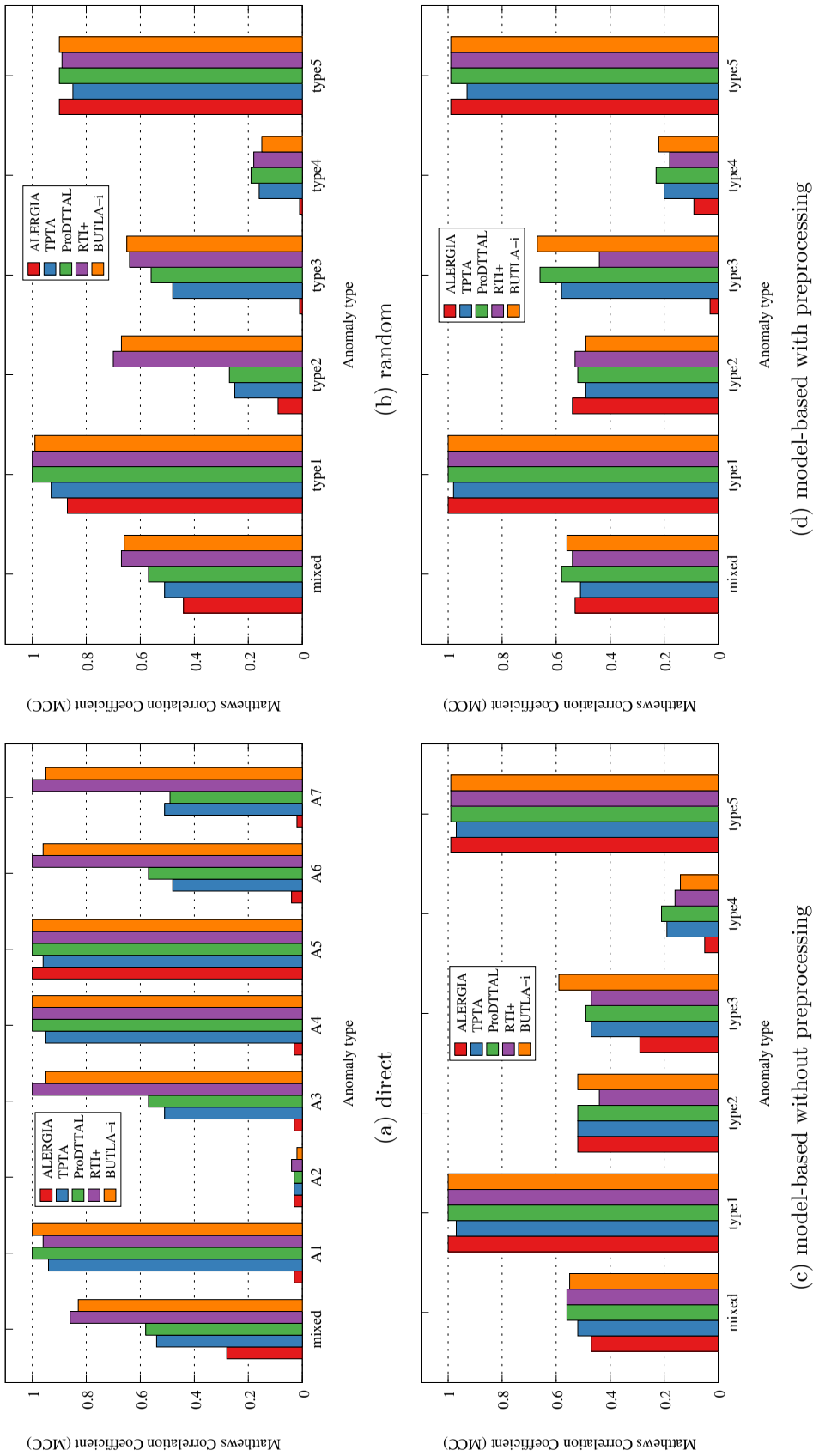


Figure 6.9: MCC for different types of anomalies sampled from a PNTTA.

6.5 Real-world Data Evaluation

In this section we evaluate the algorithms RTI+, BUTLA-i (BUTLA with improvements; cf. Section 6.2.3) and ProDTTAL on real-world data. We use data of cash machines (ATMs) that we gathered in cooperation with Diebold Nixdorf during the research project InverSa, that was partly funded by the German Federal Ministry of Education and Research (BMBF) within the Leading-Edge Cluster "Intelligent Technical Systems OstWestfalenLippe" (it's OWL). In this research project, we investigated how to integrate our approach into an ATM fraud detection system. This approach has been published in [Pri+15]. In [And+14] we presented the ATM fraud detection problem and preliminary results using non-timed sequence data.

The real-world data was gathered on a publicly available ATM in a period of nine months without registering any fraud. We only monitored process data from the workflow but no data related to the customer withdrawing money as this data is encrypted and not contained in the log files. The data was gathered via the diagnosis & serviceability (D&S) platform and from there written to a text log file. The recorded log file was 1.6 GB big and contained 15 million status messages. Before we actually performed evaluations on the data we had to apply some preprocessing and filtering.

6.5.1 Preprocessing

In this section we describe the format of the data and how we preprocess and filter the data. Listing 6.1 shows an excerpt from the log. Every line starts with a timestamp, followed by the module that produces this line (in square brackets), a message id ("MessageId"), and sometimes one or more payload values ("Value").

Listing 6.1: Excerpt from the event log of a Wincor Nixdorf ATM.

```
17.06.2013 11:37:13.968 [SEL] MessageId: SE_DOOR_OPEN
17.06.2013 11:37:14.171 [EPP] MessageId: EPP_SERIAL_STATUS, Value: FFTEST1023101407
17.06.2013 11:37:14.171 [EPP] MessageId: EDM_REMOVAL_SWITCH_STATUS, Value: 192
17.06.2013 11:37:52.921 [PRT] MessageId: TP07_SENSOR_CODE, Value: 0
17.06.2013 11:37:53.000 [PRT] MessageId: TP07_ERROR_CODE, Value: 0
17.06.2013 11:37:57.390 [CHD] MessageId: CHD_INIT_START_STATUS, Value: 2000000000000
17.06.2013 11:37:57.390 [CHD] MessageId: CHD_TYPE, Value: 50
17.06.2013 11:38:00.343 [CHD] MessageId: CHD_INIT_STOP_STATUS, Value: 2000000000000
17.06.2013 11:38:00.343 [CHD] MessageId: CHD_STATUS_TRANSPORT, Value: 0
```

For the preprocessing we create unique events by concatenating the module name and the message id and then removing both fields in a first step. After this step, the log file contains 40 different message types. As we do not process the payload values at all, we also remove them. Hence, the log only contains a timestamp and a string that uniquely identifies the message type. In the following we treat this unique identifier as the event identifier.

Up to this point it is not possible to regard this log file as a set of timed sequences on which we can apply the algorithms presented in this thesis for three reasons:

1. The log file contains periodic events logged by the D&S platform which may confuse the learning algorithms.

2. The log file does not consist of a set of sequences but of one big sequence.

3. The timestamps are absolute values instead of waiting intervals.

Solving the first problem is not as easy as it may seem. As an ATM is a very complex system with messages from various hardware components, even an expert from Wincor Nixdorf is not able to identify the periodic events. Furthermore, some messages seem periodic in some interval (e.g. occurring once a second) but regarding another interval they do not seem periodic any more. Hence, we compute some basic statistics on the periodicity of the respective events like min/max/mean period between two events of the same type and present these statistics to the expert. With his help we identify 17 different message types that are not regarded as periodic[7].

We solve the second problem with the help of an expert from Wincor Nixdorf, too. Therefore, we split the single sequence at the message id 'CHD_ENTRY_-STOP_STATUS' that indicates that a new process of card insertion has been started. Hence, we obtained roughly 13 000 sequences.

For the third problem we compute the time interval between two succeeding timestamps. As we need an entry point, the time interval of the first event is always 0. This does not matter too much, as the first event is always the same, too, as we split at a specific event. In contrast to our synthetic evaluation (cf. Sections 6.3 and 6.4) the time values are not equally distributed in a small range ($[0; 10]$) but sometimes even range between numbers with one and ten digits. As mentioned in Section 6.2.3.2 the original BUTLA KDE split does not work for large time values and may generate a lot of subevents (more than 850 subevents for 17 events as input). Therefore, we disabled the original BUTLA preprocessing formula and enabled the preprocessing only with the original KDE formula.

6.5.2 Experiment Setup

No attacks were registered in the period in which we gathered the data on the public ATM[8]. On the one hand missing attacks are positive because no monetary damage was caused. On the other hand we do not have any data of real attacks at hand. Unfortunately, there is no other way of obtaining real-world data of attacks/ATM fraud because banks do not hand over their data generously, not even to an ATM manufacturer. They do this because they are very concerned about the privacy of their customers. Furthermore, it is not possible to manipulate ATMs in a lab without damaging or destroying them. Therefore, we insert anomalies as we already did in Sections 6.3 and 6.4. For the real-world data we do not have direct anomalies because we do not sample the data from a model we could manipulate. Hence, we can only insert anomalies model-based and randomly. Opposed to our synthetic experiments before, we ran the algorithms on a bigger data set consisting of 9000 training sequences and 4000 test sequences. As usual, we interspersed 50% of anomalies into the test data. From the experiments on scaling (Section 6.6) we know that the algorithms need more time for inferring a model given a big data set. Because of the huge size of the real-world data set we have every approach run seven days with SMAC on a dedicated CPU core with 4 GB memory to find a good hyperparameter combination.

[7] Because of confidentiality we cannot give any further information about which statistics we computed and which message types were chosen/rejected.

[8] Usually, manipulations are found after some time. For this ATM, no money was stolen and no manipulations were found during and after the gathering period.

6.5.3 Results

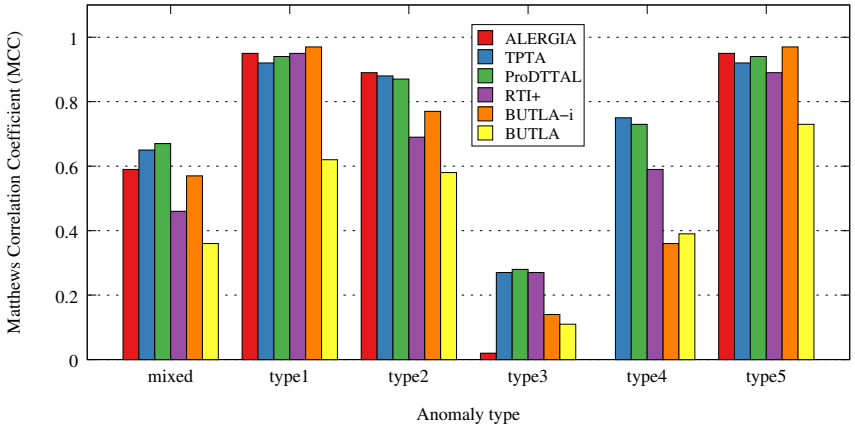
Figure 6.10 shows the result of the respective algorithm with anomalies inserted randomly (Fig. 6.10a) and model-based (without and with preprocessing; Figs. 6.10b and 6.10c). As in the experiments on synthetic data, we evaluate the algorithms on a mixed data set containing all types of anomalies (type 1–5) and create a data set for every single type of anomaly. Additionally, we add the original version of BUTLA (yellow) as comparison which performs worse than BUTLA-i (orange) for all but one anomaly type and often even worse than ALERGIA (red).

