What Does Passive Learning Bring To Adyen?

A Case Study

R. Wieman

Master Thesis

What Does Passive Learning Bring To Adyen?

A Case Study

by

R. Wieman

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on March 29, 2017 at 14:00.

Student number:	4149130	
Project duration:	May 1, 2016 – February 28, 2017	
Thesis committee:	Prof.dr. A. van Deursen	TU Delft
	Dr.ir. S.E. Verwer	TU Delft, supervisor
	Dr. A.E. Zaidman	TU Delft
	Dr. M.F. Aniche	TU Delft
	W.M. Lobbezoo	Adyen B.V.

“A picture is worth a thousand words.”

Abstract

Analyzing large numbers of log entries can be challenging, especially when there are many short log entries that each describe only one execution of the system, either successful or unsuccessful. How can one determine whether the system is working correctly, based on these logs? The logs that are of interest (e.g., log entries pointing towards some anomaly in the system) may be hidden between all the logs that are of less interest. Luckily, there are so-called passive learning tools that infer a (graph) model from such set of logs, which allows the user to oversee all paths that were taken in the system.

In this thesis, we discuss the opportunities for passive learning for Adyen, a large-scale payment company. We compare three different open source passive learning tools (namely Synoptic, InvariMint, and DFASAT) in terms of runtime performance and output complexity, and show that all tools struggle with an increasing input size. We also share the results of a survey we conducted under developers to identify their perceptions, and for which purpose(s) they would use such tools. Furthermore, we provide six examples of different types of analyses that are possible with passive learning (such as finding bugs, comparing within a context, and analyzing timings), and that are useful for the company. We include a short guide on how to adopt passive learning, and what we had to change in one of the tools to make it so useful. Finally, we show how a graph difference tool can help to compare different graphs, for example over different time intervals. This tool highlights differences in both structure and frequencies. Altogether, this shows what passive learning brings to Adyen.

Preface

This thesis is the result of a graduation project of nine months for the degree of Master of Science in Embedded Systems. Even though this work might not be that typical for an Embedded Systems masters student, I learned a lot about payments, point of sale (POS), logging, (acceptance) testing, data analysis, data security, and probably much more. This report only partially captures all the gained knowledge.

All prefaces I have seen so far used a significant part for thanking people. Even though I generally do not like to be thanked for stuff myself, I really want to thank numerous people that helped me throughout the project. Simply because without them, this report would not have laid before you.

First of all, I want to thank Willem Lobbezoo, for his guidance throughout this project. All your ideas and discussions (even though you often had to repeat yourself because I did not listen) made the project go towards this final document. Thank you for always keeping an eye on the horizon, to make sure that we would finish on time. I also learned a lot from your analyses, and your view on the performance of the product.

Next, I want to thank Sicco Verwer, Arie van Deursen, Andy Zaidman, and Maurício Aniche in particular, for their guidance from an academic perspective. Thanks to you, this document has the academic value it has. I truly learned a lot about writing down the findings in a clear and structured way. Thanks for comforting me every time I thought that my work was not interesting enough to write about; that must have been annoying.

None of this project would have been possible if Maikel and his team would not have reached out to me on the Delftse Bedrijvendagen, so I would really like to thank them for the great opportunity. Furthermore, I want to thank Eduard for giving me a place in his testing team (and at the foosball table), and Alexandros and Victor for their patience while listening to my (initial) thoughts. I also want to thank Huub for being a great buddy, and for the fresh ideas during lunch. In addition, I want to thank Peter for the extensive review of this document. Lastly, I want to thank all the developers that filled in our surveys, only to win a lousy gift card. Without your responses, this report would lack some serious grounds. Probably I could go on and name dozens of colleagues that supported me in one way or another (ranging from having a discussion over a cup of coffee to providing me with fresh insights or data), but I am too afraid that I will miss some names; if this addresses you: Thank you!

Last but not least, I want to thank my parents for their unconditional support during a decade of studies, and for keeping me motivated (or at least the attempts to do that). I also want to thank the rest of my family for their support and their interest in my work; in particular, I want to thank my brother and sister for listening to my endless conversations at dinner time. Furthermore, I want to thank all my friends that had to listen to the same conversations over a beer. Without your support and distractions, this would definitely not have been possible. Thanks, all of you!

Rick Wieman
Bodegraven, The Netherlands
March 2017

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

1 Introduction

Analyzing large amounts of log data can be challenging, especially when there are many short logs that each describe one execution of a system, either successful or unsuccessful. How can one determine whether the system is working correctly, based on these logs? The log entries that are of interest (e.g., log entries pointing towards some anomaly in the system) may be hidden between all the logs that are of less interest. Even if one finds a log entry pointing towards an anomaly, it might still be unclear how often this anomaly occurs, and whether there are related problems. Therefore, we argue that analyzing those single log files by hand is simply impossible in large systems. Luckily, tools exist that infer a *graph model* from log data, to help with visualizing the behavior of the system hidden in these files. These so-called *passive learning* tools allow the user to observe all logged paths that were taken by the system. This can help, for instance, identify whether the behavior of a system in production is correct, and whether the behavior in a test environment matches what happens in production (in an ideal world, the test environment represents the production environment with total fidelity).

We conducted a research project over nine months at a large company in the payment industry: Adyen. Adyen is a technology company that provides businesses with a single solution to accept payments anywhere in the world. The only provider of a modern end-to-end infrastructure connecting directly to Visa, MasterCard, and 250 other payment methods, Adyen delivers frictionless payments across online, mobile, and in-store channels. With offices all around the world, Adyen serves more than 4,500 businesses, including 7 of the 10 largest U.S. Internet companies. Customers include Facebook, Uber, Airbnb, Netflix, Spotify, Dropbox, Evernote, Booking.com, Vodafone, Mango, Crocs, O'Neill, SoundCloud, KLM and JustFab. Adyen processed a total transaction amount of roughly $50 billion in 2015 [3].

Each payment produces log entries in multiple different systems. Clearly, to process this total value of consumer transactions, this results in a huge amount of log data. We exclusively focus on logs from Adyen's point of sale (POS) solution, which is developed in-house in Amsterdam, The Netherlands. In a nutshell, the solution consists of an embedded device (hardware manufactured by another vendor) used in-store by merchants to safely collect the shopper's credit card and to perform the transaction together with the credit card scheme. The logs from these devices indicate what happened on the device during a transaction; they are submitted to Adyen's servers after the transaction is completed. In general, those logs consist of 15 to 25 lines that all contain a timestamp and an event. Examples of information that developers usually extract from these logs are 'why a transaction was refused for a certain merchant', and 'why some merchant performs many cancellations in a day'.

