Log Differencing using State Machines for Anomaly Detection

Log Differencing using State Machines for Anomaly Detection

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Sofia Tsoni

Master of Engineering in Computer Engineering,
University of Patras, Greece,
born in Athens, Greece.

Supervisor:

Dr. S. Verwer

Company Supervisor:

Ir. R. Wieman

Thesis Committee:

Prof. dr. A. van Deursen,	Technische Universiteit Delft
Dr. ir. S. Verwer,	Technische Universiteit Delft
Dr. ir. M. Aniche,	Technische Universiteit Delft
Ir. R. Wieman,	Adyen

The work in the thesis was conducted together with Adyen.

Style: TU Delft House Style, with modifications by Moritz Beller
`https://github.com/Inventitech/phd-thesis-template`

An electronic version of this dissertation is available at
`http://repository.tudelft.nl/`.

Contents

Summary

Huge amounts of log data are generated every day by software. These data contain valuable information about the behavior and the health of the system, which is rarely exploited, because of their volume and unstructured nature. Manually going through log files is a time-consuming and labor-intensive procedure for developers. Nonetheless logging information can expose the problematic execution of the software, even though the final outcome seem to be normal. Nowadays the automatic analysis of the log files is crucial for detecting problems, but mainly for understanding how the software behaves, which would be beneficial for the prevention of failures and improvement of the software itself. Towards that direction, this project aims the identifications of unexpected executions of the software and the determination of the root cause behind them. In more details, the expected behavior of the software can be approximated using model inference techniques and the newly incoming observed data can be analyzed to verify if they are conformed by the expected behavior.

The conformance checking method that will be used is called replay. The incoming traces will be replayed in the graph, at the point they are not validated, the alignment algorithm will take over. The sequence alignment is performed in three different ways. Two of the methods are looking for the best alignment at a specific radius around the problematic node. Additionally a global alignment technique is implemented, which is based on the famous algorithm by Needleman and Wunsch for DNA sequences. Our goal required the modification of the aforementioned algorithm to not only align two sequences, but a sequence with a tree structured model.

Finally the implemented tool provides the visualization of the differences in a way that makes it intuitive for the developers to understand what went wrong. Some additional information are also provided to make the investigation of the "anomaly" easier .

Acknowledgments

Many people contributed to this thesis in the one way or the other.

First of all, my supervisor Sicco Verwer, who was always reachable and willing to help. With his enthusiasm and his fascinating ideas he showed me how overcome the million problems that felt unsolvable at that time.

I want to thank everyone from the group for the weekly meeting. It was a great opportunity to meet people who are going through the same struggles and share our concerns and ideas.

Of course I want to thank everyone from Adyen. Being a member of this awesome team was a great experience both in personal and professional level. Especially I want to thank Rick Wieman, Daan Schipper and Jesse Fekkes for making time to discuss about my project and for providing valuable input.

Last but not least, I am grateful to my family and friends who have been standing by my side all these years and I hope they will continue being there for many more!

Sofia
Delft, August 2019

1

Introduction

Nowadays every system creates huge amounts of logs, which probably will never be investigated by a human eye. However, they contain information that may be extremely useful to understand the outcome of the executing processes and the overall behavior of the system. Using the log data, various conclusions about the running software, that produced them, can be drawn. For example, log data can be useful for the detection of potential anomalies, which may be caused by an unexpected system failure or a malicious action.

Log files contain unstructured, textual information, which makes their processing, a quite expensive procedure. Nonetheless it is worth the effort, since there are multiple uses of the included information.

This study seeks to address the issue of automatic differencing between log files. In more detail, it is the comparison between log files, in order to identify flows that were not supposed to happen either in the one or the other file. This tool can have multiple uses, like anomaly detection in recently introduced software, or improvement of test coverage, by identifying untested cases in the live environment. However, since multiple log files are generated each day, the naive approach of comparing files can not be applied in our scenarios.

The aforementioned problem will be approached by modeling the behavior of the software using its log data and compare this model (which will be called specification graph) with other log entries, on-the-fly, which may be produced by a different software or a different software version. The modeling of the software's behavior is a problem that has been researched multiple times by previous works, thus one specific way from the literature will be chosen and applied.

The main challenge that this thesis trying to address, is the development of a method that will be able to perform this comparison, with the ultimate goal to detect the flows that deviate from the expected behavior. Furthermore, it will be crucial to find also the root cause of that deviation.

1

1.1 Motivation

Every day, trillions of lines of log data are created. A single system may produce more than 10 lines for one operation, like one card transaction. This operation may consists of multiple steps that are logged, in order for the transaction to be authorized and processed. These logs contain crucial information regarding the necessary underlying operations of the corresponding software. Developers look into these files only when something critical happens, that hinders the overall execution of the software. Nonetheless, many things could have been wrongly executed, which do not impact the final state of the process [1–3]. For example, such a case is in a credit card transaction, where for some reason no PIN was asked (skipped the authentication part). The transaction will be performed without a problem, but if someone looks into its logging, an anomaly will be found. Manual investigation of the logs takes a huge amount of time and may also be too complicated for human brains. For that purpose we investigate ways in which this information can be extracted for the identification of differences between software behavior.

In the era of big data, it is critical to find ways to extract information from such a huge pool of data. Related work though [4, 5], has proven the importance of tools that make use of the log files to give insights about the software's execution processes from another point of view. It is also very important in order to verify, that the implemented software works according to its specifications. This can be accomplished by comparing the expected behavior, based on the specifications, with the actual behavior of the software, based on the execution.

1.2 Problem Statement

The goal of this thesis is to develop an automatic anomaly detection tool for software's behavior. The software used will be for processing payments, more specifically the one running at Adyen's POS terminals. The work by Wieman [6] will be used as our preprocessing step, before applying the comparison methods. In his project, Wieman tested different passive learning tools to infer graph models from Adyen's log files, which represent transactions carried out using the POS terminals. These models make it possible for the developer to see with a quick look all the paths that are taken in a system and makes the monitoring of the log files much easier. Wieman compared the tools in terms of runtime performance and output complexity. He also touched the topic of inferred model's comparison for anomaly detection purposes. His approach was annotating whole traces as new or not, which is not very helpful if one wants to understand exactly what was the issue that led to that deviating behavior.

A main part of this thesis is researching the different ways this comparison can be performed. Nowadays, checking if the execution of a system is the expected has become difficult because of the many different paths the execution may take. In this thesis, first the algorithm will be designed and implemented and then it will be applied for two different scenarios in our industry partner, Adyen.

First we want to compare if the testing environment covers well enough the real cases. Thus we have to compare the model learned by the testing log data, which are

transactions performed by robots, with real log files, produced by actual transactions with users. This will prove if the test coverage of Adyen is good enough, or if there are transaction cases that are not covered.

Secondly, we will compare different software releases, with the goal to get some insights about what changed in the new release and if everything went as planned. If the results of this comparison are successful, a log differencing tool could be used to monitor the release procedure and automatically make sure the software is ready to be released on the field.

However each of these cases presents different challenges, which will be extensively analyzed later in this thesis.

Both the aforementioned scenarios require a tool that will be able to accept as input the files under comparison and output their differences. The question is what kind of algorithms can be used to perform such an action and how efficient they can be for big data, like the log files from a big company. Finally we will touch upon the issue of visualizing these differences. Just finding the differences is not enough when we have data that can be represented by a graph, because then their textual representation does not make much sense. Thus a method for the intuitive visualization of the differences will be developed, according to the needs of the user.

1.3 Industry Partner

Adyen is a payment service provider (PSP). A PSP is a party, which offers payment services to merchants. A payment is *the action or process of paying someone or something or of being paid*. More precisely in Adyen's case the payment is the action of transferring money from a shoppers bank account to a merchants bank account. There are various ways to perform the transfer like using a credit or a debit card or direct credit etc. A sale can be either physical or electronic, with the corresponding channels being point of sale (POS) and eCommerce (eCom) respectively.

This project will focus on the POS channel, whose setup can be found in Figure 1.1. It consists of a cash register, operated by a cashier and a payment terminal used by shoppers. The cash register is commonly referred to as POS and the payment terminal as PIN Entry Device (PED).

The cash register initiates a transaction by communicating with the payment terminal through Adyen's library. The terminal, which runs Adyen's software, interacts with the shopper and requests authorization based on the different methods from the platform. The platform then communicates with the corresponding bank.

In this project we consider logs generated by the aforementioned setup, that detail each step of the transaction, like the actions of the user, the communication between the POS and the PED using Adyen's library as well as the requests and the responses of Adyen's platform.

1

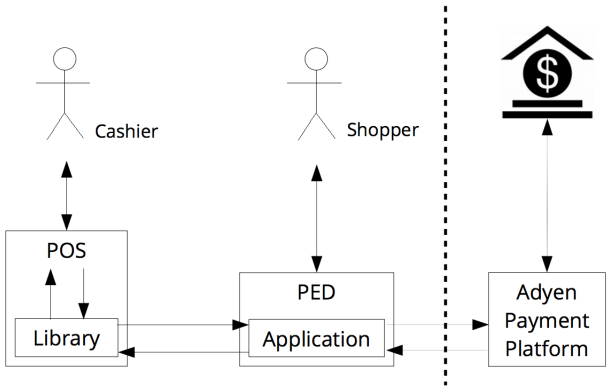


Figure 1.1: Overview of POS solution, by Wieman [6]

1.4 Proposed Solution

To solve the comparison problem, at the beginning, Model Checkers were considered. The idea was to provide the Model Checker with the two model that had to be compared and use their equivalence checking feature. The equivalence checking returns a counterexample if the two models are not equivalent and that counterexamples could be used to relearn a model and then compare the updated versions again to find another difference. However this is extremely time consuming as it will be discussed in Chapters 10 and 9.

Since the scalability of the proposed tool is an important factor, because of the large amounts of data we are dealing with, the chosen approach was based mainly on that motivation. Namely, a method that would output the differences of the corresponding logs without needing too much time is needed. For that reason, structural comparison of the models did not seem as a good approach, since because of their size, comparing every structural element of their graphical representation is a very heavy task. Thus it was decided to use a validation method, conformance checking, to approach the log differencing problem from a completely different angle.

Towards that direction, the input data of the algorithm should be one model that would represent the expected behavior of the software and the observed sequences of the events we want to validate. The result of that comparison should be the differences between the two behaviors and the root cause that led to that difference. Identifying the cause of the deviation is not always an easy case, because of the complexity of the data. Some challenges of this problem are mentioned in Chapter 6.

The proposed solution suggests conformance checking with sequence alignment between the model traces and the (incoming) observed log data. Multiple algorithms were used for the identification of the best alignment, which are compared in Chapter 7.

1.5 Research Questions

Some of the research questions that will approached in this thesis are:

- How can software's behavior comparison be performed for large amounts of data?
- What kind of differences can be identified?
- How can the differences of the software's behavior be visualized effectively?
- What kind of data are more suitable for the specification graph (to build the model)?
- How does the model size affect the results?

1.6 Contributions

The most important contributions of our work are presented in this section.

- We propose a pipeline for on-the-fly log differencing, that has not been used before, to the best of our knowledge.
- We implemented the tool that receives as input the model and the log data and outputs the differences ready for visualization.
- More than one algorithm, for the identification of the differences, are implemented, all with different advantages and disadvantages.
- One of the most famous sequence alignment algorithms was modified to align sequence with a tree, instead of sequence with sequence.
- We propose and implement an efficient way of visualizing the differences between log files, using prefix trees and the d3.js library.
- We developed the necessary code to modify a widely used graph description language to the language of one of the most popular model checkers.

1.7 Report Organization

In Chapter 2 the required background knowledge, for the reader to understand the topics of this thesis, is presented. In Chapter 3 previous studies on log analysis and log differencing are mentioned, as well as related work on conformance checking and sequence alignment, two terms we will use in the proposed methodology. Chapter 3.7 contains related work for the problem of state machine comparison. In this chapter, we focus on the different methods the comparison can be performed and which are the advantages and disadvantages of each one of them. This chapter is preliminary, in order to understand what led us in the decision to chose the proposed solution.

In Chapter 4 the format of the log files used for our research will be shown, and the dataset will be described. In addition to that, we do some data analysis

to understand the nature of the data better. Finally, the findings of our manual exploration will be presented, and the types of differences we are trying to identify based on this exploration.

Chapter 5 touches upon the ways we decided to model the software's behavior, and why these decisions were made using some theory about how the learning algorithm works and some experiments too. Chapter 10 is about model checkers and how they can be used for the model comparison purpose. Chapter 6 presents the main contribution of this thesis, how the pipeline of the proposed log differencing tool works based on conformance checking. We present three different algorithms that tackle the same problem, some with better accuracy some with better runtime performance. We build upon the simple algorithm and we present the final methodology in that chapter. The next chapter, Chapter 7 shows the performance of the aforementioned algorithms in terms of runtime and accuracy/recall. In Chapter 9 the conclusions of our work, the threats to validity and the future work will be presented.

For the evaluation purposes some interviews were conducted. The questions of the interview together with the results are presented in Chapter 8.

2 Background

In this chapter the necessary terms and concepts, which are used in this thesis will be explained. The chapter starts with the most preliminary concepts and continues with more complicated notions, which were necessary for the development of the final methodology.

2.1 Sequential Data

A discrete/symbolic sequence is defined as a finite sequence of events, such that each event can be represented as a symbol belonging to a finite alphabet [7]. In other words, a sequence is an ordered list of items of the same type. Sequences contain subsequent values of a specific variable, therefore are ideal for capturing the behavior of a variable. Sequential data can be ordered according to time, like time series or just according to their position like text or genes. In both cases the structural dependencies are extremely important, because that is what describes the behavior of the variable and should be at any cost preserved.

In this project we will deal with discrete sequences, where the order of the terms represents the order the actions happened.

2.1.1 Alphabet

The items of a sequence may belong to a simple alphabet, like the events of the DNA sequences belong to a four-letter alphabet A, C, G, T or a more complex alphabet like the transaction steps, in order to process a payment.

2.1.2 Challenges of discrete sequence data

The first challenge faced, when working with discrete sequence data is the size of the alphabet. For long sequences the alphabet size can be very large, which makes some computations difficult. Thus the computational complexity is a significant issue, usually faced when dealing with sequence data. In our case we do not know in advance the size of the alphabet, since we are dealing with various software releases the size is not constant and may change from run to run.

It is also important to know the types of anomalies that can be detected in a sequence. Suppose it is known that a normal sequence is $S = a_1, a_2, .., a_n$ getting the events of the sequence in the wrong order might be an anomaly like $S' = a_n, a_2, ..., a_1$, for example setting a transaction as approved before authenticating the payment. Another anomaly could be a sequence $S' = b_1, b_2, ..., b_n$, never faced before. Finally there could be a part of a sequence that is anomalous, like $S' = a_1, a_2, ..., b_m$, where it might be a transaction that its final state was supposed to be a_n but the state b_m was encountered, like instead of getting a `DECLINED` final state when there is not insufficient balance, getting an `APPROVED`.

In this project, the most frequently encountered type will be the anomalous subsequences (the latter example). It is not expected that the whole sequence will be anomalous in a transaction flow, neither just a single events. On the other hand, it is the subsequence of some particular (normal) events that may lead to abnormal behaviour.

The problem when dealing with subsequences is that it is not known how long that sequence may be. Finding the length of the anomalous subsequences may be a great challenge by itself.

2.2 Finite State Automata

The models that will be used in the current project are called Finite State Automata (FSA), or State Machines, or Finite State Machines (FSM). The formal definition for a Finite State Automaton is as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where each term:

1. Q is a finite set of states.
2. Σ is a finite alphabet (the set of symbols used as input).
3. δ is a transition function which maps a state and an input symbol to the next state $\delta : Q \times \Sigma \to Q$.
4. q_0 is the start state, where $q_0 \in Q$
5. F is the set of *accepting* or *final* states, where $F \subseteq Q$

The FSA reads the input string one character at a time, and moves to the next appropriate state according to the transition function (or guesses the next state, if there are multiple options).

For a given input $\sigma = \sigma_1\sigma_2...\sigma_n$ of length n, if there is a sequence of transition $q_0\delta(q_0, \sigma_1)q_1...q_n$, where q_0 is the start state and there is another state $q_n \in F$. If q_n is an accepting state, the computation is accepting and the automato A does accept the sequence w. Otherwise w is rejected by A.

An Automaton is called deterministic (Deterministic Finite Automaton or DFA) when there exists exactly one accepting or rejecting path for every possible input. The set of a words accepted by A is called the language of the automaton. In this work, we will mainly deal with DFAs, if there is another type of Automaton it will be clearly stated.

Language of a DFA: The language L of an automaton A is the set of words from the alphabet it accepts.

DFAs can be represented by a graph with nodes representing states and edges representing transitions. Transitions are labeled with the corresponding transition symbol. Typically, final states are denoted by a double circle as can be seen in Figure 2.1, where q_4 is the final state. The start states are indicated by an incoming arrow without source, or with no incoming arrows at all, state q_1 in the following example 2.1. This thesis only considers deterministic automata, if a different type is mentioned it will be clearly stated.

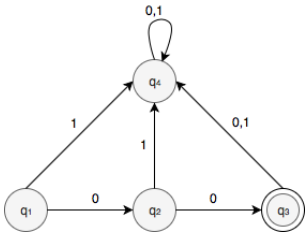


Figure 2.1: An automaton with states q_1, q_2, q_3, q_4 ans transitions 0,1, which only accepts 00

The learning procedure, in order to obtain the corresponding DFA, can be performed either using the passive or the active learning approach.

2.2.1 Passive Learning

Passive Learning is the most common way of learning in machine learning (like classical classification and clustering algorithms). As shown in Figure 2.2 the data are used to feed the learning algorithm and that outputs the model. Thus the learner has access to a fixed set of samples. The goal of passive learning is to recover the simplest model that is consistent with the data, without over-fitting to the given dataset.

Specifically for State Machine learning, using the passive approach, a fixed set of samples is provided to the algorithm and the goal is to learn the minimal consistent model. Consistent here mean that for every positive sample the automaton should accept and for every negative sample it should reject.

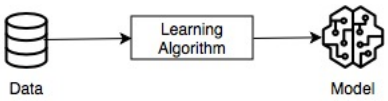


Figure 2.2: Passive Learning Framework

2.2.2 Active Learning

In Active Learning framework there is one more component added to the learning procedure. In Figure 2.3 there is the *Annotation* action which is usually performed by an oracle or an expert. The oracle can answer questions about the learner's current hypothesis and guide the learning procedure accordingly.

In the context of automata learning the most well know algorithm is the Angluin's L-star algorithm [8]. In Angluin's algorithm the oracle is called minimal adequate teacher, because it can learn an automaton in polynomial number of queries with respect to the number of states. In this framework setup the learner can ask the teacher two types of queries. *Membership queries* and *equivalence queries*. In the *membership queries* the learner asks if an input w belongs to the language of the corresponding automaton, and the teacher responds accordingly yes (if $w \in L$) or no (if $w \notin L$).

In an *equivalence query* the learner selects a hypothesis automaton H, and the teacher answers whether or not L is the language of H. If yes, then the algorithm terminates. Otherwise, the teacher gives a counterexample, i.e., a input in which$L$ differs from the language of H.

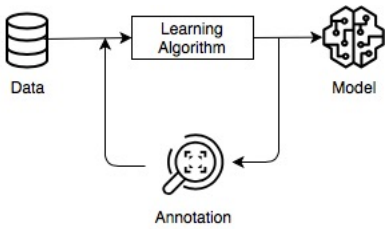


Figure 2.3: Active Learning Framework

2.3 Labelled Transition Systems

There are many different ways to describe systems, like different types of automata, process algebra, Petri nets. However there is one common concept underlying all these formalisms and is called labelled transition systems (LTS).

A labelled transition system is a triple $(S, Act, \rightarrow)$ where:

- S is the set of states,
- Act is the finite set of actions,
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation from state to state.

S is the set of all possible states of the system, Act is the set of names of externally observable actions which can be performed by the system and the transition relation represents the behaviour of the system. The following representation is used to interpret that the system in state s can perform action a and goes to state s': $s \xrightarrow{\alpha} s'$.

The automaton in Figure 2.1 would be written as an LTS in the following way: $S = \{q_1, q_2, q_3, q_4\}$, $Act = \{0,1\}$ and the transition relation contains the following transitions:

$$q_1 \xrightarrow{0} q_2,\ q_1 \xrightarrow{1} q_4,\ q_2 \xrightarrow{0} q_3,\ q_2 \xrightarrow{1} q_4,\ q_3 \xrightarrow{0,1} q_4,\ q_4 \xrightarrow{0,1} q_4$$

2.4 Graphs

Finite State Machines can be represented in terms of graphs and have a corresponding graphical representation. The fact that FSMs provide dense information about the, under learning, system's execution, make their graphical representation very useful.

A graph consists of nodes and edges that connect nodes between them. A directed graph is a graph whose edges have directions. In this thesis, we consider graphs whose edges have both directions and labels. These graphs are called labeled directed graphs. We formalize labeled directed graphs as follows.

Definition A *labeled directed graph* is a tuple $D = (N, E, L)$ where N is a set of nodes, L is a set of labels, and $E \subseteq N \times L \times L$ is a set of labeled edges.

The connection between two nodes in graph theory is called a *path* and the formal definition follows:

Definition Suppose D = (N, E, L) is a labeled directed graph. For all nodes $n, n' \in N$, a path from n to n' is a sequence of edges $\sigma \in E$, where $\sigma = \varnothing$ when $n == n'$ and $\sigma \neq \varnothing$.

2.4.1 DOT language

DOT is a graph description language. DOT graphs are typically files with the filename extension dot [1].

```
// The graph name and the semicolons are optional
graph graphname {
a – b – c;
b – d;
}
```

Figure 2.4: Graph description language for Figure 2.5.

[1] https://en.wikipedia.org/wiki/DOT_(graph_description_language)

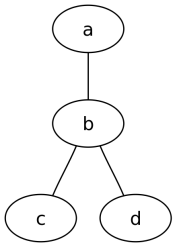


Figure 2.5: Graph representation of description in Figure 2.4

2.5 Model Checking

Model checking is a method for formally verifying finite-state concurrent systems, which is based on the exhaustive exploration of a given state space trying to determine whether a given property is satisfied by the system. Model Checkers will be used for the comparison of the DFAs, thus they are explained in the following section together with their most interesting (for our study) features.

2.5.1 Model Checkers

Another way to verify the equivalence or inequality of two state machines is using a model checker. Even thought there exist a lot of model checkers, just 4 or 5 have a feature that performs equality checking between to graphs. Some of the most famous and well developed have been included in this section. However using a model checker is not as easy as it sounds. The state machine has to be translated in the corresponding language of each model checker. None of the model checkers do seem to accept some well known format for graphs, like json or dot files.