The anomalies of type 2 are easily detectable by all approaches while the anomalies of type 3 are hard to detect. For synthetic data the results for these types of anomalies were mixed. In line with the synthetic data experiments the anomalies of type 1 and type 5 are easily detectable by all algorithms. Hence, we do not describe anomalies of type 1 and 5 in more detail.

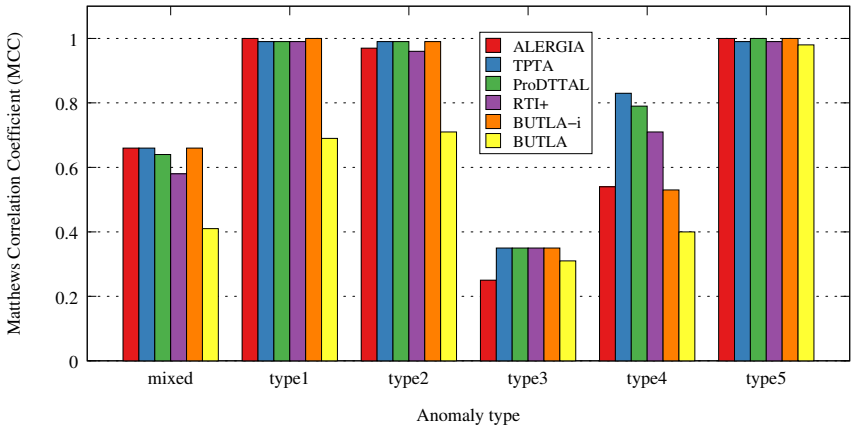
Regarding the randomly inserted anomalies (Fig. 6.10a) ProDTTAL (green) performs best on the mixed data set, closely followed by the TPTA (blue). Surprisingly, RTI+ (purple) performs even worse than ALERGIA. As we deal with anomaly detection for *timed* sequences, the anomalies of type 3 and 4 are important. For these anomaly types ALERGIA performs very badly and RTI+ is outperformed by the TPTA, ProDTTAL (green) and BUTLA-i (orange). For anomalies of type 3, TPTA, ProDTTAL and BUTLA-i perform almost equally whereas for type 4 TPTA slightly outperforms ProDTTAL which outperforms BUTLA-i. However, the TPTA was only given as a baseline because it does not generalize and consists of many states. Regarding the three timed automata inference algorithms (ProDTTAL, RTI+ and BUTLA-i), ProDTTAL slightly outperforms BUTLA-i followed by RTI+.

For the model-based anomalies (Figs. 6.10b and 6.10c) ALERGIA can detect anomalies of type 3 and 4 (time-based) surprisingly well, which should not be possible[9]. If the time-based anomalies are randomly generated (Fig. 6.10a) ALERGIA cannot detect them at all. As reported earlier we let SMAC choose whether to apply the BUTLA-preprocessing for ALERGIA. Because of that we checked whether the good performance of ALERGIA for time-based anomalies results from BUTLA-preprocessing. However, SMAC chose to not apply the preprocessing. We cannot explain the performance of ALERGIA for time-based anomalies (type 3/4) with the model-based anomaly insertion approach. Hence, the results for these types of anomalies should be taken with care. Additionally, the results of the model-based anomaly insertion approach seem odd in general. All algorithms perform almost equally except on the suspicious anomaly types 3 and 4. If we do not take anomaly types 3 and 4 into account the model-based evaluation does not yield any insights. We conclude that the model-based anomaly insertion approach with the TPTA as model is not suitable for inserting anomalies into an anomaly-free data set. In the following sections, we will evaluate the influence of the underlying PDFA-learner for ProDTTAL and the influence of different one-class classifiers.

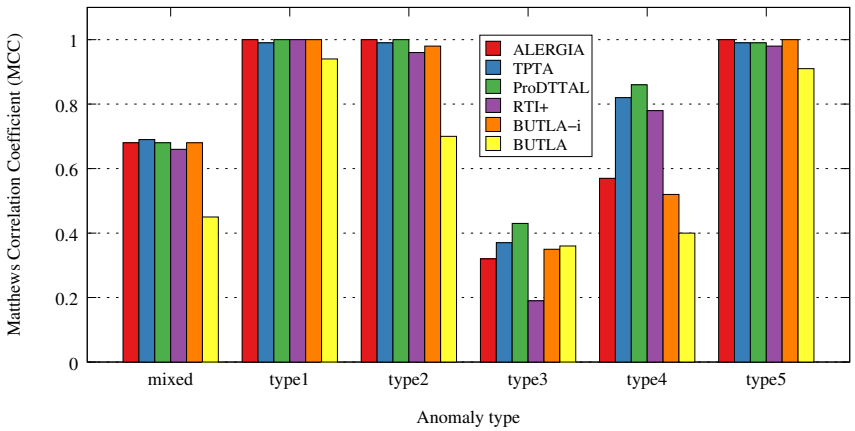
[9] Recall, that the experiments with synthetic data and the model-based anomaly insertion approach yielded results where algorithms (especially ALERGIA) performed well for (time-based) anomalies which should not be detectable for them (cf. Section 6.4.5).



(a) random



(b) model-based without preprocessing



(c) model-based with preprocessing

Figure 6.10: MCC for real-world data with different types of anomalies.

6.5.3.1 MDI vs ALERGIA

In the preliminary experiments (Section 6.3) using MDI as PDFA learner did not show any influence compared to ALERGIA. We repeat the same setup on the real-world data set to assess whether the similarity in the preliminary experiments results from the simplicity of the data. Figure 6.11 shows the comparison of ALERGIA (red) and MDI (blue) used in ProDTTAL on the real-world data set with random anomalies interspersed. The experiments are performed with SMAC for one week (as for all real-world data experiments). Surprisingly, even on the real-world data set the performance of MDI and ALERGIA as ProDTTAL learner is similar. For the anomalies of type one, three and the mixture, ALERGIA performs better than MDI whereas MDI performs better on anomalies of type three. For the remaining anomaly types the approaches perform almost equally. Hence, we conclude that MDI does not improve the anomaly detection performance even though its PDFA should represent the underlying language $\mathcal{L}$ better than ALERGIA [TDH00].

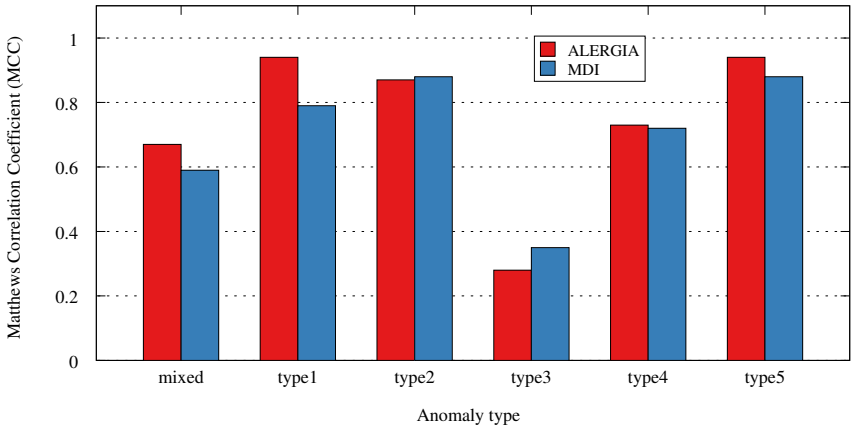


Figure 6.11: MCC for ProDTTAL with MDI and ALERGIA on real-world data with random anomalies.

6.5.3.2 One-class Classifier Influence

As presented earlier in Section 6.2.2.3 we applied different one-class classifiers for the final anomaly detection. The choice of the classifier does not heavily influence the anomaly detection approach in any experiment. Nevertheless, we show the anomaly detection performance of ProDTTAL on the real-world data set with random anomalies interspersed in Fig. 6.12. The threshold detector (red), DBSCAN (blue) and SVM (green) perform almost equally for every type of anomaly. We only give this glimpse on the classifier choice because the results look similar for all data sets. Hence, we conclude that the choice of the automaton learning algorithm and the tuning of the hyperparameters is more important than the choice of the one-class classifier (for the approaches evaluated in this thesis).

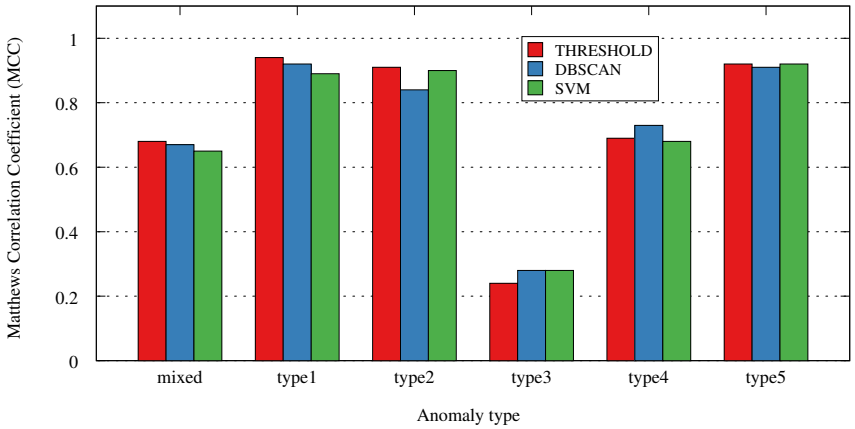


Figure 6.12: MCC for ProDTTAL and different detector methods on real-world data with random anomalies.

Summary

In this section we evaluated the algorithms RTI+, BUTLA-i and ProDTTAL on real-world ATM data. For this data set the model-based anomaly insertion approach led to unexpectedly odd results whereas the random anomaly insertion seemed reasonable. For the random anomaly insertion ProDTTAL outperformed BUTLA-i and RTI+ ranked third. Additionally, we found that the underlying PDFA-learner (MDI vs. ALERGIA) for ProDTTAL and the chosen one-class classifier (threshold vs. SVM vs. DBSCAN) do not heavily influence the anomaly detection performance. In the next section we will evaluate how well the different algorithms scale.