In this thesis, we present the lessons learned from applying passive learning in this large-scale system. We show how different passive learning tools perform when dealing with extensive log data, in terms of runtime performance and output complexity. We show how developers would use these tools for their day job, based on the results from our survey. Furthermore, we show some examples of issues we identified using passive learning, to illustrate the potential of passive learning. As a first step towards a monitoring solution based on passive learning, we present an algorithm to compare graphs originating in passive learning tools, which highlights the differences between two graphs. Altogether, we show the value of passive learning for Adyen.

1.1. Research Questions

The project focuses on applying passive learning in a large-scale payment company. It is centered around the main question of this thesis:

What does passive learning bring to Adyen?

To guide the project, we identified the following four research questions, which we will discuss and answer throughout the thesis. Ultimately, they help to answer this central question.

RQ1: How do different passive learning tools behave when dealing with extensive log data?

To get a feeling for the techniques and tools available for passive learning, we select three existing open source tools, and see how they perform on real data. We analyze the execution times (i.e., how long does it take to get a result), and the complexity of the produced graphs.

RQ2: How do developers perceive passive learning tools and techniques?

To get a better understanding for the possible opportunities for passive learning, we ask developers for their perceptions of the outcome of the selected passive learning tools. Additionally, we ask for which purpose(s) they would use these tools.

RQ3: What types of anomalies do passive learning tools identify in a system of our industry partner?

For this research question, we investigate the different types of anomalies that passive learning can find, and how they would be visible in the graphs. We use an iterative approach, where we meet a company expert on a regular basis to discuss the results, and to come up with fresh ideas regarding the analyses. We again ask the developers which of those results they deem the most useful. Subsequenty, we identify the steps that need to be taken in order to adopt passive learning in an industrial environment.

RQ4: Does a graph difference tool help to identify possible anomalies at our industry partner?

Ultimately, we would want to integrate passive learning into some analysis system that notifies developers on a regular basis of behavioral changes. To achieve this, comparing graphs appears to be an essential step, thus we present an algorithm to compare graphs, and show some examples (based on both synthetic data and real data) to illustrate how this is useful.

1.2. Structure of the Thesis

The thesis is structured as follows: first, we provide more information on Adyen and some implementation details of the systems we analyze in chapter 2. Then, in chapter 3, we introduce the passive learning tools we are using, show how they (do not) scale on our type of data (RQ1), and show that developers think these tools are the most useful for finding bugs (RQ2). To make the results more concrete, we provide multiple examples of what we found using the tools in chapter 4 (RQ3). In chapter 5, we provide an algorithm for comparing graphs, which can be used to highlight the differences, and we illustrate the algorithm with some examples (RQ4). We conclude the thesis in chapter 6.

2 About Our Industry Partner: Adyen

"On my first day, I thought: there is a credit card, the shopper inputs the PIN, we charge. How hard can it be?"
— W.M. Lobbezoo

A Payment Service Provider (PSP) is a party that offers payment services to retail merchants. Payment services basically consist of transferring money from a *shopper's* bank account to a *merchant's* bank account. This money transfer can be done based on a variety of payment instruments, like credit cards, debit cards, direct debits, et cetera. The retail world delivers its core business (selling goods and services) through a variety of sales channels. The main sales channels are webshops and physical shops, commonly referred to as respectively eCommerce (eCom) and point of sale (POS). Adyen is an omni-channel PSP that supports all retail sales channels in an integrated way.

All channels connect to the same (online) platform. This project will exclusively focus on the POS channel. A typical POS setup consists of a cash register operated by a *cashier* (or attendant) and a payment terminal used by shoppers to present their cards and for instance their personal identification number (PIN). The cash register is commonly referred to as POS and the payment terminal as PIN Entry Device (PED).

The POS initiates a *tender* (a transaction) by communicating with the PED through a library (provided by Adyen). The PED (running application software provided by Adyen) subsequently interacts with the shopper and requests a payment authorization from the platform. The POS can only communicate with the PED through the library (there is no direct communication); the PED acts as a black box to the POS device. It is also possible to use the PED without a POS; the cashier then enters the amount directly into the PED.

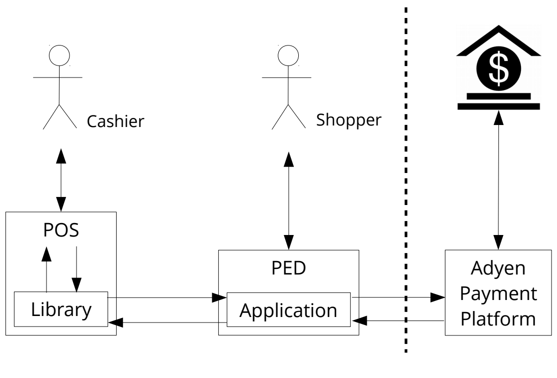


Figure 2.1: Schematic overview of the POS solution. The cashier operates some vendor POS system, which interfaces with the Adyen POS library. The shopper interacts with the PED, which communicates with both the library and the platform. The platform then communicates with the banks, but this is beyond the scope of this project.