TAPAs

TAPAs[2] [9], is one of the tools, which will be considered, using for the equivalence checking of the DFAs. TAPAs uses process algebras terms to describe the systems, which are then mapped to Labeled Transition Systems (LTSs). TAPAs consists of five components: an editor, a runtime environment, a model checker, an equivalence checkers and a minimiser. The editor allows the users to specify the concurrent systems either in a textual representation or in a graphical notation. The runtime environment generates the LBSs using the given specifications. Model and equivalence checkers are used to analyse the system behaviour. Finally minimiser can be used to reduce the LTSs size, while preserving the intended behaviour.

mCRL2

Another tool that includes equivalence checking is the mCRL2 [3] [10, 11], which is mainly used to specify and analyse the behaviour of distributed systems and protocols. The main parts seem to be the same as in the TAPAs model checker, where the processes are translated into Labelled Transition Systems (LTS), which contain all states that the process can reach, along with the possible transitions between

[2]TAPAs: a Tool for the Analysis of Process Algebras. http://rap.dsi.unifi.it/tapas

[3]mCRL2, a formal specification language with an associated toolset: https://www.mcrl2.org

those states. A powerful tool contained in mCRL2 that is our main interest is the *ltscompare*, which can check whether two LTSs are behaviourally equivalent or similar using various notions of equivalence/similarity (using 15 different algorithms like week/strong bisimilarity or equivalence). Furthermore it provides counterexamples for cases that led to the inequality of the two LTSs.

LTSmin

LTSmin [4] is also a tool for verification and equivalence checking. It stated out as a generic toolset for manipulating Labelled Transition Systems (LTS). Meanwhile the toolset was extended to a a full (LTL/CTL/μ-calculus) model checker, while maintaining its language-independent characteristics. LTSmin connects a sizeable number of existing (verification) tools: muCRL, mCRL2, DiVinE, SPIN (SpinS), UPPAAL and CADP. Moreover it supports the exporting of LTSs into various formats, making it easier to use existing tool.

2.6 Sequence Alignment

Sequence alignment is is widely used for DNA/RNA comparison to observe patterns of conservation or variability or to find common motifs in the two sequences [12]. Sequence alignment involves:

- Construction of the best alignment
- Assessment of the similarity between the two sequences based on the found alignment.

Thus when an alignment is found it has to be assessed based on some measure how good this alignment is. There are different ways to identify the alignment, namely:

- Global alignment
- Local alignment
- Multiple sequence alignment

Global alignment is when two sequences have to be aligned from the beginning till the end. Local is when it is not necessary to align the whole sequence, but small subparts of it. Last but not least, there is the multiple sequence alignment, where more than two sequences need to be aligned. This is the most difficult case of the problem.

In this thesis the global alignment will be used, specifically for two sequences, because is the case that is the closest to what we need to do. Instead of comparing two sequences though, we compare a sequence with a whole tree, but from this tree at the end only one sequence, the most similar, is chosen.

In order to perform the global alignment three actions may be taken. There is the case when the two letter at the current index are the same, which is Case 1 in

[4] https://ltsmin.utwente.nl

Table 2.1. There might be necessary to add a gap either in the one sequence or the other, these are cases 2 and 3 respectively in Table 2.1. The options for the characters comparison are the following:

	Case	**Description**
1	x_i aligns to y_i	Align$x[1...i]$ with $y[1...j]$
2	x_i aligns to a gap	$x[1...(i-1)]$aligned with $y[1...j]$, so $x[i]$aligns with gap in y
3	y_j aligns to a gap	$x[1...i]$ aligned with $y[1...(j-1)]$, so align a gap to x in $y[j]$

Table 2.1: Three choices of global sequence alignment problem

For the aforementioned choices some scores have to be determined to asses the alignment. Therefore there is a score for gap, one when the symbols are matched and an extra, when we have a mismatch. The penalty concept will be extensively explained in Chapter 3.

- Match: The two letters at the current index are the same.
- Mismatch: The two letters at the current index are different.
- Gap: The best alignment involves one letter aligning to a gap in the other string.

Example
Suppose given two possible related strings S_1 and S_2, S_1= ACGTCATCA and S_2 = TAGTGTCA, in Table 2.2 their alignments is presented, where the symbol _ represents a gap at the corresponding sequence.

S_1	_	A	C	G	T	C	_	A	T	C	A
S_2	T	A	_	G	T	_	G	_	T	C	A

Table 2.2: Aligned sequences using Edit Distance

At each step either there is a match or a gap at one of the sequences. In this example the number of changes needed for S_1 to become S_2 were computed, in order to identify the optimal alignment. Suppose that there are two possible alignments, were the first one requires 5 edits and the second one 3, the latter will be chosen as the optimal alignment, because the number of necessary modifications is smaller. This measure is called edit distance. Edit distance is a way of quantifying how dissimilar two strings (e.g., words) are to one another by counting the minimum number of operations required to transform one string into the other [13].

The problem of global sequence alignment is very difficult. Just consider that for two sequences of length n, m the different ways to align them are given by the formula:

$$\binom{n+m}{m} \quad \frac{(m+n)!}{(m!)^2} \approx= \frac{2^{m+n}}{\sqrt{\pi - m}}$$

In the following table 2.3 the number of possible ways to align two sequences of length n are computed.

n	# of ways
10	184,756
20	1.40E+11
100	9.0E+58

Table 2.3: Ways of Alignment for sequences of length n

In Table 2.3 can be seen that the problem cannot be approached greedily like computing every possible alignment for two sequences. Even for small sequences (size 10) there are multiple ways of aligning (almost 185 thousand ways). Thus a different algorithm has to be used to optimize the procedure, one widely used idea is Dynamic Programming, which is explained in the next Subsection.

2.6.1 Dynamic Programming Approach

Dynamic Programming[14], is a method for solving a complex problem by breaking it down into a collection of simpler sub-problems and storing the solution of each sub-problem, in order to solve each of those sub-problems just once. Usually, a dynamic programming approach is used on solving problems where a naive approach takes exponential time, like sequence alignment. With its general method, which is like the "divide and conquer" one, it can solve a problem in time O(n^2) or O(n^3). The difference between the dynamic programming method and the 'divide and conquer is that the sub-problems that are solved will typically overlap, while in the other approach the sub-problems are independent and can be solved totally separately. Another difference is that dynamic programming is a bottom-up approach instead of a top-down one. To give a better understanding of dynamic programming and "divide and conquer' difference, we present an execution graph of these two approaches, where in the Dynamic programming is presented on the left part.

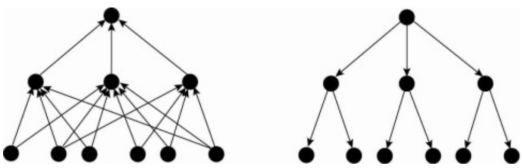


Figure 2.6: On the left the dynamic programming approach, on the right the divide and conquer

As it can be seen from Figure 2.6, the dynamic programming approach starts from the bottom nodes and goes up to every possible direction. Generally there are some main steps that you have to follow in every problem that you want to apply dynamic programming.

1. Characterize structure of optimal solution
2. Define value of optimal solution recursively
3. Compute optimal solution values with bottom-up in a table

4. Construct an optimal solution from computed values

Using dynamic programming we can reduce the problem of best alignment of two sequences to best alignment of all prefixes of the sequences.

Given an n-character sequence x, and an m-character sequence y first a initial table F with dimensions $(n+1)\times(m+1)$ has to be constructed. Each cell $F(i,j)$ of the matrix F, contains the score of similarity till that point, thus the alignment score between $x[1...i]$ with $y[1...j]$. Using this matrix the scores do not have to be recomputed at each step, but the parts of the already filled matrix are used. Since we know the best score for the alignment $x[1...i-1]$ with $y[1...j-1]$, we compute for the new incoming terms of the sequence $x[i], y[j]$ and the new cost is filled in the corresponding cell $F(i,j)$.

Finally using the complete matrix, traversing each column from the last one to the first (bottom up), the optimal alignment can be computed based on the maximum values of each column. This method will be modified and implemented in Chapter 6. More details and examples on how it works can be found in that Chapter.

2.7 Metrics

The problem we are dealing with, is not the classical machine learning problem, however we have to define the corresponding positive and negative classes. Here the most frequently used metrics for machine learning problems are presented, although in Chapter 7 we show how they will be adjusted for our needs. Suppose there are two classes one positive and one negative, the instances that are correctly classified as positive are called *True Positives*, the ones that are wrongly classified as positive (their real label shows they are negative) are called *False Positives*. The instances, which are correctly classified as negative are called *True Negatives* and the wrongly classified as negative *False Negatives*. With respect to the defined terms above, we have the following formulas:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \tag{2.1}$$

$$Recall = \frac{TP}{TP+FN} \tag{2.2}$$

$$Precision = \frac{TP}{TP+FP} \tag{2.3}$$

$$F1\ score = 2\times\frac{Recall\times Precision}{Recall+Precision} \tag{2.4}$$

3

Related Work

Software development relies on the use of models that can specify how the system is supposed to behave or how it really does. Software can be complex, thus it is usually difficult for developers to identify issues or differences between software. The log files contain information, which could be elaborated for the identification of the differences or issues, however it is not an easy task. In this Chapter the concepts of log analysis and log differencing are presented to solve the aforementioned problems.

Likewise, some of the literature regarding model checkers and conformance checking is presented, since many researchers have applied these techniques in the same domain as the one we are investigating.

Additionally, simple sequence alignment is explained. This might look irrelevant for the problem we are trying to tackle, but later in this thesis it will become clear how this method can be adjusted for log comparison purposes.

Finally, in the last Section 3.7, we tried to gather all the state machine comparison methods, to have a broad understanding of (almost) all the methods that can be applied for the particular question, before developing our own approach.

3.1 Related Work at Adyen

Adyen already conducts a lot of research that focuses on log analysis. Peter Evers [15] has developed a tool called Logness during his master thesis, with a web interface, which utilises Longest Common Subsequence algorithm to cluster logs with a severity level of WARN and ERROR together. Logness sends a notification to developers if a new log cluster is created, which can be used to monitor release as well. On the same tool worked Quincy Bakker [16], who tried to improve it by employing various techniques like requirements analysis, interviews, data visualisation and user analysis.

Rick Wieman [6] evaluates different passive learning methods and uses them to provide intuition to developers about the behavior of the software running on the POS terminals. Joop Aue [17] researches WEB API usage in Adyen and recommends ways of avoiding API faults. Daan Schipper [18] has worked on the evaluation of tracking back algorithms of log data to its origin using static analysis.

3.2 Log Analysis

In [4], Fu et al. have developed a technique for automatic anomaly detection in log files generated by distributed systems (the type of anomaly they focus on is low performances). Their technique consists of two processes: the *learning process* and the *detection process*. The *learning process* concerns the learning of a model using normal data. In order to achieve that they first convert the log messages from the training log files to log keys. Using these keys a FSA is used to model the execution path of the system. Also the execution time for each state transition has to be computed. In the *detection process* the new incoming logs can be checked for anomalies using the aforementioned learned model. The type of anomaly they focus is low performance. In more detail they split the low performance in two types, the one is modelling the execution time of the state transitions and the second the modelling of the circulation number of loops.

Similar research, but differently approached has also been performed by Mariani and Pastore [5]. They also present a technique for automatic analysis of log files and retrieval of required information to identify failure. This research focuses on another kind of anomaly, which is the failure. Their approach consists of three phases. Phase 1 is *monitoring*, phase 2 is *model generation* and phase 3 is *failure analysis*.

In the *monitoring* phase, the collecting of the log files takes place, these log files can either be generated during testing or from actual usage of the system. Nonetheless they should be successful executions, because they will be used for the learning process. The *model generation* phase, consists of three major tasks: event detection, data transformation and model inference. The event detection task, is responsible for rewriting the events of the initial log file in order to make it structured. The data trasformation task, removed the concrete attribute values contained in the log with more meaningful data flow information. The final task for this phase is the *model inference* which creates the model using the kBehavior inference engine. During the last phase, the *failure analysis*, the anomalous patterns are identified using the model learned from previous phase. The disadvantage of this methods is that a developer is needed to annotate the potential anomalous patterns as anomalous or normal.

An interesting approach for anomaly detection in DFAs using formal analysis is proposed by [19]. The goal of their work is to verify the trustworthiness of an implementation based on the specifications. For that purpose they propose a formal approach to identify vulnerabilities in an DFA using symbolic algebra. Thus they model the specification of a given DFA as a set of polynomials, such that each polynomial is responsible for describing all of the valid states and their corresponding transitions by which a specific state can be reached. The next step is to also model the implementation of the DFA using polynomials. Since both the specification and the implementation are models, the next step is to check if these two sets are equivalent. The Gröbner basis theory was used to compare the two sets, by reducing the implementation set and using the remainder the vulnerabilities could be found. In more detail zero remainder means that the implementation has followed the specifications and can be trusted. On the other hand, non-zero remainder means that there are hidden malfunctions in the implementation and also the conditions

that activate them are identified. Their research tackles the following three problems. First the problem that the formulation of the specifications of a general circuit cannot be modelled as one simple and comprehensive polynomial (that's why they use sets of polynomials, one polynomial for each state). Secondly there is the issue of cycles in the sequential circuits, where these loops can make the reduction procedure infinite. Last but not least, there is the problem that an anomaly may appear after a long repitition of normal states (after a large number of cycles).

Log analysis can be used for many different purposes, like performance bottlenecks or bugs identification. In that field state machines are widely used because of their ability to model accurately the behaviour of the system. Interesting research has been done by [20], where they make use of state machines to model the behaviour of the system in order to identify performance issues. For that purpose three different algorithms have been used, kTails [21], Synoptic [22] and Perfume [20]. The best results were obtained when using Perfume, which is an extension of Synoptic in a principled manner to account for performance information often available in system logs. Perfumes goal is to produce a representative model of a system form an execution log containing examples of system's behaviour. In order to achieve that, the state-of-the-art model inference techniques are improved to utilise performance information in the log to guide the inference process and predict unobserved executions more accurately. The process Perfume follows to infer models consists of the parsing stage, performance property miner, initial model construction, CEGAR refinement (counterexample guided abstraction refinement) and kTails coarsening, after this steps the final model is constructed. The final model, is claimed by the authors, to be easier understandable by the developers. Thus these models can be used for performance testing.

3.3 Log Differencing

Relevant research for log anaysis specifically targeting log differencing is performed by Goldstein et al. [23]. They model the behavior using the log files and then they perform diff calculation and highlight the outliers. In more detail the diff calculation part is separated in multiple steps. First they extract all the possible paths for both finite state models. Then they convert these paths into string and using a string matching algorithm [24] they find the common paths. Next they define nodes to be common to both models if they have the same label and if they are a part of at least one common path. That however means that if a state appears in a trace and unexpectedly appears in another as well, it will not be identified as added for the latter trace. The states that are not common are annotated as added or removed based on which one of the initial models they were found. Finally they present the difference model, which is actually the second model with the added and removed nodes annotated. The basic disadvantage of this approach is the creation of every possible path from the models. This may lead to unbearable run times. For that reason they have introduced a parameter, which terminates the algorithm when a predefined amount of paths is reached. As expected this mean that not all possible paths are found using this approach. Additionally the idea of annotating a node as common only if it appears in a common path is not helpful in most cases because

the same node may appear in multiple paths, in some of them it may actually be common but in other it might be an added or removed node.

Anothe log differencing tool is developed by Amar et. al [25]. Their goal is to identify the different traces between two or more log files and then model them using Synoptic. Their approach is different that what most researchers on this field usually do, where first they model each log file separately and then compare the models to identify differences. What they do in their *2KDiff* algorithm is compute the sets of k-sequences included in each of the logs, and compare the two sets to find the k-sequences that are unique to each log, which will be the k-differences. The k-differences are presented using some traces that contain that differences.

For example suppose we have two log files and their corresponding models. Any trace highlighted by the *2KDiff* algorithm over the first model is a trace of the first log file that contains at least one k-sequence missing from the second log file.

The drawback of this method is that the whole trace is annotated if only a small thing is different between the two log files. If a single state is added and the rest of the trace is the same they will find k-sequences that contain events that do not appear in both logs. Thus the whole trace will be highlighted, which means it is not possible to identify the root cause of the difference. A big advantage of their work is the extension of the *2KDiff* algorithm, the *nKDiff* algorithm which can compare n log files between them and highlight the differences in one finite state machine learned by the traces that contain at least one of the k-sequences.

3.4 Model Checking

Model checking is a method for formally verifying finite-state concurrent systems, which is based on the exhaustive exploration of a given state space trying to determine whether a given property is satisfied by the system. For that purpose efficient symbolic algorithms are used, thus extremely large state-spaces can often be traversed in minutes. A model checker takes a model and a property as inputs and outputs either a claim that the property is true or a counterexample falsifying the property. Model checkers are most usually used in hardware and protocol verification so far, with many successful results like Futurebus+ and the PCI local bus protocols [26], [27], [28], [29]. However a bigger challenge is to verify software systems because of the complexity the paths may have.

There are two general approaches as far as the model checking is concerned, *Temporal Logic Model Checking* and *Behaviour Conformance Checking*, categorized by whether the specification is a formula or an automaton.

In *Temporal Logic Model Checking* a property is expressed as a formula in a certain temporal logic. The verification can be accomplished using an efficient breadth first search procedure, which views the transition system as a model for the logic and determines if the specifications are satisfied by that model.

On the other hand, *Behaviour Conformance Checking* refer to the procedure of comparing if two models have the same behaviour. This procedure is widely applied in the business field, where it is considered a family of process mining techniques to compare a process model with an event log of the same process. It is used to check

if the actual execution of a business process, as recorded in the event log, conforms to the model and vice versa.

An extension of model checking is the equivalence checking, where two models have to be checked if they present the same behaviour. For that purpose different logic formulas may be checked to identify if the two models return the same result.

Equivalence checking is mainly used for electronic design automation, commonly used during the development of digital integrated circuits to formally prove that two representations of a circuit design exhibit exactly the same behaviour.

This method has also been used in software, for translation validation [30], [31], [32]. There is also some research done in comparison between original and rewritten programs [33].

3.4.1 History of Model Checking

The first research about model checker started in 1981 by Clarke, Emerson et al. [34], who developed a verification technology for abstract models, like a hardware or software design. By algorithmic means they determined whether a model satisfies a formal specification expressed as a temporal logic formula and if not a counterexample was returned. Then in 1982 another paper was published again with Clarke and Emerson as main authors [35]. In the 90's the states of the model that could be checked increased to more than 10^{20} [36] and also the SMV symbolic model verifier was developed [37]. End of the 90's Biere et al. approached the model checking using satisfiability procedures [38]. Finally in 2000 Clarke et al. used the counterexamples produced by the model checker in order to refine the produced model.

3.5 Conformance Checking

Verification and validation of computer simulation models is conducted during the development of a simulation model with the ultimate goal of producing an accurate and credible model [39]. Verification is thus used to establish the correctness of the systems or the protocols.

Simulation models in our case are considered the state machines that model the behavior of the transactions. Learned models approximate imitations of the real-world systems, thus it has to be verified that they actually work as expected. In this section, a different technique to compare two models is presented, using the verification techniques.

Conformance Checking is mainly used for checking how the actual behavior of the system, as recorded in logs, conforms the expected behavior as specified by the model. If we expect the model to represent the real behavior of the system, the differences that show up during the conformance checking might be bugs or new features added to the system which are not represented by the model. In each case, these differences are important information for the behavior of the system.

The conformance checking is a standard procedure, which however depends on a lot on the application domain. The difficult part is to quantify the degree of conformance of the logs for the given logs. In [40] they developed a robust replay technique to measure the conformance of the logs for the input model. They try

to provide intuitive feedback for the logs that do not conform with the model by specifying the activities that were skipped or inserted.

Skipped activities refer to activities that should be performed according to the model, but do not occur in the logs.

On the other hand, the *inserted activities exist in the logs, but should not happen according to the model.*

The problem of measuring the conformance of the logs, is not solved by that though, because different activities might have different severity when they are skipped or inserted. Therefore methods that weight equally all inserted and skipped activities do not provide correct results like [41]. The improvement that [40] suggests is the use of a cost based measure to measure the fitness of the model. The idea behind this metric is that fitness value should decrease with the increase of the amount of skipped and inserted activities. In order to normalize the metric they consider the extreme case, where every activity is inserted. However a big disadvantage of that method is that the user should provide the cost functions for the skipped and inserted activities as a parameter.

3.6 Sequence Alignment

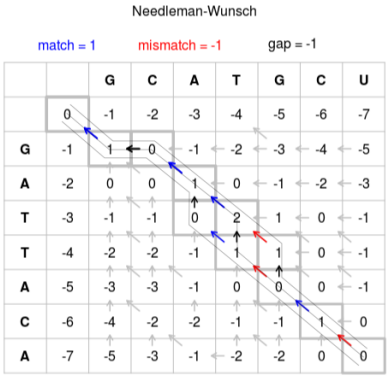
As mentioned in Chapter 2, sequence alignment can be performed in three different ways, *global*, *local* and *multiple* sequence alignment. In this section the global alignments will be covered, since this is the only type that can be applied for the problem we are facing.

The first and most widely known method for global sequence alignment is the Needleman-Wunsch algorithm [42]. They make use of the dynamic programming to improve the otherwise high complexity of finding the optimal alignment by computing every possible alignment. Needleman and Wunsch create a 2-dimensional table with dimensions the lengths of the two sequences under comparison. For a sequence x with length n and a sequence y with length m the matrix will be $(n+1) \times (m+1)$. The $+1$ at each dimension is because of the gap that can be added in the sequences. The matrix is filled with the best alignment for $x[1...i]$ with $y[1...j]$ in the corresponding cell $F(i,j)$. However at each step all the previous values do not have to be re-computed, since they are already saved in the matrix. The only necessary thing is to chose the max value from $F(i-1,j), F(i,j-1)$ and $F(i-1,j-1)$ and add the penalty for gap, match or mismatch. The scores they usually use are: match=1, mismatch=-1 and gap=-1.

For comparing GCATGCU and GATTACA the filled table can be found in Figure 3.1 [1].