6.6 Evaluation of Algorithm Scaling

In this section we present how we implement the scaling we described in Section 5.3 and its results. We perform the scaling evaluation on data sampled from a densely connected PDRTA (Fig. A.1) and consider time and space needed by the respective algorithm[10]. We measure the training time only, because the time needed for classification is very small compared to the training time. Concerning the memory consumption, it is more difficult to only measure the memory needed for training. Hence, we measure the average memory consumption with a sample rate of 1 Hz during the whole algorithm run.

6.6.1 Scaling Approaches

For every scaling approach (cf. Section 5.3) we modify an initial PDRTA by adding the respective elements (states, transitions, ...) randomly. The elements are added in the following way:

[10]For the scaling evaluation we cannot reuse the automaton from Section 6.3 because it is too small for reasonably increasing the alphabet or the number of transitions.

1. *Event-based transition*: We pick a state that does not have outgoing transitions for every symbol and add a transition to a (randomly chosen) state with a symbol such that the automaton is still deterministic (i.e. the state does not contain more than one outgoing transition with the same symbol).

2. *Time-based transition*: We pick a state and for a transition t with event e we add another transition to a (randomly chosen) state with event e but different time guards. We have to care that the time guards for event e do not overlap.

3. *State*: We pick a state that has more than one outgoing transition and bend one of the transitions to a newly created state. The new state has to become a final state.

4. *Alphabet*: We pick a transition t and exchange the event e by a newly created event e'. We ensure that the old event e still exists somewhere in the automaton.

5. *Input size*: We sample more sequences from the original automaton.

Some of the scaling approaches mentioned above are constrained by the size of the automaton while others are not. Recall that we want to apply one scaling approach at a time. On the one hand it is possible to sample any number of sequences from an automaton independent of its size. We can also add as many time-based transitions as possible by using arbitrary (non-overlapping) time intervals. On the other hand the size of the alphabet is upperbounded by the number of transitions in the automaton, hence $|\Sigma| \leq |\delta|$. If an automaton contains n transitions it is not possible to have an alphabet of size $n+1$ because we could not assign a different symbol to every transition. Therefore, the initial automaton already contains many states and a huge alphabet of equal size ($|Q| = |\Sigma|$) while no state contains outgoing transitions for every symbol. Similarly, we set the number of outgoing transitions to be an arbitrarily small fraction related to the number of states $|Q|$ (here: $\frac{1}{4}|Q|^2$) to be able to add enough (here: $\frac{3}{4}|Q|^2$) additional event-based transitions[11]. The number of (reachable) states is also bounded by the number of transitions ($|Q| \leq |\delta|$).

The parameters of the initial automaton are set to the values shown in Table 6.7. The maximal possible values for the respective elements (states, event-/time-based transitions, alphabet size) are also shown. We count a transition as time-based if there already exists one transition from a state q with a symbol e. The initial automaton contains $|Q| = 25$ states, $\frac{1}{4}|Q|^2 = \frac{1}{4} \cdot 25^2 = 156$ event-based transitions, an alphabet of size $|\Sigma| = 25$ and we sample 1000 sequences. As we want to keep the initial structure of the automaton, we can only bend all but one transition in each state, so we can add less states than we have initial transitions ($156 - 25 = 131$). For the unbounded approaches we set the maximum values in the following way: We choose the maximum number of time-based transitions in a way that the number of transitions does not become greater than for the event-based approach ($156 + 469 = 625$). The maximum number of samples is set to a value such that the experiments can still be performed in a reasonable amount of time.

In Appendix A.2.1 we show how the initial PDRTA would look like for 10 states, 50 transitions and an alphabet of size 10. We did not include the time intervals as all

[11] We can have at most $|Q|^2$ many transitions.

Table 6.7: Initial and maximal values for the runtime evaluation.

scaling approach	event-based transitions	time-based transitions	states	alphabet size	drawn samples
initial PDRTA	156	0	25	25	1000
1	**625**	0	25	25	1000
2	156	**469**	25	25	1000
3	156	0	**131**	25	1000
4	156	0	25	**156**	1000
5	156	0	25	25	**20 000**

time intervals are equal ($[1; 100]$). The PDRTA is not shown in this section because it is quite dense and does not yield any additional insights.

6.6.2 Expectations

In this section we present the expectations on how the algorithms scale in terms of runtime and memory consumption when increasing the sample size or the data complexity.

6.6.2.1 Runtime

Every algorithm computes a Prefix Tree Acceptor (PTA) and hence the runtime depends at least linearly on the size of the training set S_t^+. Thus, the algorithm runtime should grow with an increased sample size.

ALERGIA ALERGIA's runtime is cubic in the size of the constructed PTA. Hence, adding event-based transitions or states should increase ALERGIA's runtime.

TPTA As the TPTA does not perform any merges, it does not consume time for merging (as the other algorithms do) but requires a lot of samples for estimating the KDE for many transitions. If we increase the number of event transitions the TPTA has to approximate more KDEs via Monte Carlo sampling. Hence, we expect the TPTA's runtime to increase with growing sample size and growing number of event transitions and stay constant for other scaling approaches.

ProDTTAL ProDTTAL first applies ALERGIA and then approximates KDEs with Monte Carlo integration. Hence, we expect ProDTTAL to perform slightly worse than ALERGIA because the linear term for the Monte Carlo integration (cf. Eq. (4.9)) does not influence the runtime that much.

BUTLA BUTLA consumes time for computing the event split and then performs a modified version of ALERGIA[12]. For every time-based transition BUTLA increases

[12]We disabled the original BUTLA KDE formula because it produces too many subevents.

the alphabet with a subevent. Hence, we expect BUTLA's runtime to increase if we add more symbols (time-based transitions) because BUTLA has to apply a preprocessing for every symbol (time guard). Apart from that the runtime of BUTLA and ALERGIA should be similar (if both implementations work in a similar way).

RTI+ According to [VdW10] RTI+ runs in polynomial time but no other runtime bound is given. As RTI+ computes the likelihood given the current automaton and the input data before and after each split (or merge), we expect RTI+ to perform very poor in terms of size of the input data. For the rest of the scaling approaches we cannot give a reasonable expectation on the runtime of RTI+.

6.6.2.2 Memory Consumption

The memory consumption of the different algorithms is not easily measurable. For the runtime measurement we measure the time before and after the training phase to receive the time that is really needed for training. For the memory consumption we can only measure the memory consumption of the whole Java process, but we cannot distinguish what the memory is used for. Some memory is needed for reading the input, some is acquired by the Java Virtual Machine (JVM) and may remain used until the garbage collector frees this memory.

Nevertheless, we will give a short overview of what we expect. ALERGIA should need much memory for building the PTA but then the memory consumption should drop[13]. As the TPTA does not perform any merges we expect it to consume a lot of memory because it builds a KDE approximation for every transition in the unmerged PTA. ProDTTAL should require slightly more memory than ALERGIA for the additional KDE approximation. BUTLA's memory consumption should be slightly higher than ALERGIA's memory consumption because the resulting automaton will contain more states because of the event split. We expect RTI+ to consume much memory because it applies the split operation that splits transitions and then copies the subtree of the transition. Hence, every split doubles the memory consumption of the respective subtree.

Because of the difficulties in measuring the algorithms' memory consumption and a possibly misleading interpretation we present the results on the memory consumption in the appendix (Appendix A.2.2) for completeness.

6.6.3 Experiment Setup

While the anomaly detection performance of the approaches is not the main goal of the runtime evaluation we still tune the hyperparameters with SMAC optimizing the anomaly classification in terms of MCC. If we optimized for runtime instead, the hyperparameters would be chosen to execute the algorithm very fast regardless of the classification performance. Thus, we tune the hyperparameters for every approach on data sampled from the initial PDRTA. As the anomaly detection performance is independent of the runtime we intersperse anomalies into the test data artificially (cf. Section 5.2.2). We run SMAC on every data set for 1 day because the sample

[13] We cannot measure a drop in memory consumption during the training phase because we only measure the memory consumption over the whole training phase.

size is comparable to the synthetic experiments (except for the scaling in sample size). Additionally, we only use the threshold classifier because it does not bias the runtime by needing additional training time (opposed to e.g. an SVM).

In this experiment, we also compute standard error bars because the measured durations differed for repeated executions. Hence, the standard error $SE_{\bar{x}}$ of the mean $\bar{x}$ is computed as usual using standard deviation σ and sample size n:

$$SE_{\bar{x}} = \frac{\sigma}{\sqrt{n}} \tag{6.1}$$

6.6.4 Runtime Results

Before presenting the results we want to point out that every runtime value shown depends on the hyper parameters found with SMAC. It may happen that SMAC finds good hyper parameters that run fast and and lead to a good MCC and a slight increase in MCC may result in a very bad runtime. The runtime could also change (increase or decrease) for a different hyper parameter setting which SMAC could find if it runs for a longer period.

Nevertheless, the experiments show some trends on how the algorithms scale. Figure 6.13 shows the runtime behavior of the respective algorithm for the different scaling approaches. The first data point is the same for all plots even though it may seem different because of the different scale of the y-axis.

For increasing number of event transitions (Fig. 6.13a) only RTI+'s runtime heavily increases for about 200 event transitions and then starts dropping again. All other algorithms require roughly the same time with only minor oscillation. We expect RTI+'s behavior to be the source of this strange behavior because by performing a split operation it can increase the PTA size and thus the runtime.

Figure 6.13b shows the runtime of the different approaches for increasing number of time transitions. All algorithms except RTI+ require nearly constant time. ALERGIA usually runs fastest followed by BUTLA, ProDTTAL and the TPTA. For 250 time transitions the increase in ALERGIA's runtime is also visible for ProDTTAL (that uses ALERGIA internally). We assume this increase to be caused by the random generation of the initial PDRTA. For RTI+ the results are surprising again. The runtime steeply increases, stays constant in the middle (except for one drop) and drops again in the end. Again, we expect the split to be the cause of this behavior. Nevertheless, we observe a connection to the behavior for event-based transitions (Fig. 6.13a) as the runtime with 500 additional transitions is almost the same for RTI+ (2 minutes).

The increase in the number of states (Fig. 6.13c) does not give any valuable insights. Unfortunately, the way we increase the number of states while keeping the number of transitions constant leads to a very simple model. Hence, we conclude that the number of states itself does not have a big influence on the runtime, but we assume that the combination of more states and transitions may increase the runtime.