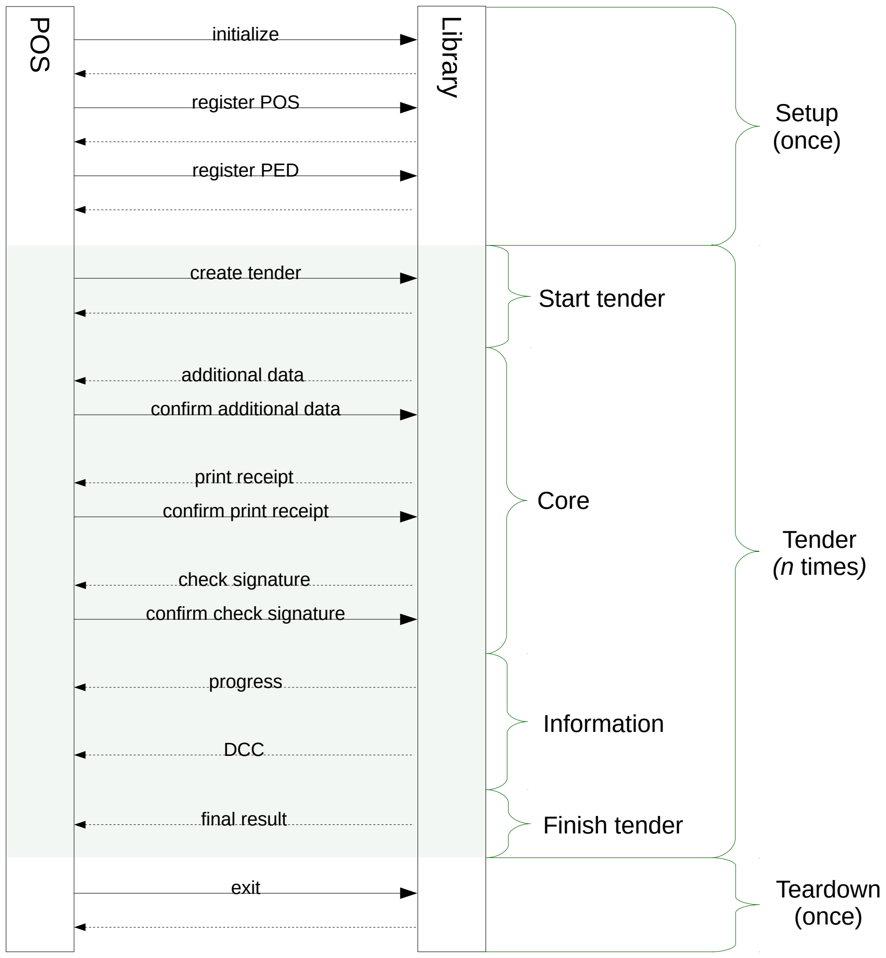


Figure 2.2: POS library calls and callbacks. Solid arrows (left-to-right) indicate calls, dashed arrows (right-to-left) indicate callbacks.

Figure 2.1 shows a schematic overview of the POS solution and its actors. The left side of the dashed line shows the situation inside some physical store, whereas the right side shows whatever happens outside the store and beyond. We only focus on the left side, the right side is beyond the scope of the project. In this chapter, we provide some background information on both the POS library and the PED, how they communicate, how they are tested, and the *transaction logs* the PED produces.

2.1. POS Library

The purpose of the POS library is to provide merchants an easy integration with the POS solution. It consists of a set of asynchronous functions, meaning that the POS can make a request (i.e., call a library function) after which a callback will be provided once that request has been completed. This interface is clearly documented in a developer manual [11].

Function calls can be considered as inputs and callbacks (in addition to return values) can be considered as outputs. The POS can only make one request at the time, thus it should in all cases wait for the callback before making any new requests. In some cases, a callback requires an explicit confirmation from the POS. The library acts to a certain extent as a proxy for the PED; it forwards in essence the calls to the PED and provides the responses back to the POS. Figure 2.2 shows the complete flow for processing one tender, although the highlighted part (Tender) can be repeated as often as desired.

The POS needs to initialize the library first. Similarly, after all payments have been processed (e.g., by the end of the day) and the POS shuts down, the library should be exited as well. After the initialization, the POS can register itself at the platform for authentication (and authorization) purposes. Once that has been completed

Figure 2.3: Example of a PIN Entry Device (PED).

successfully, the POS can register a PED at the platform. The registration process for POS and PED is also referred to as *boarding*.

After the boarding process, the library is ready to accept tenders. The POS can either create a tender or cancel a tender (using the reference of the tender from the creation callback). Callbacks are used to provide the POS with information about the tender, and to request further interactions (such as printing a receipt). Most callbacks require an explicit confirmation before the tender continues. Note that this example shows all possible callbacks for a card tender, but that in practice a varying subset of those callbacks is being used for a regular tender. This also depends on the configuration, which goes beyond the scope of this project.

The **additional data callback** provides the POS with additional information from the platform related to the shopper, such as loyalty information. This can then be used to, for example, determine a new amount (if some discount applies to this payment). The callback should always be confirmed by the POS, whereby the confirmation can contain (for instance) this new amount.
The **print receipt callback** is fired once a receipt is available for printing. For legal purposes, printing receipts is required and should therefore be confirmed by the POS as well.
The **check signature callback** is used to request confirmation of the signature of the shopper by the cashier, if a signature is required for the payment. It is evident that this callback should be confirmed as well.
The **progress callback** is used to inform the POS about the progress of the tender. This is merely for information purposes.
The **Dynamic Currency Conversion (DCC) callback** is fired if the shopper was offered DCC on the PED. DCC allows the shopper to choose whether he wants to pay in his own currency or in the merchants currency. This callback is only used to inform the POS about the fact that this has been offered, the actual outcome is provided through a progress callback.
The **final result callback** contains the final state of the tender, that is either approved, declined, cancelled, or error.

2.2. PIN Entry Device (PED)

The PED (an example of which is shown in Figure 2.3) allows both physical interaction (human interface) and digital interaction (software interface). The physical interaction is mostly related to getting inputs from and providing feedback to the shopper. For inputs, it has buttons (that can be grouped into *data* buttons and *control* buttons), an Integrated Circuit Card (ICC) reader, a magnetic swipe reader (MSR), a contactless module and sometimes even a touch screen. For outputs, it usually has a screen, a small speaker (for beeping), and sometimes even a printer. The digital interaction is twofold: on the one hand, it communicates with the platform, and on the other hand it communicates with the POS library.

The platform gets contacted multiple times during a tender. The first request usually is an optional validate request, which prepares the tender and gathers some information based on the inserted card (for example loyalty information, bank information, et cetera). The second request is an authorize request, which tries to reserve the amount from the shoppers bank account. The third request finishes the tender, by either issuing a capture (i.e., transfer the reserved amount to the merchants bank account) or a cancel (i.e., do not transfer the money). Furthermore, a brief log file is stored when the tender is finished. The details of the communication with the platform and the underlying systems are beyond the scope of this project.

At the moment, Adyen supports two different terminal operating systems: eVo and V/OS. Which operating system is used, depends on the terminal hardware model. Although they aim at minimizing differences between the two types, the implementations differ at multiple points, and the log lines differ significantly. The exact details on the differences are beyond the scope of the project.