The arrows, starting from the bottom, going up, indicate the optimal alignment based on the highest score. The blue arrows indicate we have a match in the specific character between the two sequences, when the arrow is red it means there is a mismatch between the two characters of the specific column and row and the black

[1] `https://en.wikipedia.org/wiki/Needleman\T1\textendashWunsch_algorithm`



		G	C	A	T	G	C	U
	0	-1	-2	-3	-4	-5	-6	-7
G	-1	1	0	-1	-2	-3	-4	-5
A	-2	0	0	1	0	-1	-2	-3
T	-3	-1	-1	0	2	1	0	-1
T	-4	-2	-2	-1	1	1	0	-1
A	-5	-3	-3	-1	0	0	0	-1
C	-6	-4	-2	-2	-1	-1	1	0
A	-7	-5	-3	-1	-2	-2	0	0

Figure 3.1: Needleman-Wunsch application example by Wikipedia

Sequences	Best alignments
GCATGCU	GCATG-CU GCA-TGCU GCAT-GCU
GATTACA	G-ATTACA G-ATTACA G-ATTACA

Figure 3.2: Results Needleman-Wunsch algorithm

arrow, indicates that a gap should be added in order to achieve the optimal alignment. In this example there are three best alignments, as shown in Table 3.2.

The Needleman-Wunsch algorithm is used even today, however attention should be paid in the scoring scheme. Using different values for match, mismatch and gap penalties can highly influence the outcome of the algorithm. This topic will be discussed in the next subsection.

3.6.1 Penalties

Sequence alignment algorithms usually are highly influenced by the setting of the penalties. The effect that such parameters have on the resulting alignment are not well understood yet. In this type of programs the correct setting of the parameters does not influence the speed of the run, but the produced result. Namely, for wrong parameter values a non-optimal alignment will probably be returned. The simplest scoring scheme is proposed by the prototypic global alignment algorithm of Needleman & Wunsch in [42]. They propose to reward each state match with one point and penalize the mismatched and the gaps with one point. Also in the most famous local alignment algoritm by Smith & Waterman [43] they also use a linear function as the gap penalty function. In addition to the penalties, many sequence alignment algorithms, use similarity matrices, which however will not be covered in this section because they are not in the scope of this research (they are mainly used in the bioinformatics field, where the similarity of the proteins is already defined).

The question about the impact of the scoring scheme has attracted many re-

searchers in the field of biology. Fitch & Smith [44] tried to address this problem by calculating the number of alignments until the optimal alignments are found. Then they divide the parameter space into regions based on the optimal alignments of each region.

Specifically for biological sequences a widely used method is by inverse parametric sequence alignment [45, 46]. This approach proposes setting the parameters using examples of biologically correct reference alignments. However the scope of this thesis is not about biological sequences, thus this method cannot be used.

Also the dynamically changing penalties based on some attribute have been investigated. In [47] Thompson et al. use a position-specific penalty scheme, where positions in early alignments where gaps have been opened receive locally reduced gap penalties to encourage the opening up of new gaps at these positions.

Unfortunately the tuning of the these parameters for sequence alignment is mainly focused on biological sequences, which is not the type of data used for this work. Thus all these methods are mostly for inspiration and cannot be used, without modifications, for the purposes of the current research.

3.7 State Machine Comparison

State Machines can be compared from two perspectives [48]: (1) in terms of language - the externally observable sequences of events that are permitted or not (2) in terms of their structure - the actual states and transitions that define the behaviour.

In [49] Mealy machine learning methods are used to verify the correctness of protocol implementations relative to a given reference implementation. Reference implementation defines the specification of that protocol and can be considered as the standard. Their approach first uses a states machine synthesis tool to actively learn a state machine model of the reference implementation. Given another implementation two things can be done. The first one uses a model based testing tool to generate test sequences to test whether the reference implementation and the new one have the same behavior. The other approach is to use the state machine learning tool to learn a model for the new implementation and then use an equivalence checker to check if the two models have the same behavior. The equivalence checker returns true if the two models are the same and false if they are not. When they are not equal also a counterexample is returned which is checked in the implementations if it truly is a difference. If not something did not go well in the learning procedure and one of the models has to be refined.

In their work they investigate the feasibility of the above approach for the bounded retransmission protocol, which is a variation of the classical alternating bit protocol. They implemented this protocol and referred to it as reference implementation and six other faulty variations. The goal is to identify the behavioral differences between the faulty implementations and the reference one.

3.7.1 Language Comparison

One of the most common approaches when comparing to LTS languages is to take random sample from the language of the machine that better characterises the

language as a whole, and count how many of these random sequences are correctly classified using a second state machine [50]. Another similar approach is based on random walks. A random walk is a mathematical object, known as a stochastic or random process, that describes a path that consists of a succession of random steps on some mathematical space. As far as the comparison of two DFAs is concerned random walks over one DFA will be followed and the output will be compared with the output one of the other DFA. A walk is an arbitrary input sequence that either concludes in an accepting or an rejecting state. If for enough sequences both DFAs return the same output, the two DFAs can be considered equivalent. An alternative approach is to produce random strings form the alphabet alone and then classify these as belonging to the language or not with respect to the reference LTS. In the data, we are dealing with, we expect to see specific order in some sequences, since we have to do with transactions and there is no chance that the state Payment_approved will come before enter_pin for example. That mean that only a small fraction of the random sequences will represent the behaviour of the input machines. Consequently all their differences cannot be found by that method. The fact that the aforementioned methods of this category rely on a sample subset of the whole space, makes the validity of the comparison questionable. When we are dealing with random walks, the algorithm is biased towards specific parts of the state machine.

Walkinshaw et al. [51], show how techniques from the domain of model-based testing, instead of random sampling, can be applied to compute a representative sample of the language. In the model-based testing, the assumption is that we have a model and we try to test a system which is a black box for us (implementation). Only observations about the output with specific input can be determined. The goal is to verify that the two models behave the same and the implemented system leads to the correct states.

There are also other methods that try to define a test set of important sequences in order to prove if the machines are equivalent. Peled et al. [52] was one of the first who proposed to implement equivalence queries via conformance testing. In more detail they suggested to use the conformance testing algorithm by Valilevskii [53] and Chow [54]. This method is also knows as W-method. What W-method does is given some implementation (black box system) and some specification LTS, constructs a set of sequences that should be classified equally by both LTSs.

The problem with the aforementioned methods is that they require a big size of test suite, exponential to their upper bound of states, which may be a bottleneck for any real application. During the Zulu competition, where the goal was to decrease the equivalence queries in the learning procedure using an active learning framework, the competitors learned finite automata from a limited number of membership queries without explicit equivalence queries. Thus they had to approximate the equivalence queries through clever selection of membership queries [55]. Even though in this thesis we are dealing with passive learning, the part of the equivalence of the models can be used.

3.7.2 Structural Differences

There is a limited amount of work that compares the states machines in terms of their structures. In more detail Walkinshaw and Bogdanov have worded extensively in the subject [48], [56], [57], but also other researchers [58], [59], where they also dealt with the event name abstraction issue using natural language processing techniques [60].

Walkinshaw and Bogdanov have developed the LTSdiff algorithm [2]. This algorithm computes the difference between two LTSs returning the missing and added states and transitions. It is inspired by the cognitive process of humans. The first step for the algorithm is to identify *landmarks*, pairs of states that seem to be equivalent. Potential landmarks are identified by measuring a similarity score for every possible pair of states. This score is computed by matching up the surrounding network of states and transitions. There are two different similarity measures established, local similarity is the overlap of the immediate surrounding transitions, on the other hand global similarity measures the similarity of the target states of these transitions.

Local similarity S_{ab} of two states a and b is computed by dividing the number of overlapping adjacent transitions by the total number of adjacent transitions.
$S_{ab} = \frac{|matchingAdjacentTransitions|}{|AllAdjacentTransitions|}$

State machines characterize a state both in terms of its potential past behaviour (incoming transitions) as well as its potential future behaviour (outgoing transitions), S^L_{Prev} and S^L_{Succ} respectively. The global similarity concerns the similarity of pairs of states in terms of their wider context. For two matched transitions, we want to produce a higher score if the source/target states of these transitions are almost equivalent and a lower score if they are dissimilar.

When the scores are computed, one pair of states is chosen as landmark or equivalent and it is used as the basis for further comparison in the remaining of the states and transitions in the machines.

Wieman's Comparison

Wieman [61], [6] implemented two different algorithms. The one is able to identify structural differences in the graphs and the other frequency differences. Both his algorithms perform two different steps. During the first step the differences between the *reference* and *observed* graph are identified and during the second step these differences are visualized.

Structural Differences

There are essentially two types of structural differences: introduced paths and missing paths. Introduced are paths that do not exist in the reference graph, but that are introduced in the observed graph. This potentially indicates newly introduced functionality, or bugs in the system. Missing are called the paths that existed in the reference graph, but were not present in the observed one. This potentially indicates removed functionality, or resolved bugs.

[2] LTSdiff implemented in StateChum http://statechum.sourceforge.net

3.7.3 Equivalence Checking

Equivalence checking is used to make sure two state machines are equivalent. They may be different structurally with the first look but the languages they are accepting are the same. The most frequent use of equivalence checking is for state machine minimization. However in this project we want to use it in a different context. The idea is to check if two state machines are equivalent and if they are not, perform the necessary modifications in order to become.

Equivalence Checking in the active learning framework

The most usual use case of equivalence checking is in the active learning field, where there is a teacher and a learner. When the learning algorithm converges to a stable hypothesis, a counterexample is needed for further progress. These counterexamples are the results of equivalence queries that test the equality between the hypothesised and the actual model.

Since in the field of passive learning equivalence checking is not frequently used, the methods used in active learning will be researched and their adaptation in our field of interest will be questioned or performed if possible.

Using the product of two dfas

One of the simplest methods to check if two finite state machines are equivalent is to compute their product machine. Even for the computation of the product many algorithms have been developed [62] and many researchers have tried to improve the existing algorithms in order to make them more efficient to be applied to real problems [63] [64]. Despite the wide use of state machines' product there exists no software available for that purpose.

When applying the product algorithm, there are two conditions that should be satisfied for the two automata to be equivalent.

- The two automata are **not** equivalent if got a pair q_a, q_b one is an intermediate state and the other is a final state.
- If initial state is also final state for an automaton, then in the second automaton the same should apply for them to be equivalent.

The following example will guide us through the procedure of checking if two automata are equivalent using their product.

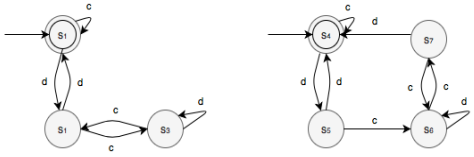


Figure 3.3: Finite State Machines under comparison

It is obvious from Figure 3.3 that the second condition is satisfied, for both automata the Initial state is also final state. In order to check if the first condition

states	c	d
(q_1, q_4)	(q_1, q_4)	(q_2, q_5)
(q_2, q_5)	(q_3, q_6)	(q_1, q_4)
(q_3, q_6)	(q_2, q_7)	(q_3, q_6)
(q_2, q_7)	(q_3, q_6)	(q_1, q_4)

Table 3.1: Pair of states for every possible action

holds, for specific pairs of the state machines' states both states should be intermediate or final. The procedure follows:

In 3.1 the first pair of states consists of the initial state of each automaton (first column, first row). Then considering all the possible actions taken from these states, from q_1 with action c we stay in state q_1, form state q_4 with state c we stay in action q_4. From state q_1 with action d we go to state q_2 and from q_4 we go to q_5. Now it has to be checked if both states in each pair have the same type (either both intermediate or both final). (q_1, q_4) are both final, (q_2, q_5) are both intermediate. Then we check from (q_2, q_5) were we go with all the possible actions. In the second row of Table 3.1 the outcome is visible, again both (q_3, q_6) are intermediate and (q_1, q_4) final. Since till now the condition 1 is satisfied we continue with the next pair (q_3, q_6). The same procedure is continued till we cover all the possible actions from every state. In the case indicated in 3.1 all the pairs contain states of the same type thus we can conclude that the two Finite State Machines are equivalent.

Symmetric Difference with Emptiness Testing

In [65] Sipser, presents the problem of determining whether two DFAs accept the same language or not as decidable. EQ_{DFA} is decidable, when $EQ_{DFA} = \{(A, B) | A$ and B are DFAs and $L(A) = L(B)\}$. The proof of this theorem can be used as algorithm to test the equivalence between two DFAs. The idea behind it is to find out if there are any state sequences, which were accepted by A, but not by B and vice versa. If the result in both cases is the empty set, then A and B should be equivalent, otherwise they are not.

The idea of this algorithm has been used for model-based testing, interesting approach to that was presented by Tappler et al. in [66]. Where they present their learning-based approach to detect failures in reactive systems. Their technique uses inferred models of multiple implementation of a common specification, which are pair-wise cross checked for equivalence. In order to check the equivalence of the two models they check all inputs and they expect that both state machines will have the same output. When an input sequence is not satisfied it is considered a counterexample, this method is based on Sipser's *Symmetric Difference with Emptiness Testing*. As mentioned above, the two state machines are equivalent if there is no input that is not satisfied by both. If a sequence is not satisfied by one of the state machines, is called counterexample and it is marked as suspicious and then is checked manually by a developer.

In more detail their approach consists of three stages. In the first phase they learn their models of several different implementations. Specifically they work with

implementations of standardized protocols or operations. During the second phase the models are pair-wise cross checked. Finally the counterexamples are analysed manually. However the drawbacks of this method is that it computationally expensive because all models have to be pair-wise checked to decrease the danger that a bug that appears in more than one implementations will stay hidden. There is always the possibility though, that a fault will be implemented by all examined implementations. Also the fault-detection capabilities are limited because of the level of abstraction they introduce during the learning procedure.

3.7.4 The Table-Filling Algorithm

Hopcroft et al. describe in their book [67] the table filling algorithm, which can be used both for DFA minimization and equivalence checking between DFAs. This algorithms treats the two DFAs as one and looks for states that are distinguishable for some input. The found distinguishable states can lead to more distinguishable states, for example if some input transitions a pair of states in a distinguishable pair of states, then also these states are considered distinguishable. Once the algorithm is completed, if the start states of the two DFAs are still not distinguishable, then the two DFAs are considered equivalent, since they accept the same language.

Many versions of this algorithm have been developed which improve it's time complexity, but also extended in order to return example sequences of input for which the automata do not return the same output. The table-filling procedure takes $O(n^4)$ runtime (every state has to be checked with every other if they are distinguishable), which makes impossible to be used in real life applications where even 10^{20} states may be encountered.

The *nlogn* Hopcroft Algorithm

One of the most famous algorithms is introduced by Hopcroft already in 1970 [67]. This algorithm was initially developed for DFA minimization, but it can also be used for equivalence checking. The runtime complexity is $O(nlogn)$, which make it much for efficient than the previously presented algorithms. However this research has the drawback that it is not well explained and justified. Thus after Hopcroft other authors [68], [69] came and re-introduced the algorithm helping in making it more theoretically correct and with better running time analysis.

Gries [68] provides a cleared explanation of the algorithms in a more understandable way, but also criticised its correctness proof and its runtime analysis, presented by Hopcroft in [67].

Comparison using Model Checkers

Interesting research on how to compare two state machines using an equivalence checker is done by [33]. In this approach the goal was to refactor software and it was implemented for some Phillips software. For that purpose a model should be learned for the legacy software and one for the new implementation. Then these two models where compared to check if there are any differences. If there are, the checker returns a counterexample. Then the models are improved using this example and the procedure is repeated. Then the models should be improved if possible, in

order to contain the aforementioned example. If it is not a problem of the learning procedure, the implementation of the new software has to be adapted to look more alike to the legacy one.

Many of the previous works have focused on translation validation using equivalence checking. In more detail in the translation validation, given two programs a software has to verify that they have the same semantics. It can be used to ensure that program transformations do not introduce semantic discrepancies and it can also be used to improve testing and debugging. A lot of real-world compilers have been examined to verify the successful translation, like different versions of GCC [30], [31], [32]. From the aforementioned methods the most promising seem to be [32], with an exceptionally low rate of false alarms. However the authors admit that their approach does not scale beyond a few hundred of instructions.

Model Checkers can also be used for various domains, like for bank supply process. In [70] the applicability of equivalence checking to validate business process is explored. The authors claim that due to state explosion problem, the formal methods are not very popular in the business domain and they try to combat that problem using an efficient procedure based on heuristic search proposed in [71]. This approach suggests to expand first the states that offer the most promising way to deduce that two systems are not equivalent. In their research they compare Grease (GREedy Algorithm for System Equivalence) [71] with CADP [3], a well known model checker. In CADP they used the bisimulation property. For the grease tool, the used heuristic function assigned a value to each node n of the graph, which was the degree of dissimilarity of the two processes in n. This functions suggests that each state should contain two not bisimilar processes so that this state can be included in the graph as soon as possible.

[3] https://cadp.inria.fr

4

Data Exploration

Data – a collection of facts (numbers, words, measurements, observations, etc) that has been translated into a form that computers can process.

Before start working with data, it is important to clearly understand what do these data represent. In some cases, it is not enough to just go through the data and understand their purpose. Because of the complex nature of the data, extensive visualizations and analysis are necessary to get the most out of the data. As mentioned above, data are formatted in a way, that computers understand. In order to make the best out of it, first they have to be transformed into a human understandable format.

For this thesis, POS transaction log data have been used. These data consist of sequences that represent the flow of a transaction. As expected, some sequences will appear very often, or specific sub-parts of the sequence (like the initialization of the transaction) while others will appear rarely.

4.1 Data Description

For this project we worked with log files provided by Adyen. These log files contained the flow of transactions taken place either in the testing environment or in the real world. The log files we are dealing with, contain an ordered sequence of events that always lead to a predefined set of final states. This set has size four and includes the state that indicates the transactions has been successfully completed (`Approved`), that the transaction was not processed for some reason (`Declined`) or the transaction was canceled either by the customer or by the merchant (`Canceled`). Last but not least, there is the `Error` final state, when something unexpected happens. The Error final state is the most unwanted case, because it points out that a sequence of actions not covered by the developers was followed.

The alphabet of the sequences in our dataset is frequently changing, since a new software release may produce an additional state in the flow. A transaction flow contains the actions performed by the customer, like `card_swiped` or `pin_digit_entry`

and the necessary communications between the back-office and the device in order to process and authorize the payment, like `verify_card_holder_succeeded`, `terminal_risk_management_succeeded`.

In Figure 4.1, an example of the transaction log can be found, and in Figure 4.2 two example sequences after the pre-processing phase are shown.

```
20170101160001  Adyen version : ******
20170101160002  Starting TX/tender_reference =******/amt=10001/currency=978
20170101160003  Starting EMV
20170101160004  EMV started
20170101160005  Magswipe opened
20170101160006  CTLS started
20170101160007  Transaction initialized
20170101160008  Run TX as EMV transaction/tender_reference=******
20170101160009  Application selected app:****** pref :******
20170101160010  read_application_data succeeded
20170101160011  data_authentication succeeded
20170101160012  validate 0
20170101160013  DCC rejected
20170101160014  terminal_risk_management succeeded
20170101160015  verify_card_holder succeeded
20170101160016  generate_first_ac succeeded
20170101160017  Authorizing online
20170101160018  Data returned by the host succeeded
20170101160019  Transaction authorized by card
20170101160020  Approved receipt printed
20170101160021  auth_code:**** psp_ref:**** pos_result_code:
                APPROVED refusal_reason:None
20170101160022  Final status : Approved
```

Figure 4.1: Sample Transaction log file

S1: Transaction_init/*** Running_swipe_transaction/*** CVM:_Signature/*** Authorizing_online/*** Request_over_WiFi_to_PAL_URL:*** Transaction_Authorized_online/xxx Using_service_code:***/*** Authorization_succeeded/*** Approved_receipt_printed/*** amount_original_/*** ARC_11/*** RC_APPROVED/*** Final_status:_Approved/***

S2: Transaction_initialized/*** Contactless,_bin_number/*** Running_contactless_transaction/*** CTLS_online_processing_ARQC/*** CVM:_No_CVM_required/*** Transaction_Authorized_online/*** Approved_receipt_printed/*** amount_original_/*** ARC_00/***

RC_APPROVED/*** Final_status:_Approved/***

Figure 4.2: Discrete sequences example

4.2 Data Analysis

Adyen is dealing with payments, thus the security in this sector is extremely important. Adyen has a separate department for the testing of the POS terminal's software. The terminal's software as well as the required POS libraries for the communication are tested using robots.

Figure 4.3: Robot for testing the POS terminals

4.2.1 Data from the Testing Environment

The testing that takes place in Adyen aims to the higher coverage of the possible scenarios that may happen using a POS terminal. Thus, as expected, the unhappy flows are the ones that should be tested the most extensively, since these are the flows that can harm the merchant the most. The distribution of the data is different for the test and live data as will also be showed in this section.

A whole robot run consists of more than 700 scenarios that try to cover the most common flows and more extreme scenarios to make sure that the software will perform as expected in the field. Figure 4.4 shows the distribution of the types of transactions that are performed at a robot run. As expected there are a lot of `Approved` transactions, since the happy flows, which are also the most frequent followed flows, have to be tested. However the `Declined` are even more. That happens because the aim of the testing is to make sure that even if the user makes the most unexpected actions or generally the most extreme executions take place, the software will behave according to its specifications.

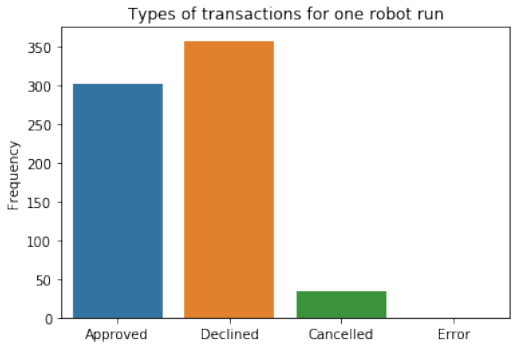


Figure 4.4: Distribution of Transaction's Final Status in Testing

4

In the payments industry the goal is to end up in the approved state flawlessly. The most *dangerous* state is the `Error` because something unexpected that the developers did not think about happened. The number of states a transaction goes through varies a lot. In more detail there can be from 10 or even less to more than 100 states. In Figure 4.5 the distribution of the number of states that the transaction go through for test data of one week can be found.

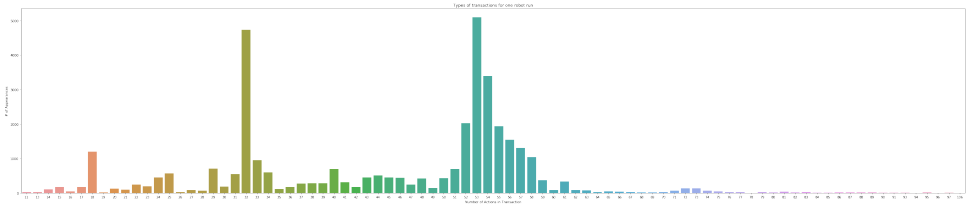


Figure 4.5: Distribution of the number of the Transaction States in Testing

For these specific data, the number of actions in each transaction can vary from 11 to 106. In Figure 4.5 we see that the most frequent number of states for the testing environment ranges from 29-59. Transactions with more than 59 events or less than 29, seem to be rare and probably test the most crazy scenarios of the software.