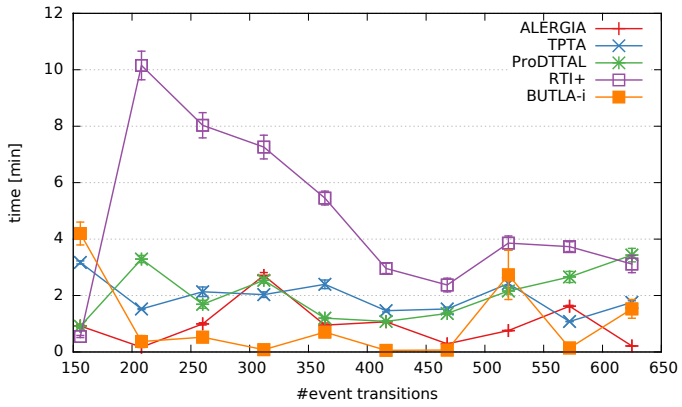
Figure 6.13d shows the runtime of the different approaches for an increasing number of symbols (bigger alphabet size). All approaches but RTI+ show a roughly constant runtime. For RTI+ we observe a similar behavior as for the increase of time-based transitions (Fig. 6.13b): A steep increase at the beginning, a constant

runtime with one small drop in the middle, and a steep drop of runtime at the end. Again, we can only explain this behavior with the split operation. Please note that the highest runtime is 70 minutes whereas it was 10 minutes for the increase in event- and time-based transitions.

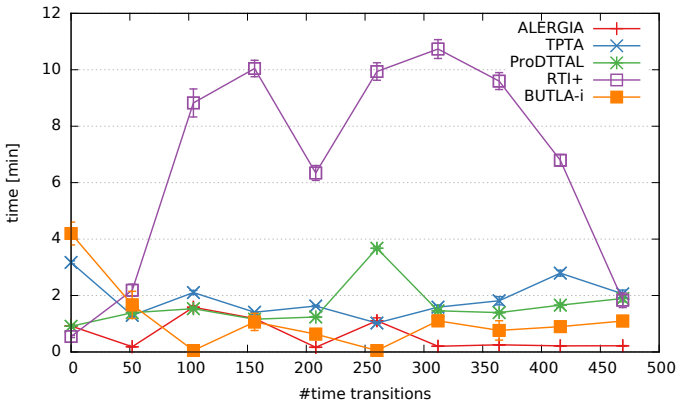
A real trend can be observed for the increase of samples in the data set (Fig. 6.13e). In this figure the y axis reaches to 1400 minutes. For ALERGIA and ProDTTAL we observe a similar trend with ProDTTAL being slightly slower than ALERGIA. The TPTA shows a slow increase in runtime, then it jumps for 9500 samples and then drops again. A possible explanation is the circular shape of the automaton we sample from. If we sample more sequences we will sample new sequences that increase the size of the PTA and hence increase ALERGIA's and ProDTTAL's runtime. For the TPTA an additional path in the tree does not heavily influence the runtime.

The data sets with 9500 samples seem to contain an unusual characteristic because the runtime of the TPTA heavily increases and no results exist for BUTLA. BUTLA runs very fast compared to the other algorithms but for more than 14 000 samples BUTLA crashes because of a stackoverflow. As this would result in a change in the implementation of BUTLA and a possible influence on other results we did not change the implementation of BUTLA but suspect the runtime to continue the trend.

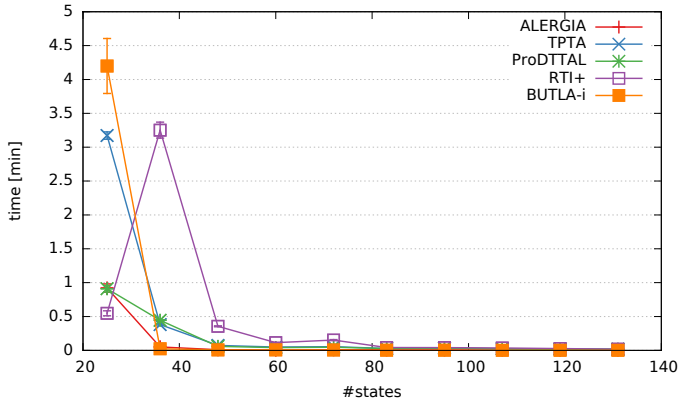
The runtime of RTI+ behaves as expected. For the likelihood ratio test RTI+ has to compute the likelihood of the current automaton given the input data set before and after each operation (split or merge). As the input data set grows the likelihood computation requires much more time. For more than 14 000 samples at RTI+ does not finish at all in the allowed runtime of SMAC.



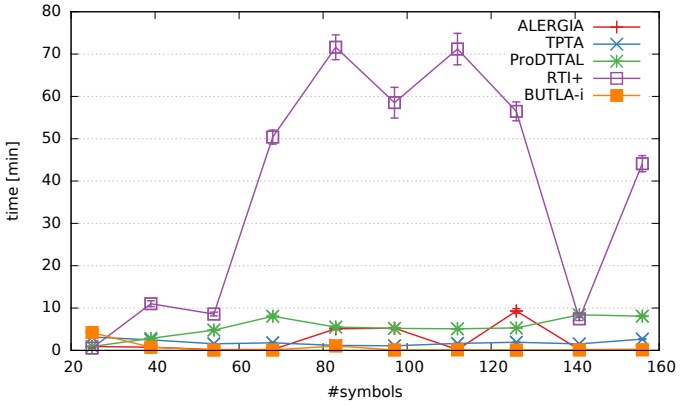
(a) Event transitions.



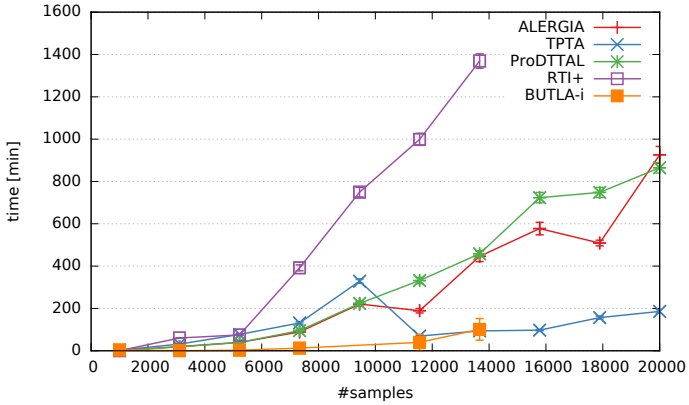
(b) Time-based transitions.



(c) States.



(d) Alphabet.



(e) Sample size.

Figure 6.13: Runtime behavior of the algorithms for different scaling approaches.

Summary

In this chapter we experimentally compared ProDTTAL with BUTLA, RTI+ and two baselines (ALERGIA and TPTA) on synthetic as well as on real-world ATM data. The synthetic data is generated by various manually created automaton models, whereas the real-world data originates from a public ATM. In the preliminary (synthetic) experiments we found that BUTLA does not perform very well in its original description from [Mai14]. Our improved version BUTLA-i performs almost optimally for the preliminary experiments.

For the remaining experiments, the random and model-based anomaly insertion approaches allow the insertion of anomalies into initially anomaly-free data. These approaches enable us to experimentally compare the approaches but the results on these modified data sets may not be as reliable as a data set that contains real-world anomalies[14]. In general, no approach dominates the other approaches on all data sets while all approaches outperform the baselines (ALERGIA and TPTA) in almost all cases. Depending on the anomaly-free data set, the results heavily varied in terms of the performance metric MCC and how the algorithms are ranked based on the MCC. Most often, ProDTTAL and BUTLA-i slightly outperform RTI+ followed by the baselines. The model-based anomaly insertion delivers unexpectedly odd results, e.g. ALERGIA detects time-based anomalies but does not model time at all. The random anomaly insertion results are inline with our expectations such that the algorithms can only detect those anomalies for which they incorporate the necessary information.

For the synthetic experiments, we can also intersperse anomalies into the data-generating model. For these directly inserted and the randomly created anomalies, RTI+ performs best if the data-generating model is a PDRTA or a PNTTA. If we sample data from a PDTTA, ProDTTAL outperforms the other approaches.

On the real-world ATM data, we insert the anomalies randomly and model-based. As before, the model-based approach yields odd results. ProDTTAL achieves the best performance for the random anomalies. BUTLA-i and RTI+ perform surprisingly badly—even worse than the baselines ALERGIA and TPTA.

We also investigated the influence of the one-class classifier and the underlying PDFA-learner for ProDTTAL. All one-class classifiers and also the PDFA-learners performed similarly after tuning the hyperparameters. If we had not tuned the hyperparameters, the performance would have heavily alternated for different one-class classifiers and PDFA-learners.

Finally, we evaluated the scalability of the algorithms in terms of runtime. For the size of the data set, the runtime increases with growing size of the data set. For the other scaling approaches, the runtime behavior does not show any obvious trend. RTI+ heavily oscillates whereas the other algorithms show almost constant behavior (or at least oscillating considerably less).

[14] No real-world data set containing anomalies was available to us.

7 Conclusion and Future Work

This chapter concludes this thesis and gives an outlook on some directions for future work.

7.1 Conclusion

In general, learning models of Discrete Event Systems (DESs) for detecting anomalies becomes more important, especially in the industry. However, only few approaches exist to learn models that deal with timed sequences. After describing the fundamentals (Chapter 2), we reviewed related work and linked the research areas Process Mining, Grammatical Inference and Sequence-based Anomaly Detection in Chapter 3. These research areas (partly) deal with sequence-based anomaly detection, but develop rather independently. Hence, we reviewed the applied approaches, models and algorithms, and pointed out the differences and similarities.

In Chapter 4 we presented the Probabilistic Deterministic Timed Transition Automaton (PDTTA), the Probabilistic Deterministic Timed Transition Automaton Learning algorithm (ProDTTAL) and the Automata-based Anomaly Detection Algorithm (AmAnDA). Compared to state-of-the-art models (PDRTA, PDTA) the PDTTA models time more granularly using Kernel Density Estimations (KDEs) instead of assuming uniform distributions (Section 4.2). For learning PDTTAs from data consisting of timed sequences, we introduced ProDTTAL (Section 4.3). ProDTTAL does neither require the specification of global and hard-to-guess histograms like Real-Time Identification from Positive Data (RTI+) nor does it require the time values to follow a Gaussian distribution like the Bottom-Up Timed Learning Algorithm (BUTLA). On the other hand, the PDTTA learned by ProDTTAL cannot represent different substructures depending on the time value associated with an event. In the experiments, we also pointed out this drawback.

The anomaly detection algorithm AmAnDA is the first algorithm that performs anomaly detection for timed automata in a sophisticated way (Section 4.4). Previous algorithms only checked for non-existing transitions or time values, whereas AmAnDA takes into account the probabilities and plausibilities of events and their time values. With minor modifications AmAnDA can be applied to models learned with RTI+, BUTLA or ProDTTAL. It requires two lists of probabilities/plausibilities to be extractable from the automaton given a timed sequence. After extracting a set of features from the two lists, a state-of-the-art one-class classification algorithm is applied to label the given sequence as normal or abnormal. Additionally, we analyze the runtime complexity of ProDTTAL and AmAnDA (both polynomial in terms of different variables, cf. Theorems 4 and 7), and show that ProDTTAL converges to

the underlying model in the limit (under some assumptions). Last, we explained why the developed algorithms cannot be easily adapted to a two-/multi-class setting (Section 4.5).