The POS library is to some extent a proxy for communication with the PED. As a result, the communication between the PED and the POS library is very similar to the communication between the POS and the library, which we will not repeat here. However, we do provide a high-level idea of the possible states of the state-machine that is running in the PED in the first subsection, and a high-level description of the protocol that ICC transactions follow in the second subsection.

2.2.1. States

The following states are related to starting the tender, finishing the tender and the core that is in between, as depicted earlier in Figure 2.2:

- Start

  The PED starts in an `INITIAL` state, after which it goes into `TENDER_CREATED` as soon as the tender starts. There are now three options: the shopper pays either by dipping, by swiping or via manual key entry (MKE). Dipping the card will initiate a complex process, described by the Europay, MasterCard and Visa (EMV) protocol. This process is documented in four books [16–19], and we will not discuss it here.

- Enrichment

  Adyen uses the card details to fetch additional data from the platform, such as loyalty information and bank information. This is called *enrichment*, indicated by `ADDITIONAL_DATA_AVAILABLE`. The POS can now adjust the amount accordingly, while the PED waits for this amount in the `WAIT_FOR_AMOUNT_ADJUSTMENT` state.

- Receipt printing

  `PRINT_RECEIPT` indicates that there is a receipt available for printing; this usually happens at the end of a tender, but can also occur multiple times (for example when the tender gets canceled). After printing, the state will be `RECEIPT_PRINTED`.

- Signature checking

  In some situations, the signature of the shopper needs to be checked. First, it needs to enter the signature (`ASK_SIGNATURE`), after which the cashier checks the signature and confirms it (`CHECK_SIGNATURE` and `SIGNATURE_CHECKED`).

- Finish

  Eventually, the state is either `APPROVED`, `DECLINED`, `CANCELLED`, or `ERROR`. Note that although this is to some extent merely informational, it wraps up the tender and is therefore considered to be part of this set of states.

Next to the core states, there are states used merely for information purposes; they only inform the POS about the progress and do not influence the flow:

- Payment instrument statuses

  Depending on the chosen payment instrument, it is one of the following:

  - `CARD_INSERTED`, followed by `WAIT_FOR_APP_SELECTION` and `APPLICATION_SELECTED` for choosing the application (e.g., Maestro or MasterCard)
  - `CARD_SWIPED`
  - `PROVIDE_CARD_DETAILS`, followed by `CARD_DETAILS_PROVIDED`

If applicable, the shopper thereafter needs to enter his PIN. The progress of this indicated rather detailed with the `WAIT_FOR_PIN`, `PIN_DIGIT_ENTERED` and `PIN_ENTERED` states.

- DCC

  If the shopper wants to pay in the local currency, it gets offered DCC. This is indicated by `ASK_DCC`, followed by either `DCC_ACCEPTED` or `DCC_REJECTED`, depending on the choice of the shopper.

- Gratuity

  If applicable (for instance in restaurants or bars), the shopper can indicate a gratuity (or tip). `ASK_GRATUITY` and `GRATUITY_ENTERED` are states related to this.

It is self-evident that systems in the payment industry have to be very secure. Therefore, no sensitive data related to the tender is shared with the POS, and all necessary outgoing communication (to the platform and beyond) is encrypted. The PED has secure elements in place to allow for hardware-based encryption, and when using ICC, the card itself can also generate cryptograms, as required by the EMV protocol. The details of the security mechanisms of the POS solution are beyond the scope of this project.

2.2.2. EMV Application Specification

Each ICC transaction on the PED goes through eleven steps, as defined in the EMV specification [18]. We will provide a brief description of those steps in this section, as (for instance) Gerts [22] already provides a detailed description. The listed order of the steps is mandatory for most steps, but for some intermediate steps, the order is allowed to be different.

1. Initiate Application Processing
   This is the first step after the application on the ICC was selected. During this step, the transaction gets prepared by providing the ICC with terminal-related information.

2. Read Application Data
   The terminal reads information needed for the transaction from the ICC.

3. Offline Data Authentication
   If both the terminal and the ICC support some common method of offline authentication, the terminal authenticates the card during this step. This can either be done statically or dynamically.

4. Processing Restrictions
   The terminal checks compatibility, whether the card may be used for the purpose of the transaction (e.g., whether it is not an ATM-only card) and whether the used application is effective and not expired.

5. Cardholder Verification
   This step ensures that the person using the card is actually the holder of the card. Based on the allowed cardholder verification method (CVM), which is listed on the ICC and in the terminal, the cardholder for instance needs to enter his PIN or put a signature on the receipt.

6. Terminal Risk Management
   To protect the acquirer, the issuers, and the system from fraud, the terminal can force an online verification for certain transactions (e.g., high-value transactions or transactions with cards that did not go online for a long time).

7. Terminal Action Analysis
   Based on the previous steps, the terminal makes the decision to process the transaction offline or to go online. This decision influences the type of cryptogram that gets requested from the ICC.

8. Card Action Analysis
   The ICC itself can also do some risk management. Based on the result and the previous steps, this leads to the decision to indeed stay offline or to go online.

9. Online Processing
   If the transaction needs to be verified at the issuer (i.e., online), that happens during this step.

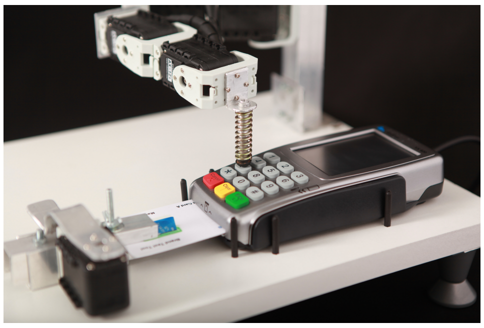


Figure 2.4: Photo of an early version of the test robot while testing a PED.

10. Issuer-to-Card Script Processing
    The issuer might need to run certain scripts on the ICC, for instance to guarantee continued functioning of the application. This step does not necessarily influence this transaction.

11. Completion
    During this last step, the transaction is finished and the final result is decided.

Note that MSR transactions cannot go through all these steps, as there is no interaction with the card (only static data from the swipe track). Similarly, contactless transactions follow a different flow.