4.2.2 Data from the Live Environment

As it can be seen in Figure 4.6, the number of `Approved` transactions are by far more than all the other types, in contrast to what happens in testing (Figure 4.4), where `Declined` transactions are the most frequent. It's interesting to notice that there are about 70 erroneous transactions out of the 41840 in the dataset, which is 0.17% of the times.

In Figure 4.7 the correlation between the number of states at each transaction and the final state is investigated. In that plot the only interesting outcome, seem to be the fact that for less than 30 symbols in the sequence, the final status is never

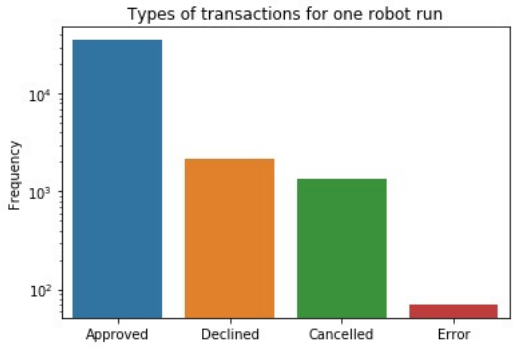


Figure 4.6: Distribution of Transaction's Final Status in Live Environment

`Approved`, for the specific data. For sequence size between 46 - 54 all the transactions are `Approved`.

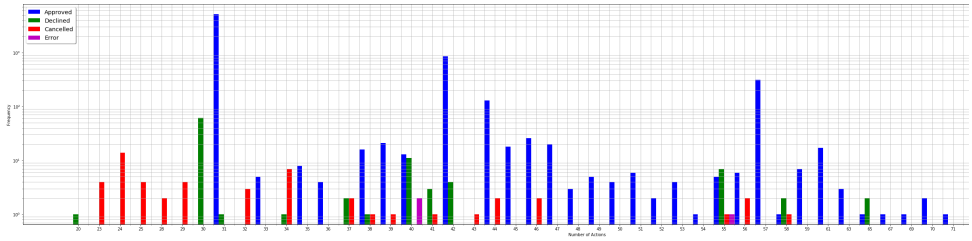


Figure 4.7: Correlation between Final Status and number of actions in live data

4.3 Types of Differences

The goal of this thesis is to identify differences between state machines, which are learned from transaction data. However the type of data we use do not restrict our solution and the overall method can be used with other data as well.

In this section the differences the algorithm aims to identify will be explained.

Case 1: Removed/Added State in Observed Graph

In this simple case a state that exists in the reference graph, does not exist in the observed one, however the previous and next states are the same (Figure 4.8).

Knowing that the following states are the same for both graphs, makes us understand that there was a feature removed or added in the observed behavior of the software. There may also be other reasons for one state to be removed or added. Nonetheless it is always important to know which states were removed in the observed graph in order to understand the causes.

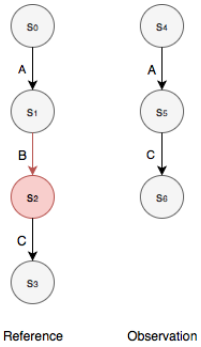


Figure 4.8: Comparison case 1

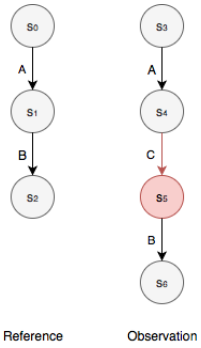


Figure 4.9: Comparison case 1

Case 2: More than one sequential states missing/added

When we are dealing with more than one wrong states that follow each other, the situation becomes much more complex. This might mean that a whole part is skipped or a whole new part added, like in Figure 4.10. However there are cases where this is not of great importance like when the states have just one outgoing edge. In that case even if it is a whole new part added, it is certain that the process will end up in the exactly same state as if the part was not existing at all.

Case 3: Renamed Edge

One type of differences that should be identifiable is the renamed edges. In that case we have an added state followed by a skipped state and the previous and next states are the same. An example can be found in Figure 4.11, where state C is replaced by state D in the observed graph.

Case 4: Edge changed location in sequence

It possible to identify a relocated state using the following pattern, like visualized in Figure 4.12. There is an added or removed state and later in the trace there is the

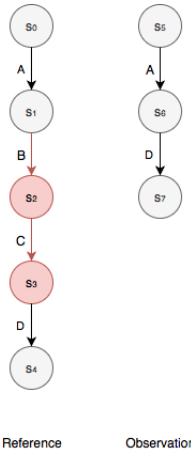


Figure 4.10: Comparison case 2

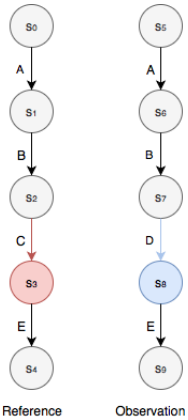


Figure 4.11: Comparison Case 3: Replaced Edge

same state with the opposite type, namely if at the beginning it was added now it should be removed and the other way around.

Case 5: Combination of multiple added and skipped state

In addition to the previous cases, it is frequent to have multiple changes that can not be easily explained. In that case it is enough to represent them in terms of added and removed states. Nonetheless this is the most difficult case, because the algorithm might easily confuse one trace with another and return wrong output.

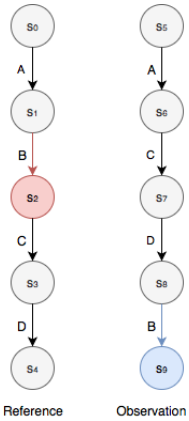


Figure 4.12: Comparison Case 4: Relocated Edge

4.4 Manual Exploration Findings

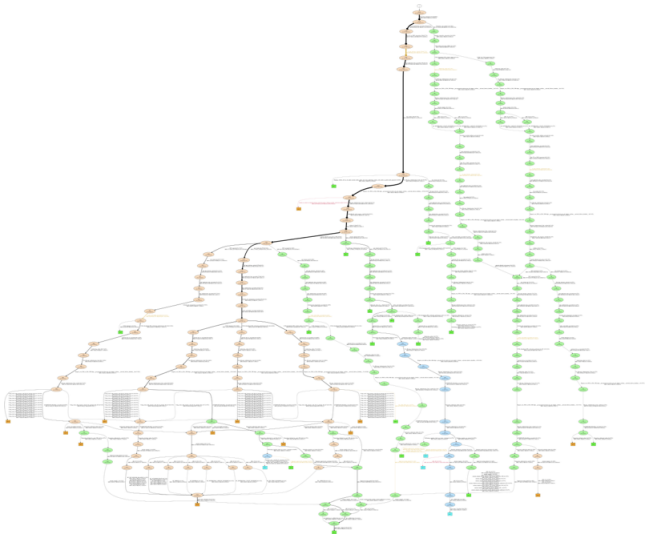
The first step before trying to automatically compare the data, was to manually find important differences. The idea was to compare various payment methods, cards and versions and see what kind of conclusions we can derive.

Comparing state machines learned from a whole test run of MSR (magnetic swipe reader) payments and one learned from the same version data on LIVE environment, a path that is not tested was found. In more detail, in the LIVE environment, a user tried to pay contactless, but the transaction was not successful and then tried to swipe the card. This is a path that cannot be found in the test model because the case where one payment method is tried, fails and then another one is tried is not tested.

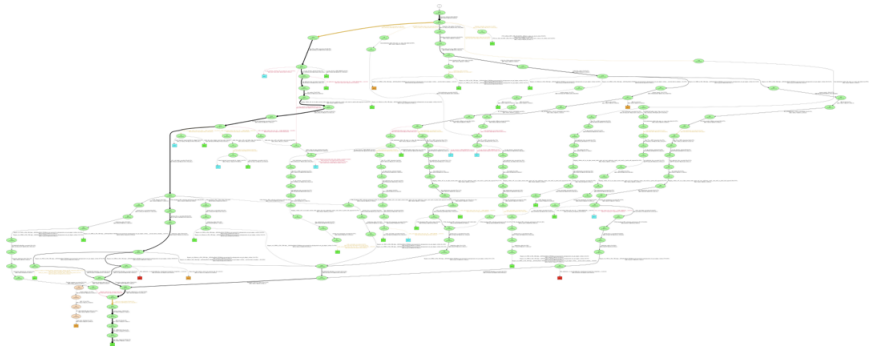
4.4.1 Structural Differences between Live and Test

The first analysis done, before implementing the comparison algorithm, was modeling the behavior of live and test systems and have a look to manually check how similar they are. However the results did not seem very promising, as visualized in Figure 4.13. The test behavior has more levels, which means that the transactions needs more steps before being marked as completed. On the other hand, the live behavior seem to be more spread horizontally. That means that more paths can be taken, in comparison to test.

Another interesting conclusion drawn by this manual comparison is what we already showed with the data analysis in Figures 4.6 and 4.4. As analyzed earlier, live transactions are usually `APPROVED`, but in test environment the most frequent final status is `DECLINED`. That is also visible in the following comparison, the test behavior in (a) has a big part on the left of the graph that is orange and the rest of the nodes seem to be colored green. On the contrary, in Figure 4.13 (b) all the nodes seem to be green, there are some orange, but the great majority is green, which constitutes another critical difference between live and test.



(a) Test Behavior



(b) Live Behavior

Figure 4.13: Big structural differences between the behavior of live and test environment for specific payment method.

4.5 Challenges

At the beginning, the idea to compare the modeled behavior of the system, according to their logs, seemed like the only solution, even though the theory behind graph comparison made it clear that it will be too heavy computationally. Especially, for the problem we are dealing with, some times multiple comparisons are needed in order to find the trace that looks more similar to the wanted one. The problem of comparing graphs is not known to be solvable in polynomial time nor to be NP-complete, and therefore may be in the computational complexity class NP-intermediate. In Adyen, hundred thousands transactions are performed each minute using the POS terminals, thus the comparison method should not take much time and it is not efficient to

model the behavior of the new incoming transactions in each iteration. Consequently the capability of the method to handle streaming data, played an important role in the decision of the algorithms that will be used.

Some of the challenges of the problem we are facing in this thesis are stated here:

- **Duplicate transitions.** A State Machine may have multiple transitions with the same label. A good replay approach should be able to tell to which transition the event belongs to.
- **Complex patterns**. Some times in a State Machines there might be complex states that deviate the accepting sequences a lot. The replay approach should not mistakenly consider the sequence as deviating because there is missing just one action from the trace.
- **Loops.** The behavior modeled by the State Machine may produce loops. These loops make it possible to create traces with infinite states. Nonetheless the loops a replay approach should still be able to map occurrences of events in the trace of the machine.
- **Sensitivity to deviations.** In cases where observed behavior is not allowed according to the State Machine, the followed approach must not be sensitive to every deviation. For example, deviations like adding a single state at the beginning of the trace, should not influence the mapping of events that occur later in the process. A replay approach should not stop after the first deviation, i.e., the alignment should map other events after the deviations.
- **Scalable.** With the availability of big data, it is extremely important for a replay approach to be able to deal with large logs, as well as large models. Consequently, the computation complexity and the memory requirements should also be taken into consideration when evaluating different approaches.

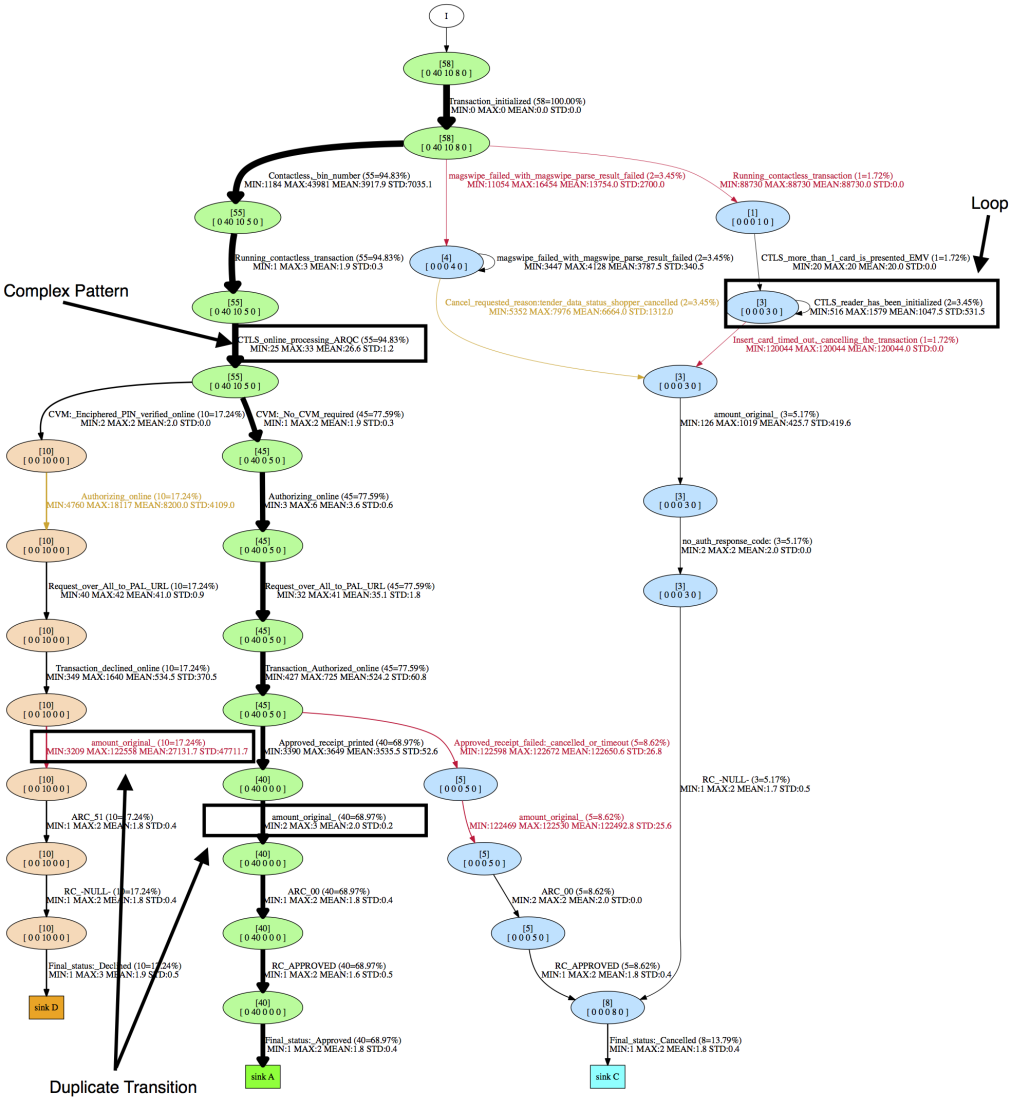


Figure 4.14: Challenges visualized in simple model

5

Model Selection

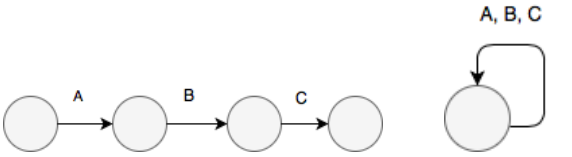
DFASAT, the tool used for the model learning, contains many features and functionality that change the model learned in huge degree. Setting all the parameters correctly is a pretty challenging task. In this Chapter, the aim is to provide the first intuitive explanation of the many parameters in Flexfringe and how to set them. In order to tune the parameters for our problem, many tries were necessary, since the data used for this thesis have some characteristics that should be taken into consideration for the choice of the right learning method.

5.1 Red Blue Merging Algorithm

When learning a state machine one of the most important components of this procedure is the state merging. Merging is used to minimize the initial representation of the data to the smaller possible representation in terms of maintaining the important information. Evidence-Driven State Merging (EDSM) starts with the construction of a augmented prefix tree acceptor (APTA) for the given data, then try all possible merges. Compute scores for each possible merge and perform the merge with the highest score [50], [72]. The idea behind computing the score is to measure the likelihood that a pair of states is equivalent. Then iterate.

The goal of merging two states is to combine them into one. Thus all input transitions of both nodes should point to the new node, which should also contain all the output transitions of both under merging states. The merge is allowed to happen only if the two nodes are consistent, that means that they should not be of different type (positive node merges only with another positive node and the same applies for negative nodes). However this can me modified and adjusted to the corresponding needs of each problem. In our case we have more than one type of nodes, there are the approved, declined, canceled and error nodes, according to the final state of each transaction. When a merge takes place and there are two same output transitions the target nodes are merged as well, this is called a determinization process.

There are many different merging algorithms, however in this section the merging algorithm of Flexfringe will be discussed. The merging algorithm of the Flexfringe we used, is similar to the red-blue fringe state merging algorithm from [50], which was



(a) No Final Red

(b) With Final Red

Figure 5.1: Final Red Parameter Usage

introduced to reduce the search space of the possible pairs for merging evaluation. The red-blue framework, as explained in [73], follows the procedure of merging mentioned above, but in addition it maintains a core of red nodes with a fringe of blue nodes. The red nodes are the identified parts of the automaton and the blue are the candidates for merging. The red-blue algorithm starts by coloring the root of the APTA red and its children blue. At each iteration the algorithm can either merge a blue node with a red node, or change the color of a blue node into red if no such merge can be found.

5.1.1 Final Red Parameter

Heule and Verwer [74] introduced a constraint on the state merging using EDSM to block merging from happening when it adds new transitions to a red state. This constraint is a consistency check which labels as inconsistent all the potential merges that add new outgoing transitions to an existing red node. According to [74] the reason for avoiding such merges is an assumption that red states are correctly identified parts of the model. Nonetheless it depends on the application use determining if it something that should be used. For the usage scenario investigated in this work the final red parameter improved the results a lot. As it can be seen from example 5.1 the addition of new transitions in a red node can create loops, which not only complicate the model, but also misinterpret the behavior of the underlying data.

In Table 5.1 the final red parameter's impact in comparison to the default parameters can be found. In Chapter 4 we analyzed the dataset and it seems that 500 states for the 17.300 traces that on average have more than 45 actions each are too few. Which means that the model learned with the default values is probably an oversimplification of the actual one. By setting the final red parameter to 1, once a red state has been learned, it is considered final and can not be modified. As explained above, this reduces the number of merges, thus we have a bigger size in the following table.

Another parameter that affects the red-blue state-merging framework is the *extend*. The impact of that parameter in our case can be found in the following Table 5.2. The *default* case contains all the parameters used by Flexfringe with their default values, which for $x = 1$. Then we just change $x = 0$ and everything else stays the same. The Table 5.2 indicates that the size of the unique states, when

Params	Alphabet Size	Time (msec)
default	562	2437.4170
f=1	1275	1111.5940

Table 5.1: Impact of final red parameter in a model learned by 17.300 traces.

changing the *extend* parameter, decreases by 17.8%. The impact of this parameter in the execution time is much more important. The idea behind this parameter in Flexfringe is that a merge candidate (blue) is only changed into a target (red) when no more merges are possible.

Params	Alphabet Size	Time (msec)
default	562	2437.4170
x=0	467	48.8862

Table 5.2: Impact of extend parameter in a model learned by 17.300 traces.

5.1.2 Lower Merging Bound

Another parameter which affects the quality of merging is the lower bound l. The lower bound determines the limit under which merge is considered inconsistent. In some cases, one of these, also the one we are dealing with now, it is better to colour a state red and do not merge it with anything new, rather than doing a bad merge. That can be succeeded by setting the lower required merging score to a positive value and exclude all the merges with score beneath that.

5.2 Sinks

The idea of sinks is pretty different in Flexfringe that it usually is in the Finite State Automata. Sinks are widely knows as the black holes of state machines. When some transitions lead to non-final states and cannot leave from these states we name them sinks.

However in Flexfringe sinks behave differently. As explained in [75] sinks contain the states that meet some user-defined conditions, like having low frequency counts, having only accepting traces (or none), triggering an error etc. Sinks make the model more interpretable because they hide irrelevant behavior of the system. Notwithstanding when comparing the behavior of various systems using state machines no information is irrelevant. The structure of the model is important and should not be changed by the sinks. Therefore, in this thesis the models learned will contain nothing in the sinks, except for the final state (the final states will be the sinks).

5.3 Heuristic

In theory heuristic defines the consistency checks for the merging. In *Flexfringe* the heuristics contain these consistency checks and the required corresponding calculation but also some other problem related functions. Flexfringe contains some well known

heuristics which can be used for specific categories of problems. The included heuristics are the following, as explained by Hammerschmidt in [76]:

- EDSM [50, 77], a heuristic for DFA identification using positive and negative data. The consistency check is based on the positive/negative labels and the heuristic score is the number of merged states.
- overlap [74], a heuristic based on EDSM used in the Stamina competition [78] for DFA identification. The consistency check is based on the overlapping outgoing transitions, the heuristic score is based on the number of overlapping transitions.
- KL-divergence [79], a PDFA heuristic using the KL-divergence between states as a heuristic. The consistency check is based on the distance between the distribution of states (as given by the KL-divergence).
- likelihood [80], a heuristic for PDFA identification.The consistency check selects a model according to its log-likelihood.
- Alergia, [76], a well known PDFA inference implementation. The consistency check uses the Hoeffding bound to measure the distance between distributions.
- RSME [76], likelihood for regression automata. The consistency check uses a mean-squared-error penalized likelihood selection condition.
- overlap4logs [6], a heuristic specifically for logs. It goes behind the idea of positive and negative data adding to types. The consistency check makes sure that the merging will be made with respect to the data type. The rare actions are grouped into a sink.

5.3.1 Adjustment of Heuristic

Most of the problems require a heuristic implemented for its specifications. Creating a heuristic means overriding some of the default Flexfringe functions to steer them towards the direction of the corresponding problem. The tool itself contains multiple heuristics for different categories of problems, nonetheless most of the problems require to adjust the tool for optimal results.

In the heuristic a consistency check and a scoring scheme can be developed for the specific problem as well as some better visualization can be added.

Two more functions that usually need some adjustment are the functions *find_end_type* and *sink_type*. In *find_end_type* what is considered as a final state can be defined. In *sink_type* is defined what we consider as a sink, in our case the sinks are the final states of the transaction flows. Additionally in *sink_type* the condition for adding thing in a sink can be defined.