In Chapter 5 we presented five types of anomalies that can occur in DES. An anomaly insertion approach was needed to experimentally evaluate the automata learning algorithms later in this thesis. We proposed two approaches how to insert these types of anomalies into anomaly-free data sets—namely the random and model-based anomaly insertion. As the name suggests, the random anomaly insertion inserts anomalies randomly whereas the model-based anomaly insertion builds a model (Timed Prefix Tree Acceptor (TPTA)) and intersperses anomalies into this model. After reviewing suitable performance metrics, we introduced an approach to complicate the learning task by increasing different aspects of the model and data set. Therefore, we proposed how to systematically increase the number of transitions, states, symbols and samples while leaving the other aspects untouched. This approach was used to evaluate the automata learning algorithms later on.

In Chapter 6 we experimentally evaluated the anomaly detection performance of RTI+, BUTLA and ProDTTAL, and compared them with the baselines ALERGIA and TPTA. We found that the original BUTLA version does not detect anomalies well (even worse than the baseline), and proposed improvements, called Improved Bottom-Up Timed Learning Algorithm (BUTLA-i). With these improvements BUTLA's performance heavily increased. We evaluated the approaches on synthetic data that we sampled from a predefined automaton in which we interspersed anomalies. Additionally, we inserted anomalies with the random and model-based anomaly insertion approaches proposed in Chapter 5 into synthetic data and real-world data from an Automated Teller Machine (ATM). The results are ambiguous depending on the data set and the anomaly insertion approach. Most often, ProDTTAL and BUTLA-i slightly outperform RTI+. Especially for the real-world data set, ProDTTAL performed slightly better than the other approaches. The choice of one-class classification algorithm did not have any important influence on the anomaly detection performance at all. Still, measuring the performance of anomaly detection algorithms is a challenging task and the performance of the different approaches varies depending on the data set. Furthermore, we evaluated the change in the runtime of the investigated approaches when complicating the model-learning task. Except for increasing size of the data set the runtime behavior did not show any obvious trend.

Finally, we conclude that the sequence-based anomaly detection problem becomes more and more important in real-world applications, especially for monitoring technical systems. Our proposed model (PDTTA), its learning algorithm ProDTTAL and the detection algorithm AmAnDA are a new approach to solve this anomaly detection problem. Especially AmAnDA helps improving the anomaly detection rate even of existing approaches.

7.2 Future Work

In this thesis we compared state-of-the-art approaches for sequence-based anomaly detection. Nevertheless, some aspects remain open for further investigation. In the following, we give a glimpse on topics that could be investigated in the future.

First, we compared ALERGIA and MDI for inferring the underlying PDTTA. Even though both algorithms showed a comparable performance in terms of anomaly detection, other Probabilistic Deterministic Finite Automaton (PDFA)-learning algorithms might be tested as well. The same applies to the usage of the one-class classifiers.

Furthermore, we reviewed a PDFA-learning algorithm and a kernel density estimation for data streams (cf. Section 3.5.2). It would be interesting whether it is possible to apply ProDTTAL and AmAnDA in the scenario of data streams. Even though the single task of PDFA-learning and kernel density estimation are solved in the streaming scenario the combination of both approaches might not be as easy as it may seem. The time plausibility approximation (i.e. the Monte Carlo integration) would have to be recomputed for every change of the kernel density estimation. Additionally, if the underlying PDFA changes after the arrival of a new data point, at least one transition changes and the density estimation itself would have to be recomputed for the changed transition. Both of the mentioned aspects require a recomputation that is not easily possible in the streaming scenario.

In the experiments (Chapter 6) we compared and evaluated algorithms from the research field of Grammatical Inference and Sequence-based Anomaly Detection. One could try to adapt approaches from Process Mining to also incorporate time values and compare its performance to the algorithms presented in this thesis. Furthermore, we systematically built our anomaly insertion and scaling approaches, but they did not work as intended in every scenario. These approaches could be fine-tuned to better create anomalies and increase the learning task difficulty, respectively.

As presented in the real-world data experiments, the log files of ATMs do not only contain time stamps and events but also additional system messages. These system messages could be incorporated into an even more sophisticated automaton model that can use the system messages to improve the anomaly detection performance.

Incorporating BUTLA's preprocessing into ProDTTAL did not result in an increased anomaly detection performance. Nevertheless, we presented improvements for BUTLA that greatly increased its anomaly detection performance. In the future BUTLA and ProDTTAL could be connected in a different manner to further increase the anomaly detection performance to monitor technical systems automatically.

A Experimental Evaluation

In the following we present settings and results of the experimental evaluation from Chapter 6. First, we show the hyperparameter values and their ranges which were used as input to Sequential Model-based Algorithm Configuration (SMAC) (Section 6.1) to optimize the respective algorithms. Then, we present the more results of the experimental algorithm scaling that suffered problems of not measuring the memory consumption correctly.

A.1 Hyperparameter Values

Table A.1 shows the hyperparameters we tuned to improve the anomaly detection performance in the experimental evaluation (Chapter 6). For every hyperparameter we provide the parameter type, the parameter name and a range from which the parameter can be chosen. The parameter type indicates to which algorithm the parameter belongs.

Table A.1: Hyperparameter value ranges.

parameter type	**parameter name**	**range**
general	automaton algorithm	{ALERGIA, TPTA, ProDTTAL, BUTLA, RTI+}
general	feature extractor	{minimal, some, most, all}
general	anomaly classifier	{OC-SVM, DBSCAN, {k,X,G}-Means, threshold-based}
ALERGIA	α	$[0; 1]$
ALERGIA ProDTTAL TPTA	BUTLA-preprocessing	{true, false}
ALERGIA ProDTTAL TPTA	BUTLA-bandwidth	$[10\,000; 1\,000\,000]$
ProDTTAL TPTA	KDE-bandwidth-estimate	{true, false}

Continued on next page

Table A.1 – continued from previous page

parameter type	parameter name	range
ProDTTAL TPTA	KDE-bandwidth	$[0.0000001; 1000]$
ProDTTAL TPTA	Monte Carlo m	$[10; 40\,000]$
ProDTTAL	PDFA-learner	{ALERGIA, Minimal Divergence Inference (MDI)}
ProDTTAL	α	$[0; 1]$
BUTLA	α	$[0; 1]$
BUTLA	bandwidth	$[10\,000; 1\,000\,000]$
RTI+	α	$[0; 1]$
RTI+	histogram bins	$[1; 100]$
OC-SVM	ν	$[0; 1]$
OC-SVM	γ	$[0; 10^{15}]$
OC-SVM	ε	$[0; 1]$
OC-SVM	kernel	{linear, polynomial, RBF, Sigmoid}
OC-SVM	d	$[0; 2\,000\,000\,000]$
OC-SVM	estimate probability	{true, false}
DBSCAN	n	$[1; 100]$
DBSCAN	ε	$[0; 1]$
DBSCAN	distance threshold	$[0; 1]$
k-Means	k	$[2; 1000]$
{k,X,G}-Means	minimum points	$[0; 100]$
{k,X,G}-Means	distance threshold	$[0; 1]$
threshold	single event	$[0; 1]$
threshold	single time	$[0; 1]$
threshold	aggregated events	$[0; 1]$
threshold	aggregated times	$[0; 1]$

A.2 Evaluation of Algorithm Scaling

In this section we present more results on the evaluation of the algorithm scaling from Section 6.6.

A.2.1 Initial Automaton

Figure A.1 shows how the initial Probabilistic Deterministic Real-Time Automaton (PDRTA) from Section 6.6.1 would look like for 10 states $(0 - 9)$, 50 transitions and

an alphabet of size 10 $(0-9)$. As already described before, the automaton does not contain a lot of transitions so that we can increase the number of transitions drastically. On the other hand we can remove some transitions without dividing the automaton into two parts to be able to bend some transitions to new states. We did not include the time intervals as all time intervals are equal ($[1; 100]$).

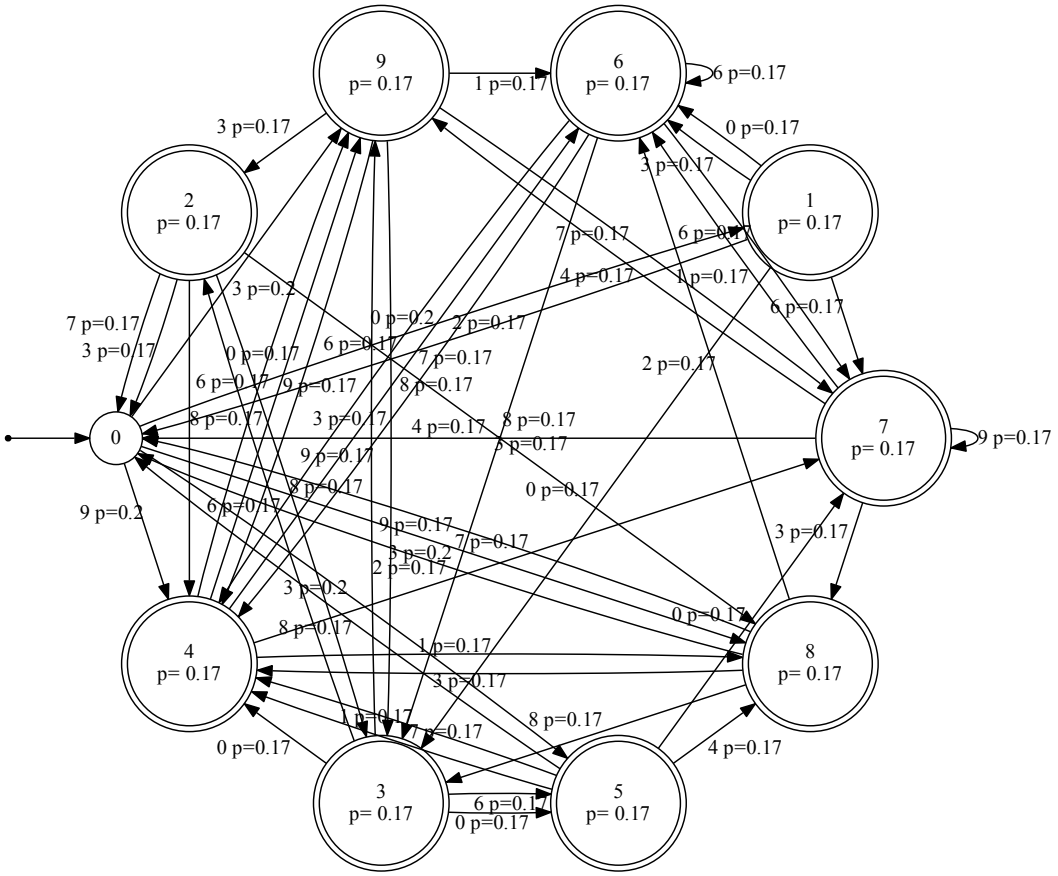


Figure A.1: Sample PDRTA for the runtime evaluation with 10 states, 50 transitions and alphabet of size 10 (without time intervals).