2.3. Test Environment

The Adyen POS solution thus consists of two components: the library software running on the POS and the application software (also referred to as *firmware*) running on the PED. Adyen used to test this solution fully manually, by manually starting transactions and observing the different end results. To industrialize (and speed up) this process, a testing robot was developed (see Figure 2.4), which simulates a POS device and the human interaction of a shopper. The robot runs a set of test payments, and compares the actual results with the expected results. With this, it tests both the library software (on the POS) and the application software (on the PED).

The test setup consists of a custom software application (controller) that both implements the library software and mimics the human interaction through the robot. The robot consists of a controllable arm with one 'finger' for pressing buttons (and drawing simple signatures on the screen of the PED), a module that can insert a payment card, a module that simulates a swiped card, a module for contactless payments, and a camera to capture the visual feedback provided to the user. In addition, the controller can simulate different types of disturbances to the environment, such as power outages (for POS and/or PED) and connectivity issues. These connectivity issues can potentially occur in the shop (e.g., a bad cable between POS and PED) and outside the shop (e.g., a broken internet connection). A schematic overview of this setup is shown in Figure 2.5. The controller relies entirely on the library to determine the necessary robot action. For example, if the library indicates that the PED waits for a card, the controller tells the robot to insert the card.

Furthermore, the test setup relies on a test version of the online platform, which is similar to the production version. The main difference is that the test platform does not communicate with the schemes, but with a so-called 'yes man', which can generate different issuer responses. Generally, those responses indicate an approval. However, any amount ending on a value between 120 and 136 will trigger a declined response with different error types for each value [11]. For example, an amount of €101,23 will trigger an 'acquirer declined' result, whereas an amount of €101,26 will indicate that the card is expired.

All test cases are described in Extensible Markup Language (XML) files, which are parsed by the controller. Such test case contains all parameters needed for a tender. For example, it can contain the tender amount, the PIN that needs to be entered, the expected output from the library, et cetera. One XML file can contain multiple test cases. Additionally, it is possible to generate combinations of parameters without specifying each test case explicitly.

Adyen currently uses the robot to perform two types of tests on the full POS solution: functional tests and

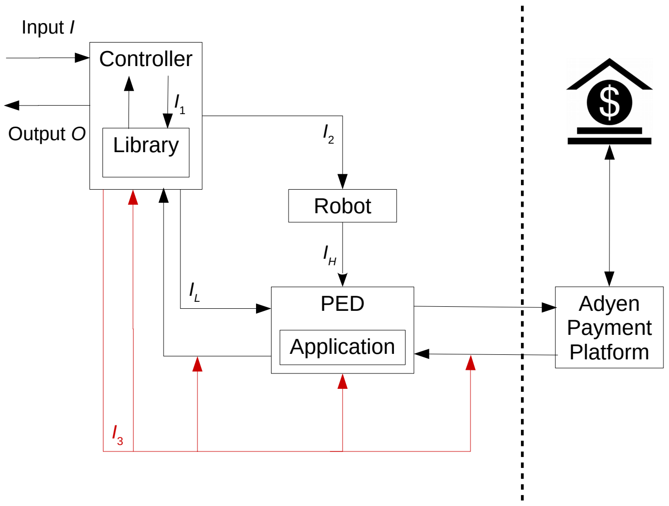


Figure 2.5: A schematic overview of the test environment. The controller takes XML files with test cases as input (I), and outputs the results of the executed test cases, including log files (O). While testing, it interacts with the library (I_1) and the robot (I_2). The library communicates in a regular way with the PED (I_L), and the robot simulates the human interaction with the PED (I_H). Furthermore, the controller can introduce environmental disturbances (I_3), such as power outages and network issues.

load tests. Functional tests try to cover as many (edge) cases as possible to ensure correct operation of the system by comparing the actual output with the expected output. Load tests check whether the system still performs as expected after multiple hours of (repeated) random payments. Besides the robot setup, there are currently no other automated test methods in place.

2.4. Transaction Log Files

Every POS transaction at Adyen is being logged accordingly in so-called *transaction logs*[1], which the PED submits to the platform. Each log line in such transaction log follows the simple format `timestamp event`, with the timestamp being formatted as `YYYYMMDDHHMMSS`, and the event as string with any character (including whitespace). Listing 2.1 lists a full example of such transaction log. On average, the transaction logs have a length of 15 to 25 lines. Those contain critical information regarding the PED, such as firmware versions, the used payment instrument (for instance ICC or MSR), the performed CVM, the final status, et cetera. If the used payment instrument was ICC and the PED thus follows the steps as defined in the EMV specification [18] (recall Section 2.2.2), the logs list (most of) those steps accordingly.

Listing 2.1: Example of a transaction log

```
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
```

[1]Note that a transaction log can also refer to the logs a database system maintains to (among others) recover from crashes. When we discuss transaction logs, we always refer to the payment (or tender) logs, unless stated otherwise.

```
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
```

The structure of the logs may vary over different firmware versions, as lines are added/updated in each release (to improve their usefulness). Furthermore, there is a significant difference between the two terminal operating systems Adyen currently supports. The application on eVo (the most common operating system) does not report explicit tender states, whereas the application on the V/OS operating system logs all tender state transitions (e.g., 'from `CARD_SWIPED` to `ASK_SIGNATURE`') in addition to several other lines that are similar to eVo. Simply put, V/OS application logs are longer and hold more information than the logs from the eVo application. Nonetheless, we focus mainly on logs from the eVo application, as this application is more widely used than the V/OS application.

Table 2.1 provides an example of some log data characteristics. It shows the size of the logs from one terminal firmware version on the eVo operating system, originating in one merchant, for varying time intervals. Note that although there are many traces, the number of unique sequences is relatively low; most of the transactions follow similar sequences of events. Note that data that is too specific, such as transaction-specific identifiers, transaction amounts and card brands, was stripped from each set to obtain these numbers (otherwise, the number of sequences and events would have been much larger).

	# logs	# unique sequences	# unique events
15 minutes	482	12	35
one hour	1953	32	60
one day	13995	101	87
one week	60267	200	116

Table 2.1: Overview of log sizes for different intervals, when transaction-specific data (such as identifiers) is stripped. Here, an event is a log line, and a sequence is a log file (i.e., a sequence of log lines). Each data set originates in the same firmware version (with identical configurations) at the same merchant.

3
About Passive Learning

"The greatest value of a picture is when it forces us to notice what we never expected to see."