In order to change the visualization of the model the functions *print_labels* and *print_dot* can be modified. In *print_labels* the data that the user wants to print on the edges are specified. By default Flexfringe prints the transition name and some statistics about its frequency. Additional information can be computed and printed and also some int value can be returned which can possibly determine the color of

the edge. This value can be accessed by *print_dot* to write the corresponding color to the dot file. Generally in this function what will be written in the dot file are determined, which means that the user can interfere with this function to change the visualization.

5.3.2 Heuristic of Log Comparison

For the log comparison, initially the overlap4logs heuristic was used, which however makes use of the sinks in a way that is not optimal for the comparison. As mentioned above, overlap4logs groups the rare transitions that lead to the same final state all together. When the goal is to compare models it is important to identify the differences that may not appear that frequently. Thus it is important to use also the information contained in the sinks. Transactions should always end with one of the predefined states. Consequently it is expected that in the model all the traces will lead to one of these four states. Which means that in our case these four final states can be considered sinks.

5.4 Statistical Checks

Flexfringe provides also the option to determine the minimum number of occurrences of states or transitions in order to be included in the statistical checks. Low frequency states and transitions may influence the statistical tests in an undesirable way, thus it is usually advised to set them higher than 0. However the business behind this thesis requires all the states and transitions to be taken equally into consideration. For that purpose the statistical tests are disabled by setting the parameters q and y, of the tool, to a very high value, regarding the count of states and transitions respectively.

6

Conformance Checking

The goal of this thesis is to find an efficient way to identify and visualize the differences between models. These models represent the behavior of some software, which in our case will be the one that runs on Adyen's POS terminals and for its inference, a lot of data are necessary. The data used for the learning process should cover most of the common flows (and preferably the uncommon as well). Consequently, for the case of Adyen, the models are very complicated. In Figure 6.1 the behavior of the contactless transactions using state machines is visualized.

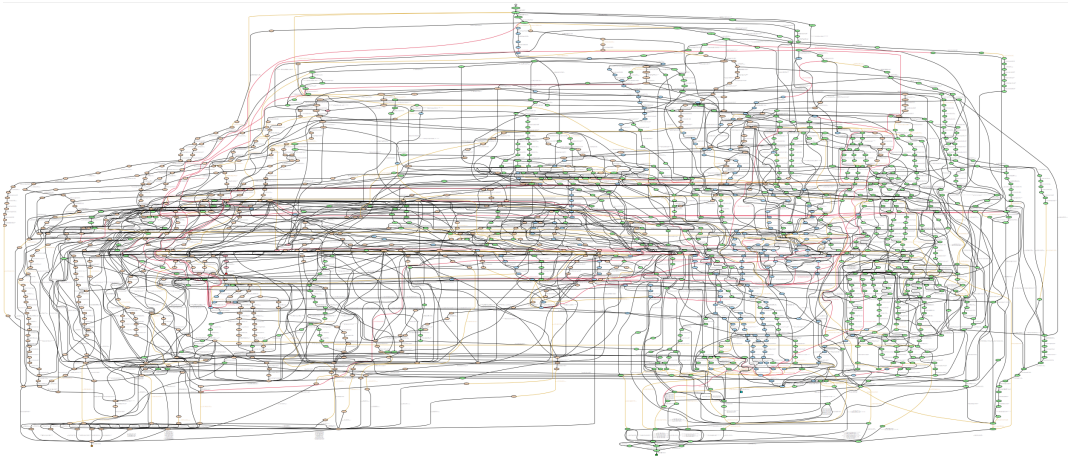


Figure 6.1: Model learned by 17300 test transactions of a software version

This image is just for the reader to get an idea of the size and the complexity of the models the algorithm should be able to deal with. Going through a graph like that is impossible for a human being, which makes the necessity of this project of higher importance.

In this chapter, the main contribution of this thesis will be presented. The proposed method is divided in three parts. The first part is the necessary pre-

processing of the data (Section 6.1). Secondly we have the differences' identification, which is implemented with three different algorithms (Section 6.2) and lastly the differences' handling (Section 6.3). An overview of the aforementioned steps can be found in Figure 6.3.

6.1 Preliminaries

The aim of this thesis is to develop a method, which will be able to efficiently identify anomalies in the behavior of the software using log data, on the fly. How the modeling of the behavior will take place is already known from previous sections and from previous work, in our case using state machines and more precisely a tool named Flexfringe [1].

The initial idea was to find and implement a graph comparison algorithm, which will receive as input two states machines, which represents the behavior of the software, with the goal to visualize the differences in a compact and clear way. However the difficulty we are facing is that it is not enough to find the first difference in the trace and then annotate it as divergent path till the leaf, as most of the related literature does [23, 25, 61]. It is really important to identify what did change that led to that divergent path.

Started implementing different graph comparison algorithms it was soon pretty clear that the complexity of comparing all the possible edges between them was unbearable for large amounts of data, especially if they change frequently, as presented in Chapter 10. The needs of this project, impose the development of a tool that will be able to handle the speed the transaction log data arrive, some times, hundreds of thousands entries per minute. The need for an algorithm without exponential complexity in terms of the exists paths was a main factor to the decision procedure.

Trying to compare discrete sequences, in our case transaction flows (which are actually sequences of states), is based on the same idea with the comparison between DNA sequences. The goal in both problems is to identify what makes them diverge at each point and modify them, so that they will become as similar as possible (align). Thus, this idea was used for the development of the comparison tool together with the replay concept of conformance checking, where traces are replayed in the model to check if they are validated or not. In addition to that, more than one algorithms were used for finding the optimal route to the requested node.

6.1.1 Pre-processing

In order to apply our proposed method, data that represent the behavior of the system are needed, and another set of dataset that should be checked if it is conformed with the model of the system. The first step of the pipeline is the pre-processing of the log files; this is fullfilled using the tool Rick Wieman [6] created. Then these pre-processed data are fed to a state machine learning algorithm, which in our case is the Flexfringe. The reason for choosing this specific tool is argued by the aforementioned work by Wieman (the tool there is named DFASAT, but we will

[1] `https://automatonlearning.net/flexfringe/`

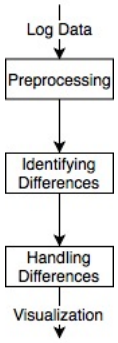


Figure 6.2: High level the structure of the method

refer to it with its new name Flexfringe). The Flexfringe outputs the model (finite state machine) that will be considered our specification graph.

6.2 Differences Identifications

The aforementioned problems and ideas led us to the pipeline presented in this section and visualized in Figure 6.3. An *assumption* taken for this pipeline is that we have some specification data and some incoming data that need to be checked. We assume that the specification data are able to represent the expected or wanted behavior of the software and are considered the basis using which, anomalies will be identified in the incoming traces. From now on, we will refer to the data used for the learning of the models as *specification data* and to the traces checked for anomalies, *observed data*.

Driven by the motivation to create a fast and effective system for large amounts of data, the decision to follow an approach completely different to what is till now used for log comparison was taken.

It is known that the modeling of the behavior takes much time (the learning procedure is time consuming), thus remodeling every time new data arrive is not a realistic option. For that purpose, the proposed method uses the model of the expected or wanted behavior of the software and the observed data that need to be checked from the logs, only pre-processed in text format.

The way we decided to find anomalies in the data is by examining if they are validated by the specification model. In this case, as we describe in Section 3 we perform conformance checking by testing if the observed traces are satisfied by the specification graph. The specification graph should model the expected behavior of the system. The more complete this model is the better results we will get.

The followed approach is split in two main parts. First is the pipeline for the detection of the differences and secondly the reduction/grouping of these results.

6.2.1 Main Pipeline Analysis

The proposed pipeline can be found in Figure 6.3. After parsing each trace from the log file, which contains the observed data (right part of the Figure), the first step is to go through the conformance checking algorithm. Conformance checking is responsible for identifying the traces that are and the ones that are not validated by replaying them in the graph. By validated is meant that the trace from the first state till the last one should appear in the model and should end up at the same final state. Each edge and state of the trace has to exist in the exactly same order in the graph. In order to achieve that, we traverse the whole tree, looking for the next transition of the incoming trace at each step. If at some point the next transition of the sequence can not be found in the expected order in the tree, the sequence alignment should start to check what went wrong.

To sum up the conformance checking part, if the trace is validated, it is annotated as existing and the algorithm continues with the next one, while if it is not validated, the differences have to be identified, which will be conducted by the sequence alignment algorithm.

The goal of the sequence alignment is to identify the parts of the trace that are same and the ones that are not. The trace is modified by adding and removing the necessary actions in order to make it aligned with the most similar one from the state machine. There are multiple differences that can be identified by the algorithm,

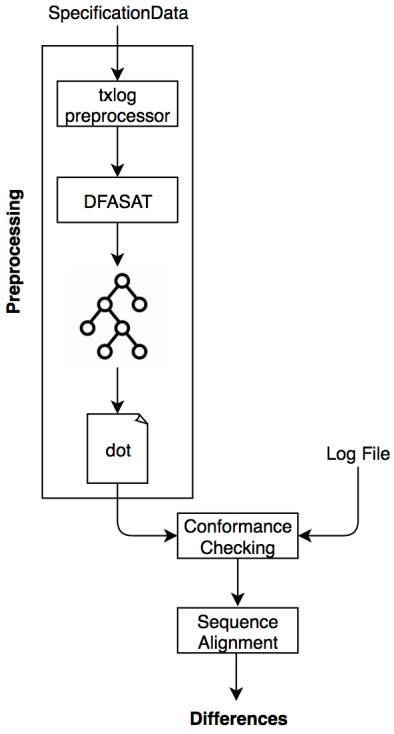


Figure 6.3: Main Pipeline of the System

extensively described in Section 4.3, but everything will be expressed in terms of *skipped edges* and *added edges*. The case where many consecutive skipped or added states appear should be investigated further, because over a number of successive skipped states, the paths are completely different and it does not make sense to find the parts that are the same.

The basic idea of the sequence alignment algorithm is explained in the Background chapter, section 2.6. There is an example presenting how two sequences can be aligned in 2.2. There, two strings are aligned, by adding gaps in the one sequence or the other. However, the algorithm presented here should only return all the modification in one of the sequences, since the other sequence, in our case is a whole tree. The result of the implemented algorithm for the Example 2.2 would be:

T/+, A, C/-, G, T, C/-, G/+, A/-, T, C, A

The + (plus) sign means that this action is added, thus it exists in the sequence but not in the tree. The - (minus) sign means that this action is removed, so the action does not exist in the sequence, although it was expected according to the tree.

The first two presented algorithms (Greedy and Best-First) are just looking for the misaligned state at each step and how they can fix it. These two algorithms are named after the searching algorithm that was used for locating the skipped or added

states.

In order to use some basic steps for all the algorithms, we modified the three choices of sequence alignment presented in Table 2.1, to suit the needs of the current problem, we end up with the following cases:

1. If the two actions are the same, a step is taken in both sequences.
2. One step can be taken in the first sequence, and if then the sequences align it is the case of an added state.
3. One step is taken in the second alignment (in our case the tree), if then the sequences align it is the case of skipped state.
4. In addition to the aforementioned cases, there is one more which is necessary because of the nature of the data. If none of the above cases is satisfied, probably the case of a replaced state is faced. That means if the algorithm moves one step in both the tree and the sequence, the next states are aligned.

Figure 6.4: Four cases for sequence alignment problem

How one of the aforementioned cases is chosen depends on the used algorithm. For that purpose we present three different algorithms, each one of them with different advantages and disadvantages. These algorithms will be presented and analyzed in the next subsection.

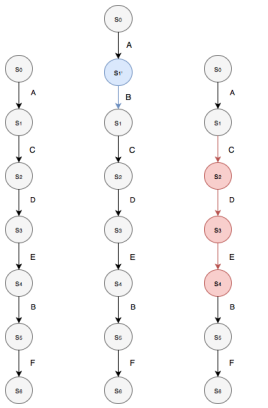
The alignment between the sequence and the tree was performed in three different ways. First the naive method, where at each step we are looking for the missing states. However the first solution is returned regardless of scores. Then an improved version where all four options mentioned in Figure 6.4 are examined at each step and the one with the best results with respect to the next states is chosen. Finally the dynamic programming approach is implemented, the only global sequence alignment algorithm which means that all states have to be taken into consideration and not just the neighboring ones as done by the previous two methods.

For each algorithm some advantages and disadvantages will be mentioned, which will be backed up in the Chapter 7.

6.2.2 Greedy Search Algorithm

The Greedy Search Algorithm is a modification of the breadth first algorithm which looks for a specific action in a limited area of the graph. The breadth first algorithm does not search in the whole graph, only the part of the graph under the node, which was lastly aligned with the wanted sequence.

The graphs are usually quite deep. For that reason the greedy algorithm is also limited in terms of the depth it will examine, as we can see in the next subsection.



(a) Transition Flow (b) Result With Depth (c) Results Without Depth

Figure 6.5: Usefulness of limiting depth case 2

6.2.3 Impact of Depth of Search for Greedy Algorithm

The proposed algorithm searches for skipped actions up to a specific depth. That is needed in order to force the algorithm stop after searching up to some depth and haven't found that action. The benefits of the depth are twofold: First is the runtime performance. If the algorithm continued running for every possible action until the leaves, the exponential complexity would make it unbearable for a medium amount of data. Second we prevent from identifying a wrong action as skipped because we searched too deep. For example using the model in Figure 7.1 and we want to align the sequence A,C,D,E,B,F as shown in 6.5(a), the results are the following: without limiting the depth, the action A is aligned, on the second step, when searching for B in the transition graph the B before F will be found, thus states C,D,E will be marked as skipped which is wrong. In reality, action B is added in the given sequence and does not exist in the transition graph after A and before C. Defining a depth of 3 will make the breadth first algorithm search for only 3 subsequent children and since it does not find B it should stop and mark B as added.
Advantages: It is relatively fast. The accuracy of this method is good.
Disadvantages: First found option is returned, which means that in some cases we do not return the optimal solution. Using this algorithm the search depth should be limited in order to return the most relevant result.

6.2.4 Best-First Search Algorithm

The Best-First Algorithm is an improvement of the Greedy one, where optimizations are used to make the implementation more efficient, but also the tree is more

extensively searched to find the best alignment. In more detail, at each step, where a transition is not in the expected order (thus the part of the sequence is not conformed by the specification graph) all four cases explained in Figure 6.4 are explored and the one with the smallest distance is chosen. In more detail, at each step, a queue is created with all the states that need to be explored, starting from the ones closer to the node under investigation. Each element of the queue is checked if it has as an outgoing edge the transition we are looking for and if yes the added and skipped intermediate states are computed (using back-propagation in the tree). We continue this procedure till a specific depth of the tree is fully explored and when the iteration stops, the results are returned together with a number that shows how far the aligned part is from the current node. From the possible alignments, the one with the smallest distance is chosen, which should be the one with the minimum modifications.
Advantages: Very fast method. Very good performance.
Disadvantages: Not the whole sequence is taken into consideration. Only the neighboring nodes to the one under investigation are checked, which means that the returned output may not be optimal.

6.2.5 Dynamic Programming

In Chapter 3 an example of how global sequence alignment using dynamic programming works for two sequences is presented. For the sake of this project, we need to align one sequence with the whole tree, thus with every sequence that can be generated using the tree (specification graph). In order to achieve that, we modified the well known sequence alignment algorithm by Needleman and Wunsch [42] to align all the sequences that can be generated by a tree with one trace (the sequence under investigation).

Algorithm Description

For each node of a tree we can define its level.

> The level of a node is defined by 1 + the number of connections between the node and the root. It starts from 1 and the level of the root is 1.

However since in this project we are mainly working with transitions, each transition of the tree will get a level assigned based on its distance to the root. The matrix for the tree/sequence alignment will have dimensions *#of transitions in the tree* × *#of transitions in trace*. The main steps of the algorithm are the following:

1. Initialize matrix
2. Compute values for matrix (Pseudocode 6.1)
3. Bottom-Up align sequences

During the initialization phase, the matrix with the correct dimensions and keys (for column and row identification) is created. The rows of the matrix represent the transitions of the tree and the rows the transitions in the observed sequence, that needs to be checked. The first row and the first column of the matrix always represent the GAP character and the scores are initialized based on the GAP penalty.

The second step of the algorithm is presented using the pseudocode. For each transition of the incoming trace, the scores at each row of the matrix should be computed. Thus for each cell, the three neighboring values should be returned, the left one, the one above and the one diagonally left up, as visualized in Figure 6.6. From these three values the maximum is chosen and based on their position encoding (0 for left, 1 for diagonal and 2 for above) the penalty that should be added is determined. A necessary modification of the original algorithm is that these previous values in our case are not always in the previous row and previous column. The table has to be searched to find the previous values such that their level is the current level -1 and the destination of the key should be the source of the current transition. Thus in the function *getThreeNeighboringCells* the matrix is search for keys in the rows that satisfy: *previous_level = current_level -1 and previous_destination == current_source*, where current is the row whose value we are computing. If the maximum value is in the cell above or left to the current one, a GAP penalty is added. If the maximum values comes from the diagonal cell, it has to be checked if there is a MATCH or MISMATCH (pseudocode 6.1 line 8).

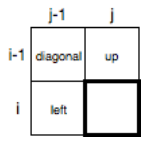


Figure 6.6: The bold cell is the one, whose value we are trying to compute. The rest are the ones that have to be returned.

Pseudocode 6.1: Dynamic Programming step 2: Compute matrix values

```
1  function computeMatrix(matrix, incoming_trace)
2      for each transition in incoming_trace:
3          for each table_row_key in matrix:
4              value, position = max(getThreeNeighboringCells())
5              if position == 0 || position == 2:
6              else if position ==1:
7                  if transition == trable_key_row.getTransition():
8                      writeMatrix(table_row_key, transition, value+MATCH)
9                  else:
10                     writeMatrix(table_row_key, transition, value+MISMATCH)
```

When the matrix is full, the alignment between the incoming trace and the most similar sequence of the tree can be found. Starting from the last column, the highest number is chosen and the corresponding row and column at that position are the items that can be aligned. Continuing to the next column we do the same and the consistency between the two subsequent transitions is checked. If the destination of the previous element is not the source of the current one, there are skipped transitions in between. If a row has more than one maximum value per column then we face an added transition.

This procedure will be better explained using an example.

Example

The idea is to annotate every transition with a level, according to their distance from the root. As it can be seen in Figure 6.7 the *Transition* $0 \rightarrow 1$ and *Transition* $0 \rightarrow 2$ have level 1, since it is the first transition after the root (default initial transition), *Transition* $1 \rightarrow 3$ and $2 \rightarrow 5$ are in the second level etc. Dynamic Programming requires at each step to get the maximum value of the three possible previous positions.

In our case we have to identify the previous level first and get the corresponding values accordingly. To the maximum value of the previous level, that leads to our current position, the penalty of the action that should be taken is added. As in the original algorithm we have penalties for match, mismatch and gap.

In the Table 6.1 the cost matrix for the alignment between the specification graph shown in Figure 6.7 and the sequence *"afee"* can be found. The costs can be found in Table 6.2. Always the first row and column of the matrix contain the *gap* action, which is indicated by $-$. At each step we check the values of the previous level, for the transitions that have destination the same as the current transition's source. With red color the highest score for each column is annotated, going from the bottom up, we see that the last e is aligned with the transition $8 \rightarrow 9$ of the graph, the other e with the transition $3 \rightarrow 4$. We notice that between transition $3 \rightarrow 4$ and $8 \rightarrow 9$ some edges are missing. The missing edges are the *skipped* edges. Then we continue with f, which is aligned with $0 \rightarrow 1$, but also a is aligned with the same transition. That means that after the first symbol is aligned with a transition all the rest do not exist in the graph, thus they are the so called *added* edges. Consequently f is *added*, and $4 \rightarrow 6$ (which is the symbol w) and $6 \rightarrow 8$ (symbol l in the graph) are *skipped.*

Finally, the algorithm will return: **a, f/+, c/-, e, w/-, l/-, e**.

level	Transition	-	a	f	e	e
-	-	0	-2	-4	-6	-8
1	**0** $\rightarrow$ **1**	-2	4	2	0	-2
1	**0** $\rightarrow$ **2**	-2	-4	-6	-8	-10
2	**1** $\rightarrow$ **3**	-4	2	0	-2	-4
2	**2** $\rightarrow$ **5**	-4	-6	-8	-10	-12
3	**3** $\rightarrow$ **4**	-6	0	-2	4	2
3	**3** $\rightarrow$ **5**	-6	0	-2	-4	-6
4	**4** $\rightarrow$ **6**	-8	-2	-4	2	0
5	**6** $\rightarrow$ **7**	-10	-4	-6	0	-2
5	**6** $\rightarrow$ **8**	-10	-4	-6	0	-2
6	**8** $\rightarrow$ **9**	-12	-6	-8	-2	4
7	**9** $\rightarrow$ **10**	-14	-8	-10	-4	2

Table 6.1: Dynamic Programming matrix for the whole tree

Penalty	Value
match	4
mismatch	-4
gap	-2

Table 6.2: Example cost table for dynamic programming approach

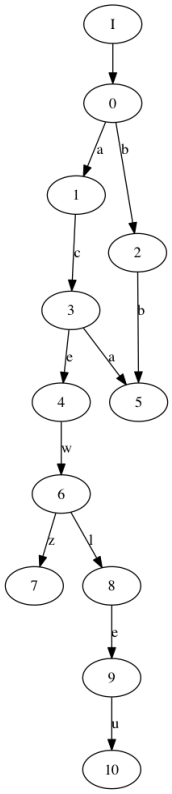


Figure 6.7: Tree with which the example sequence is aligned

Dynamic penalties

In the problem we are dealing with, the sequences of the tree may be long and many gaps might be added in the middle. That creates the problem that if a transition close to the leaf is identified as a *match*, the influence of that might be negligible to the overall score and the optimal alignment might not be found.

For that reason, dynamic costs are used. More specifically the reward for a match is increased as we go closer to the leaf. That approach has improved the results, but it is still unclear how the penalties impact the performance thus, it has to be investigated further.
Advantages: It can be mathematically proved that if there is an alignment, the algorithm will find it.
Disadvantages: It depends on the penalties a lot. With different values for the scores, different outputs are produced by the algorithm. Very slow.