A.2.2 Memory Consumption for Different Scaling Approaches

In this section we present the average memory consumption of the scaling approaches from Section 6.6.2.2. As already described, the memory consumption is not only influenced by the algorithm itself but also by the Java Virtual Machine (JVM). We measured the experimental runtime of any algorithm in terms of training time. For the memory consumption, we can only measure the memory consumption of the whole Java process. The memory of the Java process can be occupied by a lot of factors, e.g. the input data, the testing procedure, etc. Additionally, the JVM frees memory from time to time and if it reaches its memory cap. So it may seem that a process consumes a lot of memory because it never reaches its memory cap while the memory is unused but never freed. Hence, the results of the experimental memory

consumption may be misleading, but we show them for the sake of completeness.

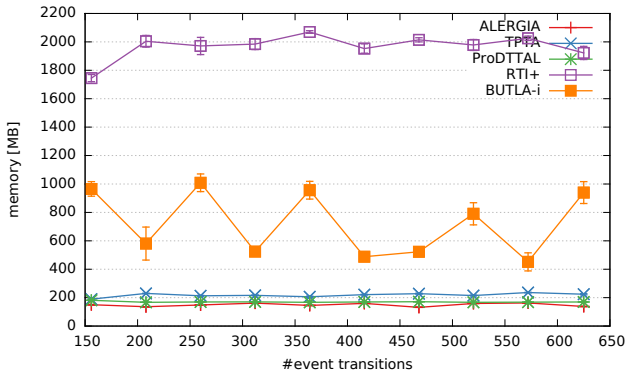
Figure A.2 shows the experimental average memory consumption for the different algorithms and every scaling approach. As for the experimental training runtime, we tuned the hyperparameters with SMAC running for one day for a good Matthews Correlation Coefficient (MCC) (anomaly detection performance) and not for a good memory consumption. The number of event and time-based transitions (Figs. A.2a and A.2b) are very similar and do not heavily influence the memory consumption of any algorithm. RTI+ constantly consumes about 2 GB of memory, while ALERGIA, TPTA and ProDTTAL only need about 200 MB. BUTLA-i oscillates between 400 and 1000 MB but we cannot explain this behavior.

The increase in states (Fig. A.2c) leads to dropping memory consumption for all algorithms. RTI+ and BUTLA-i start with a high memory consumption and drop rapidly, such that all approaches only need between 50 and 200 MB for 120 states. As already mentioned in Section 6.6.4 the increase in states lead to a very simple model that neither requires a lot of time nor space.

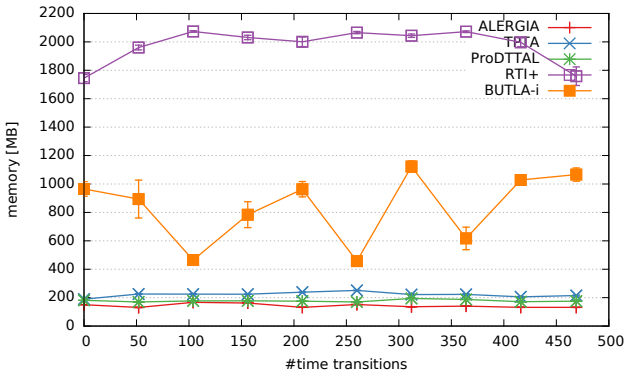
With an increasing alphabet (Fig. A.2d) the memory consumption of ALERGIA, TPTA and ProDTTAL is constant at 200 MB or increases slightly. The memory consumption of RTI+ is high at the beginning and even drops below 200 MB for 110 states and then jumps up and down without any obvious trend. Most of the runs of RTI+ did not finish in the allowed time and only the configuration that did not need much memory completed its run (but lead to a very poor MCC). BUTLA-i starts at 1000 MB for 25 states, and then seems to settle down at 600 MB from 90 states onwards.

For an increasing sample size (Fig. A.2e) BUTLA-i oscillates around 1100 MB but cannot handle more than 14 000 sequences in its original implementation. RTI+ starts at 1800 MB and then decreases to 200 MB and remains there because only a memory-efficient configuration completed in the allowed time. For 14 000 or more sequences no configuration completed in the allowed time. ALERGIA, TPTA and ProDTTAL start at 200 MB and slowly increase to 500 MB for 20 000 samples. As we sample data from an automaton that contains cycles, sampling more sequences eventually leads to longer sequences. Hence, the Prefix Tree Acceptor grows and the algorithms consume slightly more memory.

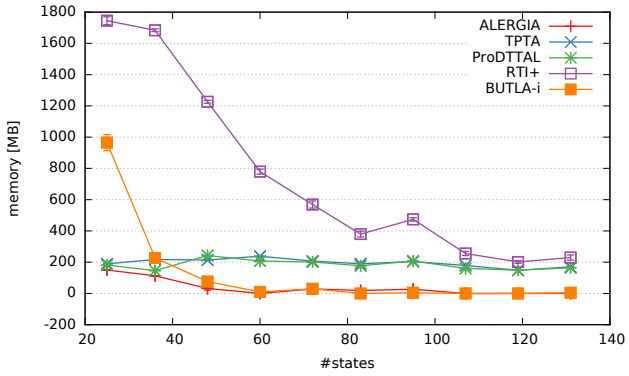
The evaluation of the memory consumption for the different scaling approaches did not yield the expected insights because of the mentioned problems with measuring memory consumption during training. Nevertheless, the experiments show that ALERGIA, TPTA and ProDTTAL require similar amounts of memory, and less memory than BUTLA-i and RTI+.



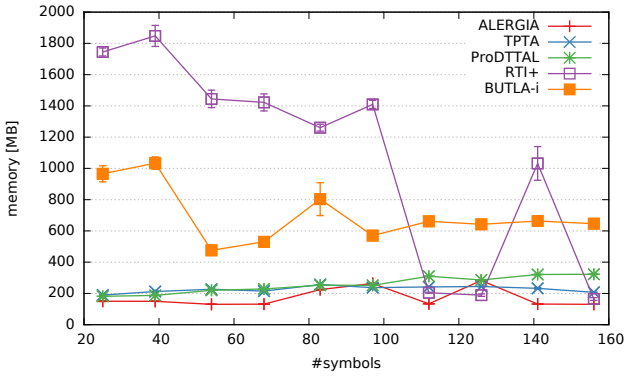
(a) Event transitions.



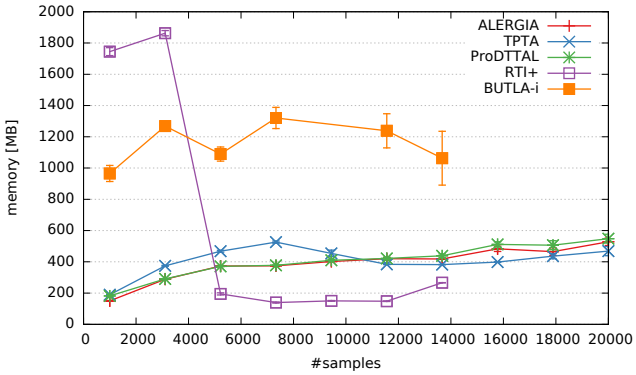
(b) Time-based transitions.



(c) States.



(d) Alphabet.



(e) Sample size.

Figure A.2: Memory consumption of the algorithms for different scaling approaches.

Acronyms

ALERGIA A popular state merging algorithm inferring a PDFA (Algorithm 1 on Page 25; cf. [CO94]).

AmAnDA (Automata-based Anomaly Detection Algorithm): An algorithm using (timed) automata for detecting sequence-based anomalies (Algorithm 7 on Page 78).

ANODA ANOmaly Detection Algorithm (Section 2.5.2.1; cf. [Mai14]).

ATM Automated Teller Machine (also Cash Machine).

BUTLA (Bottom-Up Timed Learning Algorithm): A Probabilistic Deterministic Timed Automaton (PDTA) inference algorithm (Section 2.4.3.1; cf. [Mai14]).

BUTLA-i (Improved Bottom-Up Timed Learning Algorithm): The improved version of BUTLA (Section 6.2.3.2).

DES Discrete Event System(s) (Section 2.2.2; cf. [Cas07]).

DFA Deterministic Finite Automaton (Definition 1 on Page 14; cf. [MP43]).

FTA Frequency Prefix Tree Acceptor (Section 2.4.1.2).

KDE Kernel Density Estimation (Section 2.1; cf. [Par62]).

MCC (Matthews Correlation Coefficient): A metric for an experiment result computed from true/false positives and negatives (Eq. (5.6) on Page 89).

MDI (Minimal Divergence Inference): Another popular state merging algorithm inferring a PDFA (Algorithm 3 on Page 27; cf. [TDH00]).

NFA Non-Deterministic Finite Automaton (Definition 3 on Page 16; cf. [RS59]).

PDFA Probabilistic Deterministic Finite Automaton (Definition 2 on Page 15; cf. [Rab63]).

PDRTA Probabilistic Deterministic Real-Time Automaton (Definition 6 on Page 19; cf. [VdW10]).

PDTA Probabilistic Deterministic Timed Automaton (Definition 5 on Page 18; cf. [Mai14]).

PDTTA Probabilistic Deterministic Timed Transition Automaton (Definition 12 on Page 64).

PNTTA Probabilistic Non-Deterministic Timed Transition Automaton (Definition 16 on Page 110).

PPTA Probabilistic Prefix Tree Acceptor (Section 2.4.1.2).

ProDTTAL (Probabilistic Deterministic Timed Transition Automaton Learning): A PDTTA inference algorithm (Algorithm 6 on Page 69).

PTA Prefix Tree Acceptor (Section 2.4.1.2).

RTI+ (Real-time Identification from Positive Data): A PDRTA inference algorithm (Section 2.4.3.2; cf. [VdW10]).

SMAC (Sequential Model-based Algorithm Configuration): An algorithm for tuning hyperparameters (Section 6.1; cf. [HHL11]).

SWF-net Structured Workflow net (Section 3.2; cf. [Mur89]).

TPTA Timed Prefix Tree Acceptor (Section 5.2.3).