— J.W. Tukey

A *finite state-machine* (or automaton) is a way to design and describe certain systems. The idea behind this is that a system is always in a certain state, from which it can go to another state in response to some input (either internal or external).

If there is no documentation or source code available that describes the behavior of the system, the system is basically a black box. *Passive learning* techniques approximate the state-machine that a system implements. Passive learning means that there is no need for interaction with the system: the technique relies solely on (historical) observations, such as log data.

In this chapter, we start with some background on the position of passive learning in the research. Then, we present our selection of open source tools, and a performance comparison of those tools (in terms of runtime performance and output complexity). Thereafter, we present the results of a survey that we conducted under ten developers, to identify how they see these tools (and their outputs).

3.1. Related Work

There are multiple different approaches for inferring the behavior of a potentially unknown system, mainly developed by multiple different research fields that have slightly different focuses. In this section, we provide an overview of the area of different inference techniques. We distinguish between process mining, and specification mining or grammar inference; the latter can be split into active learning, and passive learning, which is the focus of this thesis. For each area, we provide some examples of work in this field. Thereafter, we present some related work addressing comparison of different inference techniques.

3.1.1. Process Mining

Process mining [38, 40] (also referred to as business process mining [36, 39]) is a field that focuses mainly on business processes, and not so much on software systems specifically. In general, process mining uses so-called 'event logs' for this. Significant contributions in this area are implemented in the ProM framework [41], a large tool that contains several plug-ins for performing process mining and analyses on that. According to Van Der Aalst [37], process mining has been applied in a variety of sectors and organizations, such as municipalities, hospitals and banks. For example, Van der Aalst et al. [39] did a successful case study on applying process mining at the Dutch National Public Works Department (Rijkswaterstaat). Similarly, Mans et al. [27] applied process mining at a Dutch hospital; their paper focuses on the applicability of process mining in a real scenario. They were able to mine the complex, unstructured process that they had in place. Furthermore, they indicate that they managed to derive understandable models.

3.1.2. Specification Mining or Grammar Inference

Ammons et al. [4] introduce *specification mining*, an approach for discovering formal specifications of the protocols that a system should follow. The goal of the approach is to discover the temporal and data-dependence relationships of a program that uses a certain application programming interface (API). Thus, specification mining relies on (logs of) execution traces.

Cohen and Maoz [13] present a framework to select a subset of traces from a large set of logs, so that the subset likely represents the specification of the full set. This can be particularly useful when the full set is too large to process efficiently.

Grammar inference [14] seems to be closely related to specification mining (there is no clear definition on the difference), but focuses more on the inference itself than on the fact that the system follows a certain specification. Grammar inference can also be applied outside computer science; for example for inferring natural languages [15]. Generally, grammar inference is divided into *active* learning and *passive* learning, which we will both discuss here.

Active Learning

Active learning means that the system under learning (SUL) is actively queried to get the responses to different inputs, and that inferred machines are verified against that system. Using this, active learning is ultimately able to identify the full behavior of the system, as it can simply pose more queries to discover deeper. This technique resides on the fact that a (finite) state-machine always follows the same sequence of states for identical inputs. Most algorithms build upon the $L*$ algorithm by Angluin [5].

In recent work, active learning is commonly used for reverse engineering [42], or to verify an implementation against some specification [1, 2, 21, 25]. For example, Ruiter [33] used LearnLib [31] (a tool for active learning) for various EMV related analysis projects. For one of those projects, they were able to compare the models of two hardware tokens for internet banking, and to discover some undesired behavior in the older (unpatched) device. Smeenk et al. [34] infer the behavior of an embedded control device in a printer, and discover several deadlocks in that system.

Passive Learning

For situations where active learning is too time-intensive or complicated to set up, there is a *passive learning* approach, where different traces (e.g., log files from earlier executions) are parsed to infer the state-machine. As a result, passive learning can be used to analyze real-life behavior of a system (in the sense that traces relate to actual executions), whereas active learning is purely synthetic. On the other hand, passive learning is less exact than active learning, as passive learning relies entirely on (historical) observations, which may or may not be a complete representation of the system.

Intuitively, passive learning is just a matter of overlaying all traces, and determining where the paths diverge. That results in a tree of all different executions. For example, if there are ten unique traces, which are all different from the beginning till the end, then the tree would have ten branches.

Depending on the strategy and desired outcome, the tails of the different branches can be merged into the tails of other branches, in order to decrease the size of the model and thus to improve clarity. One of the most common algorithms that does exactly this is k-tails [9]. The idea behind k-tails is that if two states are followed by the same k states (i.e., have the same *tail* with length k), those two states are merged together. Thus, when k is large, this will likely result in less merges; commonly used values are $k = 1$ and $k = 2$.

There are multiple tools available that implement different strategies, such as Synoptic [6] and DFASAT [23]. We will introduce, discuss and evaluate these tools later in this chapter, thus we will not go into detail here. Another example is SALSA from Tan et al. [35], although they do not explicitly name their approach as 'passive learning'.

3.1.3. Comparing Inference Techniques

There is quite some existing work on comparing the *algorithms* and heuristics used for different inference techniques. For example, Lo and Khoo [26] introduce QUARK, a framework for empirically evaluating automaton-based specification miners. They use precision (i.e., how exact/correct is the machine?), recall (i.e., how many of the expected flows are represented by the inferred machine?) and probability similarity (i.e., how accurate are the frequencies of the different flows?) as metrics. Walkinshaw et al. [43] show how evaluating the accuracy of the inferred machines using precision and recall makes it easier to indicate whether an inferred machine is under-generalised or over-generalised. Pradel et al. [30] present a framework to evaluate how accurate mined specifications of API usage constaints are. Busany et al. [10] address the scalability problem that different algorithms that infer models from log data can have. They introduce a statistical approach to perform the analysis on a subset of the data, but with some statistical guarantees regarding the validity of the result.

3.2. Introducing a Selection of Passive Learning Tools

Passive learning tools infer state-machines from log data (or *traces* [32]). In this section, we briefly list our selection of three open source tools. For each tool, we discuss their algorithms, and their inputs and outputs. Furthermore, we show an example based on four short example logs, as listed in Listing 3.1 to Listing 3.4. We conclude this section with a brief feature comparison of the tools.