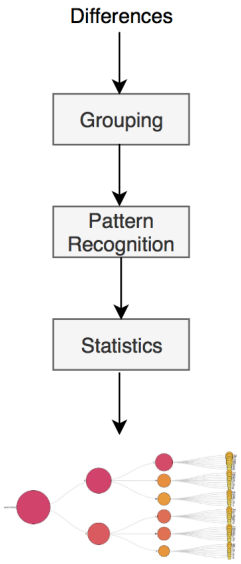


Figure 6.8: Handling of differences

6.3 Handling the Differences

The previously presented algorithms return the traces that were not conformed by the graph and the corresponding differences in terms of *added* and *skipped* edges. Each invalidated trace usually contains more than one differences which should be somehow transformed in a human recognizable format. In addition to that, there are many traces that present exactly the same differences when compared with the graph. Consequently, there are some next steps that should be performed, in order to improve the output of the algorithm and make it as comprehensible as possible. The next step is to group the traces with the same differences together, next extract patterns from the data and use them to transform the sequences of differences to more meaningful information. The followed steps for the handling of the differences are presented in the Figure 6.8.

Group Results

As mentioned above, the algorithm returns the traces and their corresponding differences to the model, however there are cases where multiple traces have exactly the same differences to the graph. The statistical analysis of the appearances of the differences may help to the explanation of their meaning and their importance. For that purpose the traces that were not satisfied by the graph are grouped based on their differences. For example, suppose the following traces and their corresponding differences in 6.3; these two traces would be grouped together even if they are not

the same, since there differences are exactly the same.

Trace	*Difference*
Trans_init, Pin_Entry, Connect_Backend, Validated, Approved	Validated: added
Trans_init, Contactless, Connect_Backend, Validated, Approved	Validated: added

Table 6.3: Example of two grouped traces

After that step, the grouped traces are returned, which decreases the amount of results to a great degree. For the example, in Table 6.3 only one trace would be returned after the grouping, which would be the representative of the specific group.

Additionally, the distribution of the group sizes could give us some intuition about the importance of the difference. In case there are some groups with outlying size, these groups might contain either unimportant or extremely important information. This is something that again requires further investigation.

The first step that should be performed is some clustering in the differences. The traces that have the same differences should be grouped together. The next step is to check the distribution of the cluster size.

Pattern Extraction

Looking at the differences the data presented it was clear that they were following some patterns. Usually there were some states that changed names in the new version of the software, or had changes order. These two are the most frequent changes between the data.

The following patterns will be used for the identification of the meaningful differences between the data. In more detail we have:

- Renamed state → edge1 added, edge2 skipped and their previous states are the same as well as the next states.
- Relocated state → edge1 removed, edge2 added and the two edges have the same name .

Representation of the Differences

The importance of the representation of the differences was pretty clear from the first stages of this thesis. The fact that the learned models are very complicated makes the visualization of the difference in the graph very difficult. After many discussions with the developers, of the software that we model, they proposed to visualize the log lines that appeared to be different between the two datasets. Since they are very familiar with the log files we decided to show to the developer the log line that is not validated and the modification that were necessary in order for the trace to get accepted. Again in terms of *added* and *skipped* actions.

The tool that was used for the visualization is d3.js [2]. Color mapping for the different types of actions, *added*, *skipped* and *existing in both logs* with colors blue, red and green respectively.

[2] https://d3js.org

Prefix Tree

A prefix tree, also called digital tree, radix tree or trie, is a kind of search tree, an ordered tree data structure used to store a dynamic set where the keys are usually strings. In this case the keys will be the actions of the transaction. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. This is extremely helpful for the visualization of the point (the action) that diverges the traces, which is the root cause of the difference. The nodes represent the state the transaction ends up, after following a specific action. The edges are labeled with the name of the corresponding event. An example of the visualization can be found in Figure 6.9.

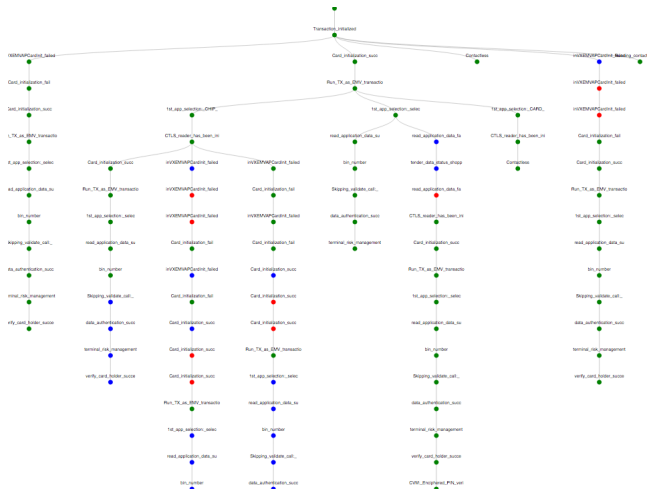


Figure 6.9: Example of the implemented Prefix Tree

As it can be seen from the Figure 6.9, the colors show the differences and the similarities between the logs. With *green* the actions that are aligned are shown, with *red* the actions that were in the model, but not in the test traces and with *blue* the actions that do not exist in the model but were introduced in the test traces. At the end of the trace, on the leaf, is the id of the transaction for the developer as a reference for the user to investigate the problem of that transaction in Adyen's system.

Additionally some features are implemented in order to make the visualization simpler, like having collapsible nodes to minimize the printed results. Hovering over the edge labels, additional information about the type of difference and the specific event are showed.

6.4 Efficiency Analysis

In Table 6.4, the runtime complexity of the various parts of our methodology is estimated. Even though with the first look the complexity of all algorithms seems to

be almost same, it has to be made clear that in the Greedy and Best-First algorithm just a small part of the tree (specification graph) is searched. On the other hand for the dynamic programming approach for the whole tree (which usually is very big) a scoring table is created. Consequently $k >> m$, since with fixed m usually around 25 transitions are investigated and k is more than 1000. It will also be proven in the Chapter 7, that the runtime performance of Greedy and Best-First algorithms is much better than Dynamic Programming.

Algorithm	**Complexity**	**Explanation**
Conformance Checking	O(n)	n: number of actions per trace
Sequence Alignment (Alg. 1 & 2)	O(n*m)	n: number of actions per trace m: depth of search in tree
Sequence Alignment (Alg. 3)	O(k*n)	n: number of actions per trace k: transitions of the tree
Total for Greedy & Best-First	**O(l*n*m)**	l: number of traces n: number of actions per trace m: depth of search
Total for Dynamic Programming	**O(l*n*k)**	k: transitions of the tree n: number of actions per trace l: number of traces

Table 6.4: Runtime complexity of the various algorithms used, Algorithm 1: Greedy, Algorithm 2: Best-First, Algorithm 3: Dynamic Programming

Accordingly, investigating the space complexity of the sequence alignment algorithms the space needed for dynamic programming is much more than the other two approaches. This happens because for the dynamic programming a table with dimensions $k * n$ has to be computed for each trace. The other two methods, do not store any other information, except of the chosen alignment for each incoming action.

Algorithm	**Space Complexity**	**Explanation**
Greedy	O(n)	n: number of actions per trace
Best-First	O(n)	n: number of actions per trace
Dynamic Programming	O(n*k)	k: transitions of the tree n: number of actions per trace

Table 6.5: Space complexity for each trace

7

Experiments

The evaluation of the log differencing tool is not an easy task. The data provided by Adyen are real transaction logs from various releases of the POS terminal's software both in live and test. Never before has been tried at Adyen to compare the behavior of their software in order to identify the runtime differences, thus, it was more or less an unknown field for everyone. The proposed pipeline is completely different to what has been tried by related work for the same problem. Consequently there is no direct comparison with various methods, nonetheless we implemented three different algorithms which will be extensively compared in the current Chapter.

7.1 Experimental Configuration

The algorithms presented in this work are implemented in Java. However the analysis and the mutation of the data, as well as the evaluation is performed in Python (by calling the jar files). The visualization of the differences is accomplished with the d3.js library, thus there is also javascript and html code for the handling the results of the algorithms.

All the experiments run locally in a Macbook Pro with 16GB memory and processor Intel Core i7 at 3.1GHz.

7.1.1 Experimental Approach

The evaluation of the presented problem is not that easy, because it is not a classical supervised learning problem. The algorithm returns for each trace the differences that are presented with respect to the specification model. Unfortunately there are no labels given and in the production level manual inspection of the results is necessary. Nonetheless to test the performance of the current, tool mutation testing will be performed, as a way to artificially create labeled data and then evaluate the tool using them.

The results will be evaluated in trace level, meaning all the returned differences should be correct for the result to be considered correct and to difference level, where each returned difference will be checked. In the second case some measures from the confusion matrix will be used. The notion of True Positives - TP (differences, that

were correctly identified) is helpful in this problem, as well as the False Positives - FP (the algorithm thought is a difference, but it is not). The False Negatives -FN will be computed as the complement of TP thus, the differences the algorithm should have found but didn't. The True Negatives - TN (the parts of the sequence that do not have any differences and were correctly identified by the algorithm) will only be computed, when accuracy has to be estimated. In the trace level, the whole sequence will just be marked as correct or wrong.

The evaluation will be conducted using two different datasets, one dummy and a real one. The dummy is used to explain the tool functionalities and the real one to evaluate the performance of the algorithms. In the real dataset there are no labels, therefore the concept of mutation testing will be used. The original data will be modified and the types of modifications will be used as labels, in order to create a labeled dataset.

7.2 Evaluation with dummy Dataset

This section presents some examples of the methodology explained in 6.3 using a dummy dataset to make clear the output of the algorithm as well as the expected to work and fail cases.

The following example shows the specification graph and the traces that had to be checked. In the table 7.1, for each trace, the output of the algorithm is shown. In addition, the third column explains in which case of difference each output refers to.

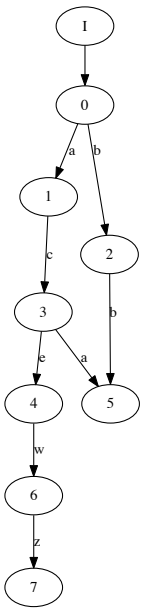


Figure 7.1: Specification State Machine for the example

Data	Algorithm Output	Case
a, b, b	Added State→a	Added State
a, c, e, z	Skipped State → w	Skipped State
a, e, f, w, z	Skipped State → c, Added State → f	Both Added & Skipped
a, z	Skipped States → c, e, w	Multiple Skipped States
a, w, z, c, e, w, z	-	Multiple Appearances [1]

Table 7.1: Algorithm input and output for various cases

The first case in Table 7.1, starts with the symbol a, which exists in the graph and leads to node 1, which however does have only one outgoing edge, c. In the observed sequence, a b has to follow the a. Consequently we have to check two scenarios. One is the possibility that the transition a is added and one that there might be some skipped states between a and b. Looking at the graph in Figure 7.1 we notice that there is no b at any of the children, if at the first step the a is followed. We can conclude then, that a is added since the sequence b,b is validated by the specification graph.

The sequence at the second row of Table 7.1, is starting with the symbol a, which exists in the specification graph at the first level and it takes us to node 1. Then the symbol c is an outgoing transition of node 1 and it takes us to node 3. The next symbol of the sequence is e and it leads us to node 4. The final symbol, z is not an outgoing edge of node 4, thus it is either skipped or added. The skipped case is checked first and it seems to be the case since one of the children of that node have z as an outgoing edge. The node is number 6, thus the transition w is skipped.

The same procedure is followed for every trace that has to be aligned. The last sequence is a bit challenging a,w,z,c,e,w,z, the output of the algorithm depends on the depth parameter. The output can be either $a,w/+,z/+,c,e,w,z$ which has edit distance 2, or $a,c/-,e/-,w,z,c/+,e/+,w/+,z/+$, whose distance is 6. Nonetheless the first two algorithms (Greedy and Best-First) that do not consider the whole sequence but just the closest n-neighboring nodes (according to the depth parameter), will output the second alignment. Dynamic Programming works for these complicated case, since it aligns the whole sequence, before returning the one with the minimum cost.

However when the method has to be applied in real company data way too many differences were returned, namely for 2144 traces 144 traces had one or more differences. Thus the new step of the algorithm was introduced, which clusters the traces based on the differences they have. In that way it was feasible to examine if a difference frequently appears in the results.

[1] the correct output for this case depends on the depth allowed to perform depth-first, see Subsection 6.2.3.

Match	Mismatch	Gap	Dyn.	Mutation	Accuracy	F1-score
3	-4	-2	yes	2 added	53%	69%
3	-4	-2	no	2 added	51%	65%
4	-2	-2	no	2 added	18%	46%
4	-2	-2	yes	2 added	25%	47%
6	-2	-2	no	2 added	14%	24%
6	-2	-2	yes	2 added	12%	31%
6	-4	-2	no	2 added	56%	70%
6	-4	-2	yes	2 added	51%	71%
6	-6	-2	no	2 added	55%	70%
6	-6	-2	yes	2 added	51%	67%
6	-6	-4	no	2 added	54%	70%
6	-6	-4	yes	2 added	53%	69 %
6	-6	-6	no	2 added	13%	22%
6	-6	-6	yes	2 added	20%	33%
6	-10	-6	yes	2 added	50%	66%
12	-10	-6	no	2 added	55%	70%
12	-10	-6	yes	2 added	53%	69%
14	-10	-6	no	2 added	59%	74%
14	-10	-6	yes	2 added	56%	71%
14	-12	-6	no	2 added	50%	66%
14	-10	-8	yes	2 added	49%	65%
14	-10	-8	no	2 added	57%	72%

Table 7.2: Evaluation metrics for various parameter values. The presented values are the average of 10 executions.

7.3 Parameter Tuning for Dynamic Programming

As mentioned in 3.6.1, the penalties influence the outcome of the sequence alignment using dynamic programming in a huge degree. Nonetheless there is no standard way to find the best values, additionally it is heavily dependent on the problem. Since sequence alignment, using dynamic programming, is not widely used for log differencing, we will greedily try to understand the impact these scores have in our implementation. For that purpose the algorithm will be run with various values and the accuracy and F1 score will be computed in the difference level. The mutations are random and for consistency purposes each experiment is run 10 times and the average accuracy, recall and F1 score are presented in Table 7.2.

It is interesting to notice how small changes in the penalties influence the accuracy of the tool. For example for the following setting: match=3, mismatch=-4 and gap=-2 (no dynamic costs) the accuracy is 51%, however when match=4 and mismatch=-2 the accuracy drops to 18%.

7.4 Difference Detection in Real Dataset

During the first phase of the mutation testing, new states are randomly added and some are randomly removed. However the last state cannot be neither removed nor changed (since there are four fixed final states). That's the only necessary restriction.

7.4.1 One Change per Log Line

First the most simple mutation of the data was tried. Each line is modified by adding or removing an existing event of the transaction log line.

For this testing scenario the specification graph is learned from 17300 transactions of the testing environment of *Adyen*. The fact that the data are from the testing environment it means they are pretty distinctive and they create a more wide graph than the same amount of real data would create, because many different scenarios need to be covered. The used data are the result of filtering of an initial dataset in order to contain *contactless* transactions. This aims to the reduction of the graph and the modeling of the behavior of a specific form of payment. However if the user tries contactless it fails and then tries a different method this will also be included in our dataset. The reason for choosing specifically *contactless* transactions is because they constitute the most common and consequently the most extensively tested method. The incoming data that we will try to match with the graph will be again these 17300 transactions but altered, in a way that they do not exist in the specification graph.

In the Table 7.3 we can see how the algorithms perform for each mutation. The same experiment runs 10 times with random modification in the incoming data. One important observation from these results is that there are some transitions that are much more important than others. When these transitions are removed the algorithms mixes up the observed trace with others that look almost the same, the impact the mutation's position is further investigated in Subsection 7.4.4.

Algorithm	# Traces	Mutation	TP	FP	Time (sec)
Greedy	17300	1 skipped	15421	1819	259.9781
Best-First	17300	1 skipped	16327	917	11.4503
Dynamic Pr.	17300	1 skipped	10823	4550	12780.6448

Table 7.3: One skipped state

In Figure 7.2 the time for every part of the tool for the different algorithms is visualized. Using the optimizations of Flexfringe [2] the time needed for the model learning is almost negligible. The preprocessing needs triple the time of the model learning procedure. The alignment algorithm (together with the conformance checking) take a lot of time for Greedy and DP. Especially for DP we had to divide the time by 10 to make it visual. The Best-First Algorithm needs the same time as the model-learning procedure. The burden of DP is extreme if we compare it with the other methods.

[2] `https://automatonlearning.net/flexfringe/`

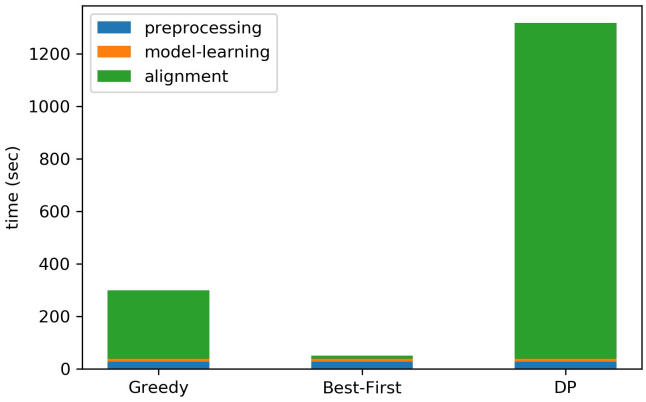


Figure 7.2: Comparison of the overall tool for the different algorithms. Important! The time for DP is divided by 10, for visualization purposes.

7.4.2 Two Sequences Changes per Log Line

In order to test how the tool performs for more complicated differences, two subsequent mutations are conducted in each transaction log line. Table 7.4 clearly shows that the DP approach performs worst for two skipped or added transitions. In addition to that, it also needs much more time to execute in comparison to the other two algorithms. The Best-First approach seems to be the most efficient (both fast and good performance) for the two subsequent mutations. From Table 7.4 some conclusions about the performance of the algorithms for the different mutations can be drawn. For example Best-First performs better for added transitions in comparison to skipped ones, on the other hand the exactly opposite happens with the greedy algorithm.

Algorithm	**# Traces**	**Mutation**	**TP**	**FP**	**Time**
Greedy	17300	2 skipped	29284	5225	238.49357
Best-First	17300	2 skipped	30464	4016	10.8728
Dynamin Pr.	17300	2 skipped	20712	13341	13404.1993
Greedy	17300	2 added	28547	6002	243.9584
Best-First	17300	2 added	32360	2237	8.1962
Dynamin Pr.	17300	2 added	24673	9994	15692.8251

Table 7.4: Difference level Evaluation. Two subsequent mutations (either two skipped or two added transitions), for the DP the penalties are: match=3, mismatch=-4, gap=-2

7.4.3 Multiple Sequenced and Unsequenced Modifications

In this section, the performance of the algorithms for multiple mutations are investigated. From Table 7.4 it was shown that the algorithms Greedy and Best-First perform better for minor mutations. In this section, more severe mutations are examined. As we can see in Table 7.5 when new states are added that haven't been encountered before the algorithm always finds them. On the other hand, as we increase the number of randomly added states from the existing ones, the performance of the Greedy algorithm drops.

In Table 7.6 the performance for the Dynamic Programming is presented. The more transitions are added the better the algorithm can find them. As presented using the dummy data, the fact that dynamic programming aligns the whole sequence before determining the modifications makes it work best for multiple mutations.

Size	*# Traces*	*Mutation*	*Correct*	*Wrong*	*Accuracy*	*Time(sec)*
17300	17300	50 new	17300	0	100%	1282.7432
17300	17300	100 new	17300	0	100%	1788.1801
17300	17300	1 existing	14657	2643	84.7225%	303.5634
17300	17300	2 existing	12391	4909	71.6242%	357.9557
17300	17300	3 existing	10767	6533	62.2369%	461.3995

Table 7.5: Results for added states either new or existing with Greedy Algorithm, trace level evaluation.

Match	**Mismatch**	**Gap**	**Mutation**	**Accuracy**	**Recall**	**F1**	**std**
14	-10	-6	3 added	65%	100%	78%	2.1447%
14	-10	-6	4 added	73%	100%	84%	2.6833%
14	-10	-6	5 added	76%	100%	86%	3.5213%
14	-10	-6	6 added	81%	100%	89%	0.8944%
14	-10	-6	7 added	83%	100%	90%	1.0954%
14	-10	-6	20 added	94%	100%	96%	0%
14	-10	-6	100 added	99%	100%	99%	0%

Table 7.6: Performance of Dynamic Programming for multiple subsequently added transitions.

7.4.4 Impact of the Mutation's Position

We investigated further the impact of the position of the mutation to the amount of errors per execution. Again the experiments are conducted with 17300 traces, were we perform two simple mutations and a complex one, namely replaced state (more information in Section 4.3). Then for the results we count the number of mistakes for each position that the corresponding mutation took place. It is obvious in Figure 7.4, that all the mutations behave in a similar way. In more detail, when the mutation happens in one of the first 5 states, the amount of error is the highest. That happens because the first actions in a transaction are important, since they

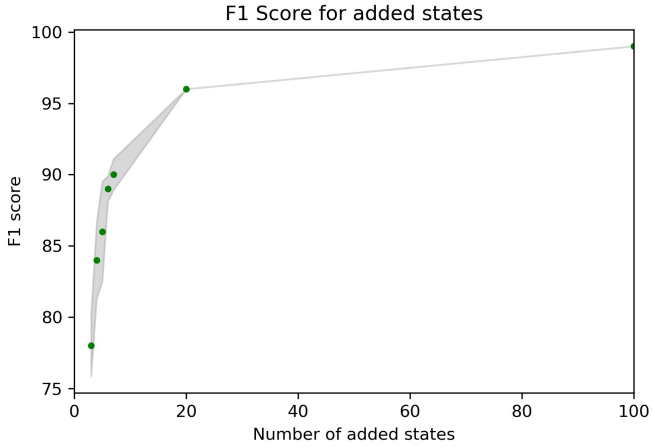


Figure 7.3: F1 score visualized with error, information from Table 7.6.

determine the way of payment for example. For added states close to the end of the sequence (after position 40) there are no errors.

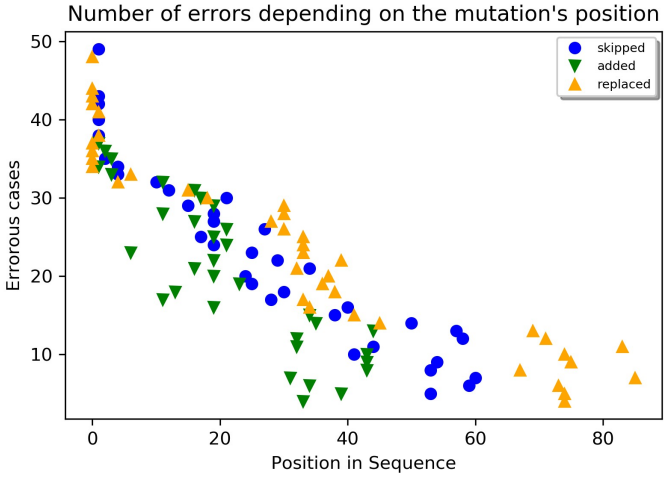


Figure 7.4: Number of errors per position in the sequence

Most of the mutated transactions are almost impossible to happen in real life scenarios because the removal of some states means either we have a different payment scenario or something is really wrong with the system.