WF-net Workflow net (Section 3.2; cf. [Mur89]).

Notation

α A confidence parameter (usually an input for automata merging algorithms).

Σ The alphabet containing symbols.

δ The transition function of an automaton.

t A transition of an automaton.

e An event (symbol).

F The set of final states of an automaton.

q A single state of an automaton.

Q The set of states of an automaton.

q_0 The start state of an automaton.

s A sequence.

s^+ A positive sequence representing normal behavior.

s^- A negative sequence representing abnormal behavior.

S A set of sequences.

s_t A timed sequence.

S_t A set of timed sequences.

S_t^+ A set of positive timed sequences (representing normal behavior).

S_t^- A set of negative timed sequences (representing abnormal behavior).

$S_t^{+/-}$ A set of positive and negative timed sequences (representing normal and abnormal behavior).

s_e An event sequence (without time values).

S_e A set of event sequences (without time values).

S_e^+ A set of positive (untimed) event sequences representing normal behavior.

S_e^- A set of negative event sequences (without time values) representing abnormal behavior.

$S_e^{+/-}$ A set of positive and negative (untimed) event sequences (representing normal and abnormal behavior).

v A time value.

$\mathbf{x}$ A vector.

$\mathbf{X}$ A set of vectors.

$\mathbf{X}^+$ A set of positive vectors (representing normal behavior).

Bibliography

[Aal05] Wil M. P. van der Aalst. “Business alignment: using process mining as a tool for Delta analysis and conformance testing.” In: *Requirements Engineering* 10.3 (2005), pp. 198–211.

[Aal11] Wil M. P. van der Aalst. *Process Mining*. Springer, 2011.

[Aal98] Wil M. P. van der Aalst. “The Application of Petri Nets to Workflow Management.” In: *Journal of circuits, systems, and computers* 8.01 (1998), pp. 21–66.

[AB01] Wil M. P. van der Aalst and Twan Basten. “Identifying commonalities and differences in object life cycles using behavioral inheritance.” In: *International Conference on Application and Theory of Petri Nets*. Springer, 2001, pp. 32–52.

[AD52] Theodore W. Anderson and Donald A. Darling. “Asymptotic theory of certain "goodness of fit" criteria based on stochastic processes.” In: *The annals of mathematical statistics* (1952), pp. 193–212.

[AD94] Rajeev Alur and David L. Dill. “A Theory of Timed Automata.” In: *Theoretical Computer Science* 126.2 (1994), pp. 183–235.

[Aka98] Hirotogu Akaike. “Information theory and an extension of the maximum likelihood principle.” In: *Selected Papers of Hirotugu Akaike*. Springer, 1998, pp. 199–213.

[AM05] Wil M. P. van der Aalst and Ana Karla A. de Medeiros. “Process mining and security: Detecting anomalous process executions and checking process conformance.” In: *Electronic Notes in Theoretical Computer Science* 121 (2005), pp. 3–21.

[And+14] Maik Anderka, Timo Klerx, Steffen Priesterjahn, and Hans Kleine Büning. “Automatic ATM Fraud Detection as a Sequence-based Anomaly Detection Problem.” In: *Proceedings of the 3rd International Conference on Pattern Recognition Applications and Methods (ICPRAM’14)*. SciTePress, 2014.

[AS11] Wil M. P. van der Aalst and Christian Stahl. *Modeling Business Processes: A Petri Net-Oriented Approach*. The MIT Press, 2011.

[AS12] A. Azarian and A. Siadat. “A global modular framework for automotive diagnosis.” In: *Advanced Engineering Informatics* 26.1 (2012), pp. 131–144.

[ASS11] Wil M. P. van der Aalst, M. Helen Schonenberg, and Minseok Song. “Time prediction based on process mining.” In: *Information Systems* 36.2 (2011), pp. 450–475.

[AWM04] Wil M. P. van der Aalst, Ton Weijters, and Laura Maruster. “Workflow mining: Discovering process models from event logs.” In: *IEEE Transactions on Knowledge and Data Engineering* 16.9 (2004), pp. 1128–1142.

[Bai11] Raphael Bailly. "Quadratic weighted automata: Spectral algorithm and likelihood maximization." In: *Journal of Machine Learning Research* 20 (2011), pp. 147–162.

[Bau+70] Leonard E. Baum, Ted Petrie, George Soules, and Norman Weiss. "A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains." In: *The Annals of Mathematical Statistics* 41.1 (1970), pp. 164–171.

[BCG12] Borja Balle, Jorge Castro, and Ricard Gavaldà. "Bootstrapping and Learning PDFA in Data Streams." In: *Proceedings of the 11th International Conference on Grammatical Inference (ICGI'12)*. 2012, pp. 34–48.

[BGV92] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. "A Training Algorithm for Optimal Margin Classifiers." In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory (COLT '92)*. ACM, 1992, pp. 144–152.

[Bif+10] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. "MOA: Massive Online Analysis." In: *Journal of Machine Learning Research* 11 (2010), pp. 1601–1604.

[BJ06] David M. Blei and Michael I. Jordan. "Variational inference for Dirichlet process mixtures." In: *Bayesian analysis* 1.1 (2006), pp. 121–144.

[Bre01] Leo Breiman. "Random Forests." In: *Machine Learning* 45.1 (2001), pp. 5–32.

[Bud+06] Suratna Budalakoti, Ashok N. Srivastava, Ram Akella, and Eugene Turkov. *Anomaly Detection in Large Sets of High-dimensional Symbol Sequences*. Tech. rep. TM-2006-214553. NASA Ames Research Center, Moffett Field, California, 2006.

[Cas07] Christos G. Cassandras. *Introduction to discrete event systems*. Springer, 2007.

[CG08] Jorge Castro and Ricard Gavaldà. "Towards feasible PAC-learning of probabilistic deterministic finite automata." In: *Grammatical Inference: Algorithms and Applications*. Springer, 2008, pp. 163–174.

[Cho56] Noam Chomsky. "Three models for the description of language." In: *IRE Transactions on Information Theory* 2.3 (1956), pp. 113–124.

[CLM01] João B. D. Cabrera, Lundy Lewis, and Raman K. Mehra. "Detection and classification of intrusions and faults using sequences of system calls." In: *ACM SIGMOD Record* 30.4 (2001), pp. 25–34.

[CMA03] Philip K. Chan, Matthew V. Mahoney, and Muhammad H. Arshad. *A Machine Learning Approach to Anomaly Detection*. Tech. rep. CS–2003–06, Department of Computer Science, Florida Institute of Technology, Melbourne, 2003.

[CMK08] Varun Chandola, Varun Mithal, and Vipin Kumar. "Comparative Evaluation of Anomaly Detection Techniques for Sequence Data." In: *Proceedings of the 8th International Conference on Data Mining (ICDM'08)*. IEEE, 2008, pp. 743–748.

[CO94] Rafael C. Carrasco and José Oncina. "Learning Stochastic Regular Grammars by Means of a State Merging Method." In: *Proceedings of the 2nd International Colloquium on Grammatical Inference and Applications (ICGI'94)*. Springer, 1994, pp. 139–152.

[CT04] Alexander Clark and Franck Thollard. "PAC-learnability of Probabilistic Deterministic Finite State Automata." In: *Journal of Machine Learning Research* 5 (2004), pp. 473–497.

[DE15] Ayhan Demiriz and Betul Ekizoglu. "Using location aware business rules for preventing retail banking frauds." In: *Proceedings of the 1st International Conference on Anti-Cybercrime (ICACC'15)*. IEEE, 2015, pp. 1–6.

[Don+05] Boudewijn F. van Dongen, Ana Karla A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and Wil M. P. van der Aalst. "The ProM framework: A new era in process mining tool support." In: *Proceedings of the 26th International Conference on the Applications and Theory of Petri Nets (ICATPN'05)*. Springer, 2005, pp. 444–454.

[DS06] Wentao Dong and Youngsung Soh. "Image-based fraud detection in automatic teller machine." In: *International Journal of Computer Science and Network Security* 6.11 (2006), pp. 13–18.

[DT00] Colin De La Higuera and Franck Thollard. "Identification in the limit with probability one of stochastic deterministic finite automata." In: *Grammatical Inference: Algorithms and Applications*. Springer, 2000, pp. 141–156.

[Dub+04] Didier Dubois, Laurent Foulloy, Gilles Mauris, and Henri Prade. "Probability-possibility transformations, triangular fuzzy sets, and probabilistic inequalities." In: *Reliable computing* 10.4 (2004), pp. 273–297.

[Est+96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. "A density-based algorithm for discovering clusters in large spatial databases with noise." In: *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD'96)*. AAAI Press, 1996, pp. 226–231.

[FS95] Yoav Freund and Robert E Schapire. "A desicion-theoretic generalization of on-line learning and an application to boosting." In: *European conference on computational learning theory (EuroCOLT'95)*. Springer, 1995, pp. 23–37.

[Gib15] Samuel Gibbs. *Airbus issues software bug alert after fatal plane crash.* `https://www.theguardian.com/technology/2015/may/20/airbus-issues-alert-software-bug-fatal-plane-crash`. [Online; accessed 25-April-2016]. 2015.

[Gol67] E. Mark Gold. "Language identification in the limit." In: *Information and Control* 10.5 (1967), pp. 447–474.

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

[Gre15] Andy Greenberg. *Hackers Remotely Kill a Jeep on the Highway—With Me in It*. `http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/`. [Online; accessed 25-April-2016]. 2015.

[Hal+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. "The WEKA data mining software: an update." In: *ACM SIGKDD explorations newsletter* 11.1 (2009), pp. 10–18.

[Han09] David J. Hand. "Measuring classifier performance: a coherent alternative to the area under the ROC curve." In: *Machine Learning* 77.1 (2009), pp. 103–123.

[HE04] Greg Hamerly and Charles Elkan. "Learning the k in k-means." In: *Advances in neural information processing systems* 16 (2004), p. 281.

[HHL11] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. "Sequential model-based optimization for general algorithm configuration." In: *Learning and Intelligent Optimization*. Springer, 2011, pp. 507–523.

[HLV03] Wenjie Hu, Yihua Liao, and V. Rao Vemuri. "Robust Support Vector Machines for Anomaly Detection in Computer Security." In: *Proceedings of the International Conference on Machine Learning and Applications (ICMLA'03)*. 2003, pp. 168–174.

[Hoe63] Wassily Hoeffding. "Probability inequalities for sums of bounded random variables." In: *Journal of the American statistical association* 58.301 (1963), pp. 13–30.

[Hul12] Mans Hulden. "Treba: Efficient Numerically Stable EM for PFA." In: *Proceedings of the 11th International Conference on Grammatical Inference (ICGI'12)*. 2012, pp. 249–253.