The first tool is Synoptic [6], a tool that was designed specifically for analyzing large sets of arbitrary log data without having the need to specify complicated models. The second tool is InvariMint [7], from the same authors as Synoptic. InvariMint implements multiple inference algorithms and aims mainly at improving understandability of those algorithms. The third and last tool is DFASAT [23], a tool that is still under development and which takes execution *traces* as input; this tool requires more domain-specific configuration/tweaking, but should therefore have a better performance and more flexibility.

There are more tools available that do similar jobs, but that do not fit our datasets. For instance, CSight (short for concurrent insight) [8] analyzes logs from a *distributed* system. As we do not deal with a distributed system here, in the sense that all logs are individual and sequential, we do not discuss that tool. Walkinshaw et al. [44] introduce MINT (or EFSM(Inference)Tool). MINT considers data incorporated in execution traces during inference, as events might be related to certain measurements, for instance. The authors illustrate this using a mine pump example, that switches on for certain levels of water and methane. The events in the transaction logs in our dataset do not incorporate such data values, thus we do not investigate this tool. Ohmann et al. [29] do something similar for resource usages in Perfume, such as memory usage or execution times. However, due to the nature of the company, we simply cannot make use of any web-based tool for analyzing their log data.

Listing 3.1: Example log file 1

```
2017−02−01T12:00:01Z − Event A
2017−02−01T12:00:02Z − Event B
2017−02−01T12:00:03Z − Event C
2017−02−01T12:00:04Z − Event D
```

Listing 3.2: Example log file 2

```
2017−02−01T12:01:01Z − Event A
2017−02−01T12:01:02Z − Event C
2017−02−01T12:01:03Z − Event D
```

Listing 3.3: Example log file 3

```
2017−02−01T12:02:01Z − Event A
2017−02−01T12:02:02Z − Event D
2017−02−01T12:02:03Z − Event E
```

Listing 3.4: Example log file 4

```
2017−02−01T12:03:01Z − Event A
2017−02−01T12:03:02Z − Event B
2017−02−01T12:03:03Z − Event E
```

```
2017-02-01T12:03:04Z - Event C
2017-02-01T12:03:05Z - Event D
```

3.2.1. Synoptic

Beschastnikh et al. [6] present Synoptic[1], a tool which aims to make state diagram inference from log files easy, mainly for system debugging purposes. The tool starts by parsing the execution traces out of those files by using (user-specified) regular expressions that indicate the format of the log; this results in a *trace graph*. Thereafter it mines three types of temporal *invariants* from this trace graph, which indicate the relations between different events in a trace that always hold (e.g., a is never followed by b). Using the trace graph, it creates a *partition graph* that contains all events and possible relations between those events. This partition graph then gets refined (or split), so that it actually satisfies the mined invariants. The refined graph gets coarsened (or merged) again, as the refinement heuristic is said to be imperfect (i.e., the result is too refined). When it can no longer coarsen the model (that is, if the invariants would be violated if more merges would take place), Synoptic has finished and outputs the model.

In short, Synoptic outputs a graph model in which all *nodes* represent a log event. On the edges in the model, the (relative) number of traces that follow this transition is shown. This is particularly useful to see which traces occur most of the times, and which traces almost never occur. The downside of using relative numbers is that it is hard to determine the absolute number of traces at a certain point down the tree. Only recently the possibility to display the absolute counts was added.[2]

Example. Figure 3.1 shows an example output for the aforementioned logs. We capture only the specific event letters (i.e., A/B/C/D/E), to keep the graph compact. To obtain this graph, we ran Synoptic as follows:

```
./synoptic.sh /path/to/example-logs/example_[1-4].log \
-o /path/to/output/example_synoptic \
-r "^.*Event\\s(?<TYPE>)$"
```

3.2.2. InvariMint

The authors of Synoptic also created InvariMint[3] [7]. InvariMint aims at improving understandability of inference algorithms, by describing an approach to model inference algorithms declaratively. Other than Synoptic, InvariMint models log lines on the *edges*, and the nodes are basically anonymous (without a meaning). It ships with a few different algorithms following the declarative approach, among which k-tails [9] and Synoptic's algorithm.

Their new implementation of the algorithm from Synoptic yielded some improvements, especially in terms of performance. However, where Synoptic produces models that are in essence consistent with the traces, InvariMint produces models that *approximate* the traces. For instance, InvariMint may introduce transitions (edges) that were not in the original traces.

Example. Figure 3.2 shows example outputs for the aforementioned logs, for InvariMint's k-tails implementation for both $k = 1$ and $k = 2$. We again capture only the specific event letters, to keep the graph compact. To obtain the graphs, we ran InvariMint as follows, with $k \in \{1,2\}$:

```
./invarimint.sh /path/to/example-logs/example_[1-4].log \
-o /path/to/output/example_synoptic \
-r "^.*Event\\s(?<TYPE>)$" \
--minimizeIntersections=true \
--invMintKTails=true \
--kTailLength=k
```

[1]Available from `https://github.com/ModelInference/synoptic`; we use commit 5e667e69.
[2]When we retrieved the first results from Synoptic halfway June 2016, this functionality was not yet available.
[3]Available from `https://github.com/ModelInference/synoptic`; we use commit 5e667e69.

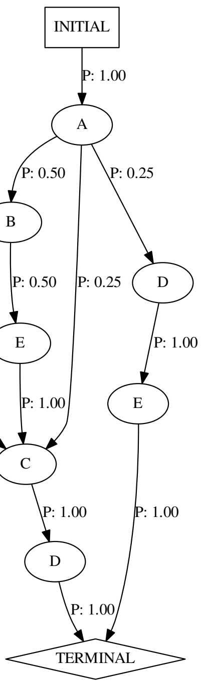
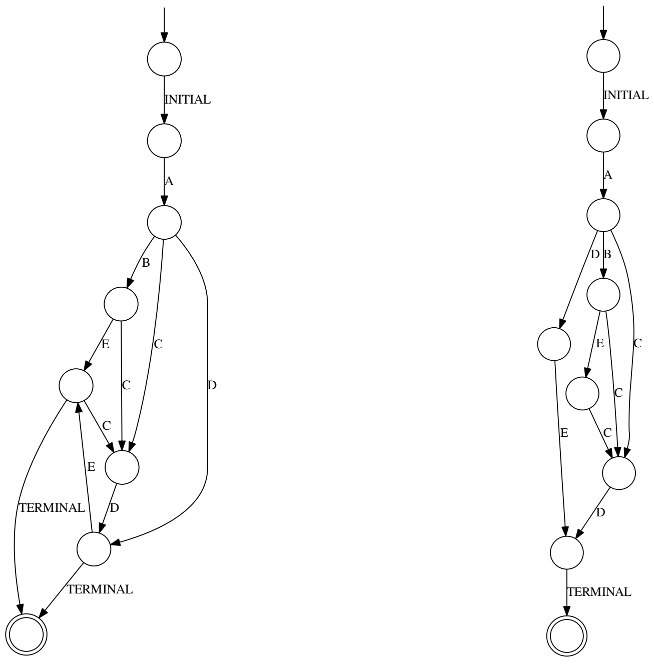


Figure 3.1: Example outputs of Synoptic for the example logs. The nodes contain the events, and the edges indicate the probability of taking that edge. For example, $A \to B$ has a probability of 50%, as it is present in two of the four logs.