7.4.5 Impact of Model Size in the Accuracy

Another attribute that influences the performance of the log differencing tool is the model size. As expected, the bigger the model the more complicated, thus the more difficult to identify the requested difference. When the models are big there are many traces that look alike, which means that with the mutation testing, when a trace is modified the algorithm may think that it looks more to a different trace. The impact of the model's size in the performance is visualized in Figure 7.5. The increase of the error rate is severe for just 5000 traces more (from learning a model using 12300 traces to 17300), the error from 4.5% goes to approximately 12%. Increasing the traces by 5000 more, thus learning a model from 22300 traces increases the error rate 6% more.

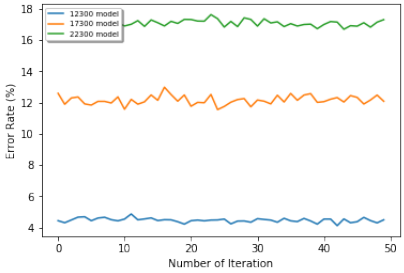


Figure 7.5: Error rate of the tool for difference model size.

7

7.5 Scalability

As mentioned multiple times, one of the goals of this thesis is to find a method, which will scale up to a huge amount of traces that should be investigated and to the number of added states. We will focus on the added because they increase the number of iterations.

Having a fixed specification graph, we experiment using various number of added states. It is obvious from Figure 7.6 that the Best-First algorithm is the fastest and Dynamic Programming the slowest. Comparing Best-First and Greedy approach, the latter seem to be slower (more than 10 times). This is not really related to the algorithm itself, rather to the fact that in the Best-First implementation many optimizations for speed up and memory were used.

In the following table 7.7, the time needed for the executing of each algorithm for just one trace can be found. It is obvious the the Best-First algorithm is the fastest one, having a huge difference with the dynamic programming approach, which is more than 100 times slower.

The presented results prove that the application of a tool like that in production is definitely feasible, especially with the Best-First algorithm, which is the fastest. The fact that each observed trace is separately aligned makes the use of parallelism possible, therefore even dynamic programming which is relatively slow can be applied.

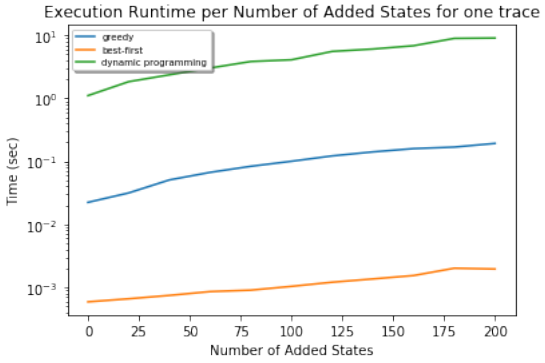


Figure 7.6: Runtime for various numbers of added states

Algorithm	Time Per Trace (sec)
Greedy	0.015028
Best-First	0.000662
Dynamic Programming	0.996206

Figure 7.7: Running time for one trace for the three different algorithms

7.6 Identified Differences

7

In this section, the actual results of the comparison with real data will be presented. Even though somebody could expect that there will be just few differences between live and test data, or between two subsequent software releases, that did not seem to be the case.

7.6.1 Live and Test Differences

An important result of this research is that test and live behavior of a software can be very different. For that experiment, the test environment was modeled using 17300 traces, the data of the robot transactions for a week. The observed data used, were 2200 traces from the live environment. 600 transactions out of the 2200 were not conformed by the model using the replay technique, thus they had to be aligned. After the alignment we had 67 groups of differences. In more detail 25 of these groups contained traces, which after the alignment were conformed by the model and 42 that were not validated at all.

The fact that 27.27% (600/2200) of the transactions needed alignment to be validated by the test behavior, shows us how different the expected from the actual behavior of a software can be.

7.6.2 Configurations Differences

One of the differences found by the tool can be seen in Figure 7.8. In this tree some of the differences between live and test data can be seen. The second trace has

multiple added and skipped states, specifically we have marked two skipped states that show that the authentication procedure between test and live is different. With red are the actions that are followed in the testing environment and the blue states in between, are the events that determine the authentication procedure for the live environment.

At the third trace three subsequent skipped states are identified. There are three states that exist in the test environment but not in the live.

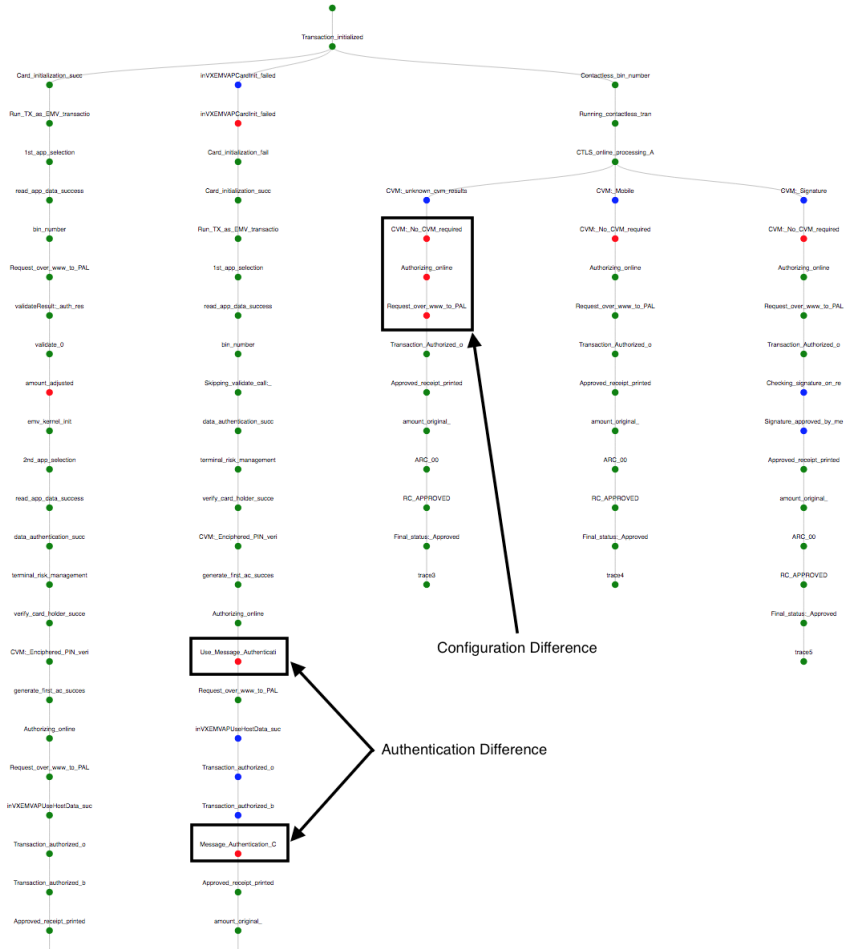


Figure 7.8: Configuration Differences

7.7 Discussion

From the performed experiments, multiple useful conclusions can be drawn. Algorithms Greedy and Best-First work well in identifying minor changes, like one or two

subsequently changed states. However their performance is not good for multiple and more complicated changes. On the other hand, the dynamic programming solves this issue, since it does not perform very well identifying small differences in the incoming sequences, but as the differences become more complicated the results drastically improve. The reason behind this behavior is that Greedy and Best-First focus on the neighboring nodes thus, for small differences they find the best candidate for alignment. In contrast, dynamic programming considers all possible transitions of the tree as possible alignments. In that way, the complicated mutations can be found, because the score is based on the alignment of the whole sequence, but for minor differences the algorithm might find an over complicated alignment with transitions that are out of context. That is also related to the depth optimization described in Section 6.2.3 for the Greedy algorithm, where the depth parameter has been introduced in order to improve the results. Nonetheless, in dynamic programming we couldn't limit the search in a similar way.

8

User Study

For the evaluation of the visualiztion method implemented for the log differencing tool, qualitative interviewing was used. As mentioned by Kvale in [81], research interviews are *attempts to understand the world from the subjects' point of view, to unfold the meaning of peoples' experiences, to uncover their lived world prior to scientific explanations.*

The reason why interviewing was chosen for the evaluation is to understand and evaluate the individual outcomes for each user, since our tool may have multiple functionalities. In that way, we will manage to see the differences between participants' experiences and outcomes. Last but not least, we aim to find where this tool can be mostly used, by interviewing people from various teams.

Based on the definition provided by Callor et al. [82] this interviewing aims to tier 3, namely understand and refining. By discussing with the participants, program satisfaction data and *lessons learned* can be found for the improvement of our tool and the determination of the future improvements. In addition, the program impact to its users needs to be identified. In order to achieve that, results are shown to the participants and they provide feedback on how useful the visualized information is. This is tier 5, based on the aforementioned work by [82].

8.1 Interview Design

The main goal of this interview is to understand how easily this tool can be used by the participants. There are some closed questions along with some more free questions, with more informal conversational style. The interview is split in three parts. In the first one, the goal is to get to know the participant, in the second we check its familiarity with log files. We necessarily need participants that use log files in their everyday work. After the second part, a real example of the implemented visualized is shown to the participant and some simple things are explained. More specifically, we show Figure 8.1 to the participants which represents the differences between live and test for a specific software version. The color mapping is briefly explained, by saying what each color is and then they have 1 minutes to play with

the visualization. Then the Tool Evaluation questions are asked. Finally there is some time for open conversation, ideas and feedback.

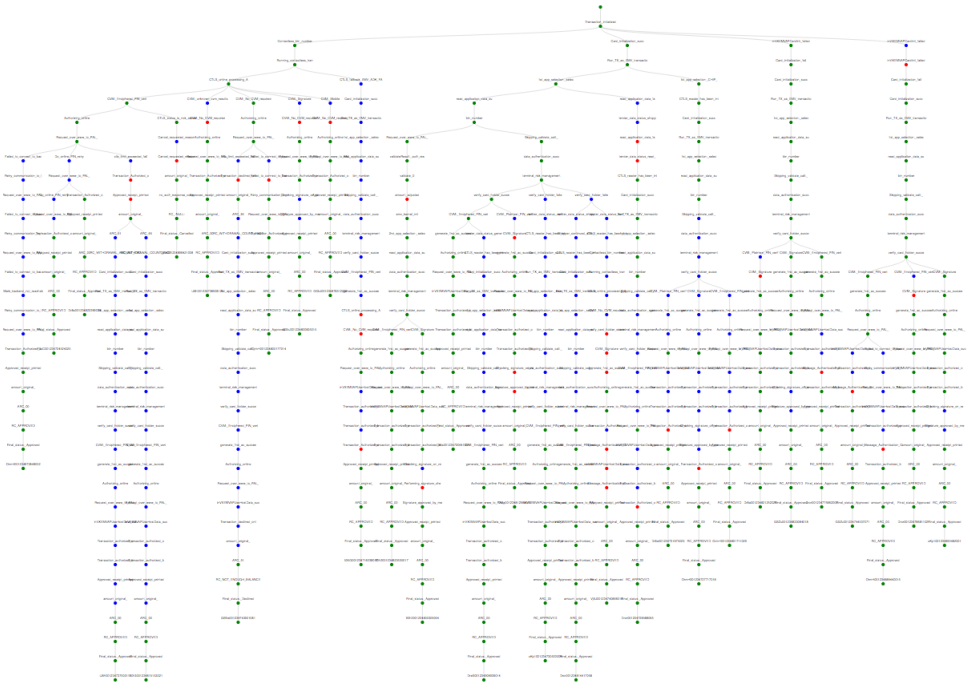


Figure 8.1: Differences between test and live shown to participants

8

8.2 Interview Questions

Personal Information

- Role and Team

Familiarity with Logs

- How often do you need to need information from log files?
- Do you ever have to compare log data?

Tool Evaluation

- Would you spend time to review all differences?
- Did you expect so many differences between live and test?
- Do you see anything noticeable in the graph?
- Is the color mapping helpful?

- Is the visualization easy?
- For which purpose do you think this tool would be helpful?

Feedback

- Free feedback and ideas

8.3 Results

For the user study, the questions presented in the previous Sections were asked to 11 people from Adyen, where in Figure 8.2 their roles in the company can be found. All the employees asked, were often working with log files, most of them had to retrieve some information from logs almost every day.

All the participants had a minute to *play* with the visualization tool. Most of them found things that looked interesting to what they already knew about the behavior of the system.

The visualized differences were a lot and none of the participants was willing to investigate each one of them. Only one person mentioned that if it was an one time procedure, he/she was willing to do it. All the participants proposed the use of some filters. As most of them indicated, the most frequent or the most rare differences are the ones that have to first be investigated.

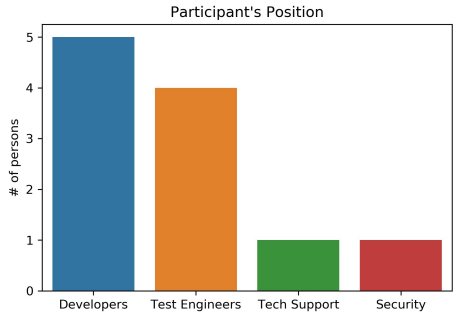
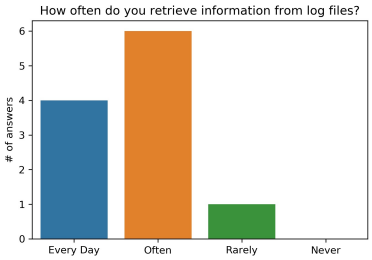


Figure 8.2: Role of participants at Adyen

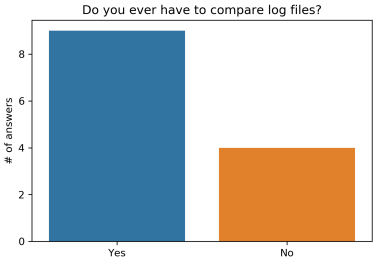
The majority of the interviewees did not expect so many differences between live and test, although there were some developers that were really happy that it is so clearly pointed out by this tool the insufficiency of the current test coverage.

The results of the question "For which purpose is this tool helpful?" are interesting. The participants were from different teams and had different expertise. Most of the back end developers and the automation engineers proposed the improvement of the *Test Coverage* as the main purpose of this tool. However front end developers, people from support and security believe that it can be used for *Documentation* purposes, to show how a transaction should be processed and compare it with anomalous transactions, but also for giving *Merchant Insights*, by for example comparing the volume of their transactions for two subsequent weeks.

The discussions with Adyen's employees helped us improve the tool but also gave us a good understanding of how useful it can be. Almost all the participants were really satisfied with this tool and believe it can have multiple uses.

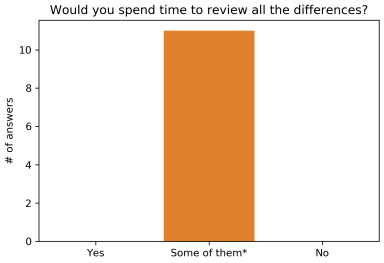


(a) How often do you retrieve information from log files.

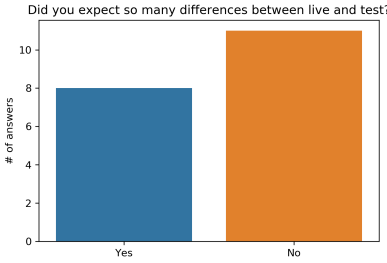


(b) Do you ever need to compare log data.

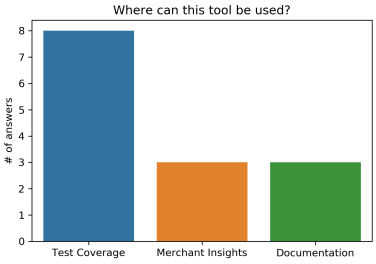
Figure 8.3: Questions about familiarity with logs.



(a) Would you spend time to review all differences.



(b) Did you expect so many differences.



(c) For which purpose is this tool helpful.

Figure 8.4: Questions about tool evaluation.

9

Conclusion & Final Remarks

The goal of this thesis is to develop a tool that will be able to differentiate log files, with the purpose to detect and visualize potential anomalies in the software execution. The main difficulty was to find the right methods to perform the comparison between the log data, since the requirements of the system entailed the handling of large amounts of data. In this research, various comparison algorithms were tried, some not suitable at all, some more than others (Chapter 7, 10). Additionally an algorithm from another field -bioinformatics- was modified to serve our purposes, which seem promising, especially for complicated patterns.

The comparison of log data does not seem to be a field which is thoroughly investigated, however the results prove that the behavior of the software many times is different than the expected. Just in the simplest case, comparing the behavior of the test environment with the live transactions showed that there are way more differences that most of the people expected (Figure 8.4.b). We showed that the test coverage can be improved by comparing test with live and learn from their differences (uncovered test cases were revealed). Furthermore the release monitoring procedure can be improved by comparing the logs of a stable software version with the new introduced release.

To conclude, we managed to create a tool, which compares the expected behavior of a system and the observed. Output of the comparison are the aligned traces grouped based on their differences. There is also the option to visualize the deviating traces using a data structure that makes the result understandable for the developer. Additional information about the deviating traces is provided in the visualization in order to investigate the difference further.

9.1 Reflection on Research Questions

At this point it would be useful to go back and reflect how this thesis answered the initially set research questions.

1. How can software's behavior comparison be performed for large amounts of data?

The answer to the first question is twofold, because in order to answer if there are

comparison methods for large amounts of data, first all the possible methods for any amount of data should be gathered. Thus we tried in this thesis to gather as many methods as possible to address the specific issue. These methods are presented in Chapter 3.7, where we realized that comparing two DFAs is not efficient for large amounts of data and some more general approaches have been proposed in Chapter 3. A very important outcome of this work, is the modification of the sequence alignment algorithms that are usually used in bioinformatics, for our problem. At the end we propose three different implementations, each one of them with district advantages and disadvantages.

2. What kind of differences can be identified?

The types of differences we aimed to find are described in Section 4.3. There are some simple and some more complicated patterns that the algorithm should be able to identify. However every difference will be represented in terms of added and skipped states are explained in the aforementioned section. The effectiveness of the algorithm to identify the differences is analyzed in Chapter 7. We proved that all the patterns can be found, however there are some, which seem to be more difficult than others. Also the three different algorithms we implemented do not perform that good at every difference. Nonetheless the performance is very promising since the problem we are dealing with is not trivial.

3. How can the differences of the software's behavior be visualized effectively?

This is a difficult question, since it is not easy to measure how good a visualization is. However we implemented a visualization using d3 that seem to be very helpful for the developers. To evaluate the impact the visualization does, a user study was conducted, where after discussions with more than 10 developers we concluded that it is actually a good way to visualize the differences between log files, but also some really clever improvements were proposed. Some of them were implemented, some are proposed as future work.

4. What kind of data are more suitable for the specification graph (to build the model)?

The data for building the specification model should have some characteristics, they should be as "complete" as possible, which means the functionality of the software should be covered and not only the most frequent flows. For that reason, the data from the robots (test transactions) seem to be the most appropriate, since they cover a wide range of possible flows and not only the happy ones (Chapter 7 for more details). Nonetheless it also depends on the comparison we are interested to perform. In some cases the data should just include specific flows with more detail, like when comparing two subsequent weeks of data for a specific merchant, there for the specification graph, it would help to have a model that would cover all the normal flows, but if it is not possible just the most frequent ones are necessary.

5. How does the model size affect the results?

This question is addressed in Chapter 7. We showed that the bigger the model, the more complex, the worse the results. In Chapter 5 the learning decision were explained that led to the models we are actually using later on.

9.2 Threats to Validity

As mentioned previously, the data constitute one of the most important factors that affect the results. It is important to compare data that have some common behavior and the detection of the deviating flows will make some sense.

- **Correctness of Models** That's a general problem not easily approached. A model can be evaluated in terms of its simplicity, fitness, generalization and precision. In this project, we did not deal with the correctness of our models and we made the assumption that when a difference is returned by the comparison algorithm, it is an actual difference and not a mistake of the learning process.
- **Tool's performance is specification model sensitive.** That is maybe something expected, that the more complex the model the worse the results of the sequence alignment will be. Models with more than 1000 states were used for the experiments and the results are good, but we also concluded that for smaller models the performance is even better. That also means, that for bigger models the performance is worse. What is the "optimal" size of a model is a topic that concerns the research community a lot, because for small models (more merges) information is lost, and for big models (many states) it becomes very complicated, without good generalization. Consequently the balance between these two should be found. However, with more data, inevitably bigger models will be created thus, the question that needs to be answered is how much data is enough?
- **Dynamic Programming heavily depends on the defined penalties.** As explained in Chapter 7, the scores play a very important role to the dynamic programming approach. For some scores the accuracy may be less than 20%. By the method trial and error, various penalties' values are tried and the ones with the best results are chosen. We also tried to improve our results by introducing dynamic scoring. Nonetheless the specific problem needs further investigation and is quite possible that the best possible value is not found. To conclude the influence the scores have to the performance of the algorithm, can be considered a limitation, since we do not know the way to tune the parameters effectively.
- **Unlabeled real data.** Since there are no labeled data in terms of what differences appear between the comparing logs, we had to mutate the dataset by ourselves. We tried to resemble the actual differences, however it would always be more accurate to evaluate it with real data that we know what differences they present.

9.3 Future Work

The nature of the data makes the tool produce too many differences, which means that the manual inspection is too time consuming. However it is proven that some differences are not meaningful and should be removed from the results or at least

filtered if the user chooses to. For example the configurations differences between test and live constitute a big part of the returned results and hinder the detection of important deviations, which could be bugs of the system or missing test cases. Consequently there could be further investigation on how these redundant differences could be filtered out. In addition to that a database with the non-anomalous differences could be created to avoid the aforementioned problem.

The log differencing tool between live and test data can be used to check the test coverage and create automatically test scenarios. With the pipeline presented in Figure 9.1 the live transactions that deviate from the expected behavior (the one indicated by test data) could automatically create a test case which would run at Adyen's testing framework and verify the correct or wrong execution according to the specifications. During the initial stages of this project a developer would be required, to manually annotate the flows that need to be tested and then automatically a test case will be created in order to achieve that. At some point the test coverage will be much better thus, every difference identified by the log differencing tool could be tested and annotated as anomaly or not for future improvement of the identification process.