[ISS00] Katsuhiko Ishiguro, Hiroshi Sawada, and Hitoshi Sakano. "Multi-class boosting for early classification of sequences." In: *Statistics* 28.2 (2000), pp. 337–407.

[JSW98] Donald R. Jones, Matthias Schonlau, and William J. Welch. "Efficient global optimization of expensive black-box functions." In: *Journal of Global Optimization* 13.4 (1998), pp. 455–492.

[KAK14] Timo Klerx, Maik Anderka, and Hans Kleine Büning. "On the usage of behavior models to detect ATM fraud." In: *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI'14). IOS Press*. 2014, pp. 1045–1046.

[Kle+14] Timo Klerx, Maik Anderka, Hans Kleine Büning, and Steffen Priesterjahn. "Model-Based Anomaly Detection for Discrete Event Systems." In: *Proceedings of the 26th International Conference on Tools with Artificial Intelligence (ICTAI'14)*. IEEE, 2014, pp. 665–672.

[KMB12] Fábio Natanael Kepler, Sergio L. S. Mergen, and Cleo Zanella Billa. "Simple Variable Length N-grams for Probabilistic Automata Learning." In: *Proceedings of the 11th International Conference on Grammatical Inference (ICGI'12)*. 2012, pp. 254–258.

[KMD10] Kenneth Kennedy, Brian Mac Namee, and Sarah Delany. "Learning without Default: A Study of One-Class Classification and the Low-Default Portfolio Problem." In: *Artificial Intelligence and Cognitive Science* (2010), pp. 174–187.

[KW95] Robert E. Kass and Larry Wasserman. "A reference Bayesian test for nested hypotheses and its relationship to the Schwarz criterion." In: *Journal of the american statistical association* 90.431 (1995), pp. 928–934.

[Lip+00] Richard Lippmann, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das. "The 1999 DARPA off-line intrusion detection evaluation." In: *Computer networks* 34.4 (2000), pp. 579–595.

[Llo82] Stuart P. Lloyd. "Least squares quantization in PCM." In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137.

[Mai+11] Alexander Maier, Asmir Vodenčarević, Oliver Niggemann, R. Just, and M. Jäger. "Anomaly Detection in Production Plants Using Timed Automata." In: *Proceedings of the 8th International Conference on Informatics in Control, Automation and Robotics (ICINCO'11)*. SciTePress, 2011, pp. 363–369.

[Mai14] Alexander Maier. "Identification of Timed Behavior Models for Diagnosis in Production Systems." PhD thesis. University of Paderborn, 2014.

[Mas+10] Mohammad M. Masud, Qing Chen, Jing Gao, Latifur Khan, Jiawei Han, and Bhavani Thuraisingham. "Classification and novel class detection of data streams in a dynamic feature space." In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD'17)*. Springer, 2010, pp. 337–352.

[Mas+13] Mohammad M. Masud, Qing Chen, Latifur Khan, Charu C. Aggarwal, Jing Gao, Jiawei Han, Ashok Srivastava, and Nikunj C. Oza. "Classification and Adaptive Novel Class Detection of Feature-Evolving Data Streams." In: *IEEE Transactions on Knowledge and Data Engineering* 25.7 (2013), pp. 1484–1497.

[Men+10] Ole J. Mengshoel, Mark Chavira, Keith Cascio, Scott Poll, Adnan Darwiche, and Serdar Uckun. "Probabilistic Model-Based Diagnosis: An Electrical Power System Case Study." In: *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans* 40.5 (2010), pp. 874–885.

[Mer09] James Mercer. "Functions of positive and negative type, and their connection with the theory of integral equations." In: *Philosophical transactions of the royal society of London. Series A, containing papers of a mathematical or physical character* 209 (1909), pp. 415–446.

[MP43] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity." In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[Mur89] Tadao Murata. "Petri nets: Properties, analysis and applications." In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580.

[Nig+12] Oliver Niggemann, Benno Stein, Asmir Vodenčarević, Alexander Maier, and Hans Kleine Büning. "Learning Behavior Models for Hybrid Timed Systems." In: *Proceedings of the 26th International Conference on Artificial Intelligence (AAAI'12)*. AAAI, 2012, pp. 1083–1090.

[OG92] José Oncina and Pedro García. "Identifying regular languages in polynomial time." In: *Advances in Structural and Syntactic Pattern Recognition* 5 (1992), pp. 99–108.

[OW07] Joel Ouaknine and James Worrell. "On the decidability and complexity of Metric Temporal Logic over finite words." In: *Logical Methods in Computer Science* 3.1 (2007).

[Pap16] Max Papisch. "Anomalieerkennung in zeitlichen Eventsequenzen mit Hilfe von BUTLA." Bachelor thesis. Paderborn University, 2016.

[Par62] Emanuel Parzen. "On estimation of a probability density function and mode." In: *The Annals of Mathematical Statistics* (1962), pp. 1065–1076.

[Pla83] Robin L. Plackett. "Karl Pearson and the chi-squared test." In: *International Statistical Review/Revue Internationale de Statistique* (1983), pp. 59–72.

[PM00] Dan Pelleg and Andrew W. Moore. "X-means: Extending K-means with Efficient Estimation of the Number of Clusters." In: *Proceedings of the Seventeenth International Conference on Machine Learning (ICML'00)*. 2000, pp. 727–734.

[Pow11] David M. W. Powers. "Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation." In: *International Journal of Machine Learning Technology* 2.1 (2011), pp. 37–63.

[Pre+96] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C*. Vol. 2. Cambridge university press, 1996.

[Pri+15] Steffen Priesterjahn, Maik Anderka, Timo Klerx, and Uwe Mönks. "Generalized ATM Fraud Detection." In: *Proceedings of the 15th Industrial Conference on Data Mining (ICDM'15)*. Springer, 2015, pp. 166–181.

[Rab63] Michael O. Rabin. "Probabilistic automata." In: *Information and Control* 6.3 (1963), pp. 230–245.

[RNM12] Ben Reardon, Kara Nance, and Stephen McCombie. "Visualization of ATM Usage Patterns to Detect Counterfeit Cards Usage." In: *Proceedings of the 45th Hawaii International Conference on System Science (HICSS'12)*. IEEE, 2012, pp. 3081–3088.

[RS59] Michael O. Rabin and Dana Scott. "Finite Automata and Their Decision Problems." In: *IBM Journal of Research and Development* 3.2 (1959), pp. 114–125.

[San+15] Jayro Santiago-Paz, Deni Torres-Roman, Angel Figueroa-Ypiña, and Jesus Argaez-Xool. "Using Generalized Entropies and OC-SVM with Mahalanobis Kernel for Detection and Classification of Anomalies in Network Traffic." In: *Entropy* 17.9 (2015), pp. 6239–6257.

[SC78] Hiroaki Sakoe and Seibi Chiba. "Dynamic programming algorithm optimization for spoken word recognition." In: *IEEE transactions on acoustics, speech, and signal processing* 26.1 (1978), pp. 43–49.

[Sch+99] Bernhard Schölkopf, Robert C. Williamson, Alexander J. Smola, John Shawe-Taylor, and John C. Platt. "Support Vector Method for Novelty Detection." In: *13th Annual Neural Information Processing Systems Conference (NIPS'99)*. Vol. 12. MIT Press, 1999, pp. 582–588.

[SHX02] Qing Song, Wenjie Hu, and Wenfang Xie. "Robust support vector machine with bullet hole image classification." In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 32.4 (2002), pp. 440–448.

[Sil86] Bernard W. Silverman. *Density estimation for statistics and data analysis*. Vol. 26. CRC press, 1986.

[SY12] Chihiro Shibata and Ryo Yoshinaka. "Marginalizing Out Transition Probabilities for Several Subclasses of PFAs." In: *Proceedings of the 11th International Conference on Grammatical Inference (ICGI'12)*. 2012, pp. 259–263.

[TDH00] Franck Thollard, Pierre Dupont, and Colin de la Higuera. "Probabilistic DFA Inference Using Kullback-Leibler Divergence and Minimality." In: *Proceedings of the Seventeenth International Conference on Machine Learning (ICML'00)*. 2000, pp. 975–982.

[Val84] Leslie G. Valiant. "A theory of the learnable." In: *Communications of the ACM* 27.11 (1984), pp. 1134–1142.

[Vat11] Andrea Vattani. "K-means requires exponentially many iterations even in the plane." In: *Discrete & Computational Geometry* 45.4 (2011), pp. 596–616.

[VdW10] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. "A Likelihood-ratio Test for Identifying Probabilistic Deterministic Real-time Automata from Positive Data." In: *Proceedings of the 10th International Colloquium Conference on Grammatical Inference: Theoretical Results and Applications (ICGI'10)*. Springer, 2010, pp. 203–216.

[VDW11] Sicco Verwer, Mathijs De Weerdt, and Cees Witteveen. "Learning driving behavior by timed syntactic pattern recognition." In: *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*. AAAI, 2011, pp. 1529–1534.

[VEH14] Sicco Verwer, Rémi Eyraud, and Colin de la Higuera. "PAutomaC: a probabilistic automata and hidden Markov models learning competition." In: *Machine learning* 96.1-2 (2014), pp. 129–154.

[Vod13] Asmir Vodenčarević. "Identifying behavior models for hybrid production systems." PhD thesis. Paderborn, 2013.

[VWW11] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. "Efficiently identifying deterministic real-time automata from labeled data." In: *Machine Learning* 86.3 (2011), pp. 295–333.

[Wel03] Lloyd R. Welch. "Hidden Markov models and the Baum-Welch algorithm." In: *IEEE Information Theory Society Newsletter* 53.4 (2003), pp. 10–13.

[WFP99] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. "Detecting Intrusions Using System Calls: Alternative Data Models." In: *Proceedings of the 1999 IEEE Symposium on Security and Privacy*. IEEE, 1999, pp. 133–145.

[WS04] Ke Wang and Salvatore J. Stolfo. "Anomalous Payload-Based Network Intrusion Detection." In: *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID'04)*. Springer, 2004, pp. 203–222.

[WW12] Dominik Wied and Rafael Weißbach. "Consistency of the kernel density estimator: a survey." In: *Statistical Papers* 53.1 (2012), pp. 1–21.

[Yan+09] Yuhong Yan, Philippe Dague, Yannick Pencolé, and Marie-Odile Cordier. "A Model-Based Approach for Diagnosing Fault in Web Service Processes." In: *International Journal of Web Services Research (IJWSR'09)* 6.1 (2009), pp. 87–110.

[Zho+03] Aoying Zhou, Zhiyuan Cai, Li Wei, and Weining Qian. "M-kernel merging: Towards density estimation over data streams." In: *Proceedings of the 8th International Conference on Database Systems for Advanced Applications (DASFAA 2003)*. IEEE, 2003, pp. 285–292.