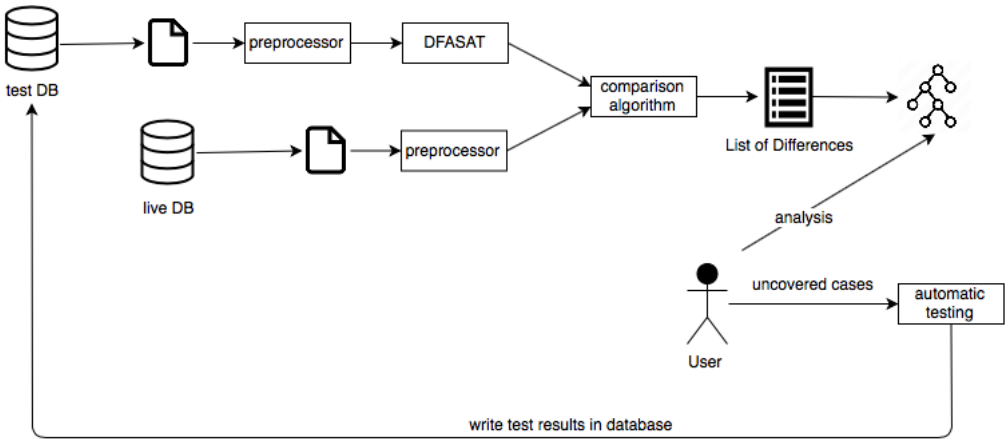


Figure 9.1: Automatic Testing using log differencing tool.

In Figure 9.2 a different scenario partly implemented by this project can be found. In this case, the goal is to compare the usual behavior of one merchant with the observed transactions. The comparison of one log line is very fast and using some optimizations and parallelism it is feasible to develop a system that will be able to check every observed transaction on the fly. The learning of the merchant's behavior is a time-consuming procedure, which however needs to be performed only once per week in order to keep the system up to date with the most recent data.

Finally based on the feedback provided during the interviews we propose the use of some filters to reduce the amount of the results in the visualization. More specifically, the transactions could be filtered based on the payment type, like contactless or

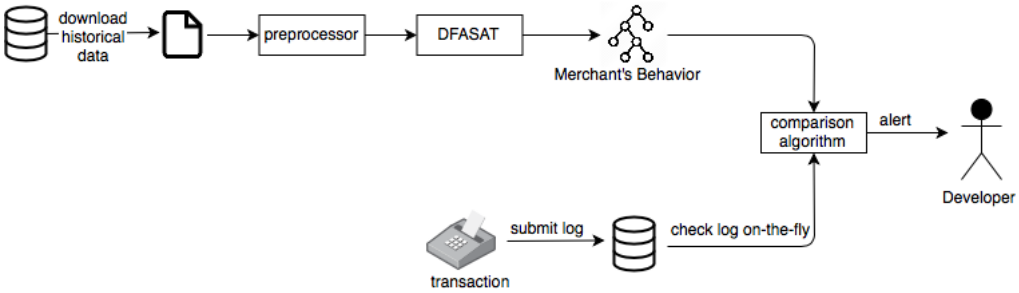


Figure 9.2: Alert system based on Merchant's expected behavior.

inserting the card or the final state. In Figure 9.3 an example of the filters that could be applied is presented.

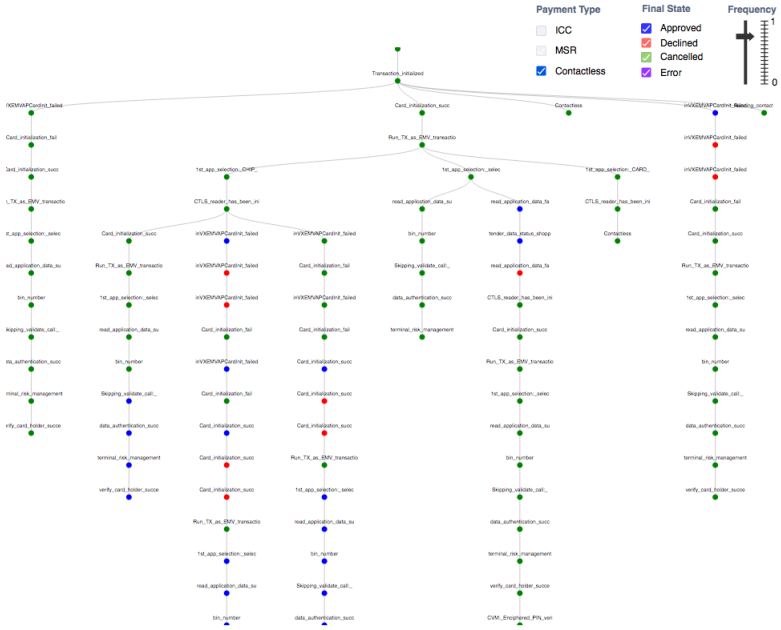


Figure 9.3: Proposed filters for the visualization.

10

Model Checkers

One of the most powerful features of model checkers, is that they can efficiently determine if two learned models are equivalent or not, and in case they are not, return a counter example. There are different algorithms to determine the equivalence of two models, some strict some more relaxed, as explained in Chapter 2. In this Chapter it is investigated, how model checker's equivalence checking can be used for the comparison of state machines.

The are two reasons, why two models are not equivalent, of course there is the obvious reason, that there is a difference that makes them at some point deviate, but there is also the possibility that one of them is learned correct. In that case, the models were supposed to be equivalent, but a flaw at the learning process produced an unexpected model. Just by the output of the equivalence checker, one counter-example, it is difficult to identify if there was an issue in the learning procedure or a difference between the models. The models used for comparison are supposed to have differences in our case thus, every difference returned by the model checker will be considered a difference and will be handled as one. This is an assumption we make, that the models are correctly learned and all the counter-examples from the model checkers, are actual differences between the two models. Future work should reconsider that and investigate further the correctness of the learning procedure.

Using model checkers there is no other way of identifying differences between two state machines, other than applying active learning. Suppose a model checker returns a difference between the models as a counterexample, there is no other way to avoid this difference show up in the next iteration, instead of relearning one of the two models in order to satisfy this property. Then, the relearned model and the unmodified one have to be fed to the model checker again, to check if they are equivalent. Of course it is necessary to keep track of all the returned counterexamples in order to detect all the differences.

10.1 Preprocessing

The chosen model checker was mcrl2, because it has an active community still working on it and it is well documented. In addition to these, it also has many

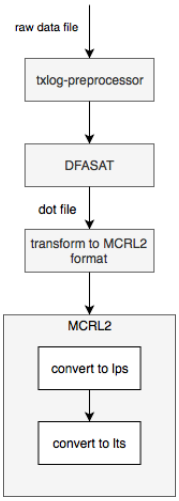


Figure 10.1: Preprocessing for Model Checker's approach

features and a big variety of algorithms.

The required flow to use the unstructured log files with the model checkers in shown in Figure 10.1. The first thing that enters the flow is the unstructured dataset, in our case the log files. They go through the *tx-log preprocessor*, which as mentioned above, structures the information included and outputs a log file with the right structure to get accepted by the DFASAT. DFASAT is our learning tool for the state machines.

DFASAT outputs .dot files, which can not be parsed by the mcrl2. For that purpose a python program was developed to read the .dot files and translate it into the mcrl2 language [83]. How the dot file looks like can be found in 10.2 and how its corresponding mcrl2 format is, in 10.3.

```
digraph DFA  I -> s0;
s0 -> s1 [label="transition_1"];
s0 -> s2 [label="transition_2"];
s2 -> s5 [label="transition_3"];
s1 -> s3 [label="transition_4"];
s3 -> s4 [label="transition_5"];
```

Figure 10.2: Dot file format

```
act transition_1, transition_2, transition_3, transition_4, transition_5;
```

```
    proc root=s0.transition_1 + s1. transition_2;
        s2 = s5.transition_3;
        s1 = s3.transition_4;
        s3 = s4.transition_5;
        s4 = delta;
init root;
```

Figure 10.3: MCRL2 file format

However that is not enough in order to use the equivalence checking, the mcrl2 file has to be translates from mCRL2 process specification to a linear process specification (LPS), which can be done with the command *mcrl22lps*. There are three different linearisation methods for that purpose, however we just use the regular one, which is the restricted Greibach Normal Form. This is not the most suitable method if the graphs contain loops, because if some process has an infinite number of control states, the tool will attempt to generate all of them, causing it to run out of memory

The final step of the preprocessing is necessary to visualize or compare the graph with another one, the transformation of the lps to lts using the *lps2lts* command, which generates a labelled transition system from a linear process. The whole pipeline for the transformation of the raw log file to an mcrl2 format can be found in figure 10.1.

At that point the comparison can be performed if the two files have the same format using the command *ltscompare*.

10.2 Comparison Pipeline

The chosen pipeline for the comparison of two state machined using model checkers can be found in 10.4. The active learning framework has been used for this approach, since at each iteration the output of the model checker should be used for the relearning of one of the input models. Each counter example is a difference between the two behaviors and then it is resolved in order to identify the next one. Consequently the procedure is like modifying the traces of model 2 in example 10.4 to look like the ones in model 1.

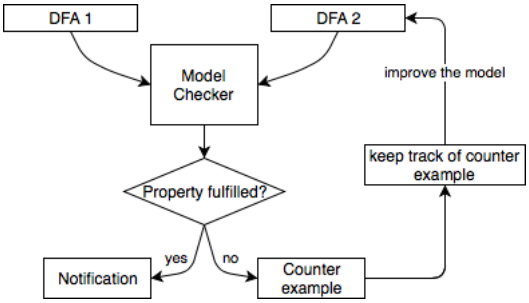


Figure 10.4: Model Checker Active Learning Framework

10.3 Discussion

The implementation using Model Checkers requires multiple iterations to identify one difference at each step and multiple repetitions of the learning process, fixing something small at the time. Working with small graphs, with not many differences makes this approach feasible. However in our case, where graphs have hundreds of nodes (at the best case) and lots of differences this approach takes a really long time just to change the first difference and relearn the model. Nonetheless at the beginning it was not possible to see that clear; throughout this thesis' procedure more knowledge about the domain was gained and after a while and some tries it became clear that this method can not be used for real data and real world specifications.

The main disadvantage of this approach is the fact that every iteration takes a lot of time. In addition to that the comparisons that are usually performed are between models with huge structural differences. That means, many iterations are required to make the two graphs equivalent. For the comparison of the graph presented in Figure 6.1, which represents the test behavior, with the live behavior learned only by 2200 transactions more than 5.000 iterations were necessary. Actually the model checker did not find the two graphs equivalent after that amount of iterations, thus the execution was stopped.

Another disadvantage of this method, is that we can not prove the completeness of the returned differences. The fact that at each iteration, one of the models is relearned, to resolve one of their differences, might resolve a difference that is not found yet. This is another issue that requires further investigation, but it is not performed in the current work, since it was decided to not continue further with this approach, because it does not seem to have much potential.

Bibliography

References

[1] Martin Hiller, Arshad Jhumka, and Neeraj Suri. Propane: an environment for examining the propagation of errors in software. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 81–85. ACM, 2002.

[2] Christina Warrender, Stephanie Forrest, and Barak A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, 1999.

[3] T-TY Lin and Daniel P Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *IEEE Transactions on Reliability*, 39(4):419–432, 1990.

[4] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *International conference on Data Mining (full paper)*. IEEE, December 2009.

[5] Leonardo Mariani and Fabrizio Pastore. Automated identification of failure causes in system logs. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, ISSRE '08, pages 117–126, Washington, DC, USA, 2008. IEEE Computer Society.

[6] R. Wieman. *What Does Passive Learning Bring To Adyen?* Master's thesis, Delft University of Technology, 2017.

[7] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection for discrete sequences: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 24(5):823–839, 2010.

[8] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[9] Francesco Calzolai, Rocco De Nicola, Michele Loreti, and Francesco Tiezzi. *TAPAs: A Tool for the Analysis of Process Algebras*, pages 54–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

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

[11] Sjoerd Cranen, Jan Friso Groote, Jeroen JA Keiren, Frank PM Stappers, Erik P De Vink, Wieger Wesselink, and Tim AC Willemse. An overview of the mcrl2 toolset and its recent advances. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 199–213. Springer, 2013.

[12] David W Mount. *Bioinformatics: sequence and genome analysis. 2nd*, volume 692. Cold Spring Harbor, NY: Cold Spring Harbor Laboratory Press. xii, 2004.

[13] Eric Sven Ristad and Peter N Yianilos. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):522–532, 1998.

[14] Richard Bellman. The theory of dynamic programming. *Bull. Amer. Math. Soc.*, 60(6):503–515, 11 1954.

[15] Peter Evers. *Finding relevant errors in massive payment log data.* Master's thesis, Delft University of Technology, 2017.

[16] Quincy Bakker. *Designing, developing and evaluating a log management tool for developers to improve monitoring practices.* Master's thesis, University of Utrecht, 2018.

[17] Joop Aue. *An exploratory study on faults in web api integration.* Master's thesis, Delft University of Technology, 2017.

[18] Peter Evers. *A large-scale evaluation of tracing back log data to its origin with static analysis.* Master's thesis, Delft University of Technology, 2018.

[19] F. Farahmandi and P. Mishra. Fsm anomaly detection using formal analysis. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 313–320, Nov 2017.

[20] T. Ohmann, K. Thai, I. Beschastnikh, and Y. Brun. Mining precise performance-aware behavioral models from existing instrumentation. In *36th International Conference on Software Engineering, ICSE Companion 2014 - Proceedings*, pages 484–487, 2014. cited By 8.

[21] Alan W Biermann and Jerome A Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE transactions on Computers*, 100(6):592–597, 1972.

[22] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 267–277. ACM, 2011.

[23] Maayan Goldstein, Danny Raz, and Itai Segall. Experience report: Log-based behavioral differencing. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 282–293. IEEE, 2017.

[24] Robert S Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[25] Hen Amar, Lingfeng Bao, Nimrod Busany, David Lo, and Shahar Maoz. Using finite-state models for log differencing. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 49–59. ACM, 2018.

[26] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the futurebus+ cache coherence protocol. volume 6, pages 217–232, Mar 1995.

[27] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, pages 46–51, June 1990.

[28] Yu-Tong He and Ryszard Janicki. Verifying protocols by model checking: A case study of the wireless application protocol and the model checker spin. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '04, pages 174–188. IBM Press, 2004.

[29] David Basin, Cas Cremers, and Catherine Meadows. *Model Checking Security Protocols*, pages 727–762. Springer International Publishing, Cham, 2018.

[30] George C. Necula. Translation validation for an optimizing compiler. *SIGPLAN Not.*, 35(5):83–94, May 2000.

[31] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. *SIGPLAN Not.*, 39(1):1–13, January 2004.

[32] Aditya Kanade, Amitabha Sanyal, and Uday Khedker. A pvs based framework for validating compiler optimizations. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '06, pages 108–117, Washington, DC, USA, 2006. IEEE Computer Society.

[33] Mathijs Schuts, Jozef Hooman, and Frits Vaandrager. Refactoring of legacy software using model learning and equivalence checking: An industrial experience report. In *Proceedings of the 12th International Conference on Integrated Formal Methods - Volume 9681*, IFM 2016, pages 311–325, Berlin, Heidelberg, 2016. Springer-Verlag.

[34] E.Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241 – 266, 1982.

[35] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: A practical approach. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 117–126, New York, NY, USA, 1983. ACM.

[36] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142 – 170, 1992.

[37] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, Philippe Schnoebelen, and Pierre Mckenzie. *SMV — Symbolic Model Checking*, pages 131–138. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[38] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, pages 317–320, New York, NY, USA, 1999. ACM.

[39] Jerry Banks, II Carson, Barry L Nelson, David M Nicol, et al. *Discrete-event system simulation*. Pearson, 2005.

[40] Arya Adriansyah, Boudewijn F van Dongen, and Wil MP van der Aalst. Conformance checking using cost-based fitness analysis. In *2011 IEEE 15th International Enterprise Distributed Object Computing Conference*, pages 55–64. IEEE, 2011.

[41] Anne Rozinat and Wil MP Van der Aalst. Conformance checking of processes based on monitoring real behavior. *Information Systems*, 33(1):64–95, 2008.

[42] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[43] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[44] Walter M Fitch and Temple F Smith. Optimal sequence alignments. *Proceedings of the National Academy of Sciences*, 80(5):1382–1386, 1983.

[45] Eagu Kim and John Kececioglu. Learning scoring schemes for sequence alignment from partial examples. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 5(4):546–556, 2008.

[46] Dan Gusfield and Paul Stelling. [28] parametric and inverse-parametric sequence alignment with xparal. In *Methods in enzymology*, volume 266, pages 481–494. Elsevier, 1996.

[47] Julie D Thompson, Desmond G Higgins, and Toby J Gibson. Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic acids research*, 22(22):4673–4680, 1994.

[48] Neil Walkinshaw and Kirill Bogdanov. Automated comparison of state-based software models in terms of their language and structure. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(2):13, 2013.

[49] Fides Aarts, Harco Kuppens, Jan Tretmans, Frits Vaandrager, and Sicco Verwer. Learning and testing the bounded retransmission protocol. In *International Conference on Grammatical Inference*, pages 4–18, 2012.

[50] Kevin J Lang, Barak A Pearlmutter, and Rodney A Price. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In *International Colloquium on Grammatical Inference*, pages 1–12. Springer, 1998.

[51] Neil Walkinshaw, Kirill Bogdanov, and Ken Johnson. Evaluation and comparison of inferred regular grammars. In *International Colloquium on Grammatical Inference*, pages 252–265. Springer, 2008.

[52] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. *Black Box Checking*, pages 225–240. Springer US, Boston, MA, 1999.

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

[54] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.

[55] David Combe, Colin de la Higuera, and Jean-Christophe Janodet. Zulu: An interactive learning competition. In Anssi Yli-Jyrä, András Kornai, Jacques Sakarovitch, and Bruce Watson, editors, *Finite-State Methods and Natural Language Processing*, pages 139–146, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[56] Kirill Bogdanov and Neil Walkinshaw. Computing the structural difference between state-based models. In *2009 16th Working Conference on Reverse Engineering*, pages 177–186. IEEE, 2009.

[57] Neil Walkinshaw and Kirill Bogdanov. Comparing software behavior models. Technical report, Technical report: CS-08-16, The University of Sheffield, Sheffield, UK, 2008.

[58] Udo Kelter and Maik Schmidt. Comparing state machines. In *Proceedings of the 2008 international workshop on Comparison and versioning of software models*, pages 1–6. ACM, 2008.

[59] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. Matching and merging of statecharts specifications. In *Proceedings of the 29th international conference on Software Engineering*, pages 54–64. IEEE Computer Society, 2007.

[60] Jochen Quante and Rainer Koschke. Dynamic protocol recovery. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 219–228. IEEE, 2007.

[61] Rick Wieman, Maurício Finavaro Aniche, Willem Lobbezoo, Sicco Verwer, and Arie van Deursen. An experience report on applying passive learning in a large-scale payment company. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 564–573. IEEE, 2017.

[62] Samuel C Hsieh. Product construction of finite-state machines. In *Proc. of the World Congress on Engineering and Computer Science*, pages 141–143. Citeseer, 2010.

[63] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, pages 365–373, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[64] Giampiero Cabodi, Paolo Camurati, Fulvio Corno, Paolo Prinetto, and Matteo Sonza Reorda. The general product machine: a new model for symbolic fsm traversal. *Formal Methods in System Design*, 12(3):267–289, Apr 1998.

[65] Michael Sipser. *Introduction to the theory of computation.* PWS Publishing Company, 1997.

[66] Martin Tappler, Bernhard K Aichernig, and Roderick Bloem. Model-based testing iot communication via active automata learning. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 276–287. IEEE, 2017.

[67] David Gries. Describing an algorithm by hopcroft. *Acta Informatica*, 2(2):97–109, Jun 1973.

[68] John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

[69] Timo Knuutila. Re-describing an algorithm by hopcroft. *Theoretical Computer Science*, 250(1):333 – 363, 2001.

[70] Giuseppe De Ruvo, Antonella Santone, and Domenico Raucci. Powerful equivalence checking in the bank supply process. In *Services (SERVICES), 2014 IEEE World Congress on*, pages 87–94. IEEE, 2014.

[71] Nicoletta De Francesco, Giuseppe Lettieri, Antonella Santone, and Gigliola Vaglini. Heuristic search for equivalence checking. *Software & Systems Modeling*, 15(2):513–530, 2016.

[72] John Abela, François Coste, and Sandro Spina. Mutually compatible and incompatible merges for the search of the smallest consistent dfa. In *International Colloquium on Grammatical Inference*, pages 28–39. Springer, 2004.

[73] SE Verwer, MM De Weerdt, and Cees Witteveen. An algorithm for learning real-time automata. In *Benelearn 2007: Proceedings of the Annual Machine Learning Conference of Belgium and the Netherlands, Amsterdam, The Netherlands, 14-15 May 2007*, 2007.

[74] Marijn JH Heule and Sicco Verwer. Software model synthesis using satisfiability solvers. *Empirical Software Engineering*, 18(4):825–856, 2013.

[75] Sicco Verwer and Christian A Hammerschmidt. flexfringe: a passive automaton learning package. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 638–642. IEEE, 2017.

[76] Christian Hammerschmidt. *Learning Finite Automata via Flexible State-Merging and Applications in Networking*. PhD thesis, 2017.

[77] K Lang. Evidence driven state merging with search. *Rapport technique TR98–139, NECI*, 31, 1998.

[78] Neil Walkinshaw, Bernard Lambeau, Christophe Damas, Kirill Bogdanov, and Pierre Dupont. Stamina: a competition to encourage the development and assessment of software model inference techniques. *Empirical software engineering*, 18(4):791–824, 2013.

[79] Franck Thollard, Pierre Dupont, Colin de la Higuera, et al. Probabilistic dfa inference using kullback-leibler divergence and minimality. In *ICML*, pages 975–982, 2000.

[80] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. A likelihood-ratio test for identifying probabilistic deterministic real-time automata from positive data. In *International Colloquium on Grammatical Inference*, pages 203–216. Springer, 2010.

[81] Jeanne M Plas, Steinar Kvale, and STEINAR AUTOR KVALE. *Interviews: An introduction to qualitative research interviewing*. Sage, 1996.

[82] S Callor, SC Betts, R Carter, and M Marczak. State strengthening evaluation guide. *Tucson, AZ: USDA/CSREES & University of Arizona*, 1997.

[83] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck Van Weerdenburg. The formal specification language mcrl2. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.