(a) $k = 1$ (b) $k = 2$

Figure 3.2: Example outputs of InvariMint for the example logs. The edges show the events that take the edge.

3.2.3. DFASAT

DFASAT[4] is a novel tool based on work from Heule and Verwer [23]. In its core lies a greedy merging algorithm, based on the Blue-Fringe algorithm [24]. First, a tree-shaped graph (an augmented prefix tree acceptor (APTA)) is created from the traces. The idea of this tree is that it illustrates at which points traces diverge. Second, states are merged together, with the condition that the merge is *consistent*. This consistency check depends on the chosen heuristic, but an intuitive example is that an accepting and a rejecting state can never be merged.

During the merging, the algorithm uses the notion of red and blue states; red states are states that cannot be merged with any other red state, blue states are the next merge candidates. This merging process is repeated iteratively until no other merges are possible (i.e., when all states are red). The conditions for merging are determined by the chosen heuristic; one of the included heuristics is *overlap driven*, which is based on *ALERGIA* [12]. The idea behind the overlap driven heuristic is simple: the nodes that overlap the most (i.e., that have the most common outgoing edges) are merged together. DFASAT has a modular setup, which allows users to rather easily write their own merge conditions, consistency checks, et cetera (in C++).

The input traces follow a *fixed* but simple format, an example of which is shown in Listing 3.5. The first line contains the number of traces and the alphabet size (i.e., the number of different symbols), the lines thereafter contain the trace type (i.e., whether the line ends in an accepting or a rejecting state), the number of symbols on that line and the symbols themselves, all separated by whitespace. Note that, as a consequence, the events themselves cannot contain any whitespace.

Listing 3.5: Example log files 1-4 in a format for DFASAT

```
4 5
1 4 A B C D
1 3 A C D
1 3 A D E
1 5 A B E C D
```

After processing, it outputs a graph, just like the other tools. On the edges, the symbols (in this case, the log lines) are shown, and the nodes show the number of traces passing through the node. DFASAT by default adds so-called sinks for low-probability transitions, to avoid clutter in the resulting automaton. It also provides several configuration parameters, such as the t and y parameters, which indicate the number of times a state respectively a symbol must occur in order to be taken into account during the merging process. Using those parameters, the level of detail can be tweaked (to a certain extent). Another option is to use DFASAT with a satisfiability solver, but according to one of the authors, this functionality might actually be broken, thus we do not use that here.

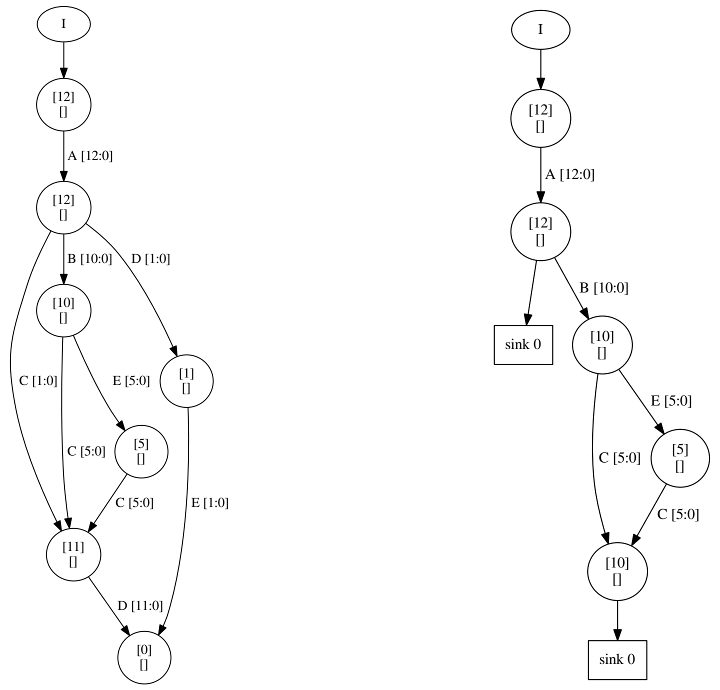
Example. To run DFASAT on the example logs, we first need to convert them to the aforementioned input format for DFASAT. The result of this (manual) conversion is shown in Listing 3.5. However, to demonstrate the statistical features of DFASAT, we duplicated example log 1 and 4, so that they “occur” 5 times; this is shown in Listing 3.6.

If we execute DFASAT with the overlap driven heuristic while taking all traces into account (i.e., with $t = 0$ and $y = 0$), we obtain the graph shown in Figure 3.3a. However, if we only take the more significant traces into account (i.e., with $t = 5$ and $y = 0$), we obtain Figure 3.3b. Observe that DFASAT introduced the aforementioned sinks for the infrequent paths ($A \to C$ and $A \to D$). In addition, as C is always followed by D, DFASAT also introduced a sink there.

To obtain these graphs, we ran DFASAT as follows, with $t \in \{0, 5\}$:

```
./dfasat -t t -y 0 -m2 -n 1 -b 0 -d 20000 \
-q overlap_driven -z overlap_data \
-o output \
/path/to/example-logs/example_dfasat_more.log \
"/bin/true"
```

[4]Available from `https://bitbucket.org/chrshmmmr/dfasat`; we use commit e6c66019.



(a) $t = 0$, $y = 0$ (b) $t = 5$, $y = 0$

Figure 3.3: Example outputs of DFASAT (with the overlap driven heuristic) for the example logs. The nodes indicate the number of traces going through the node. The edges indicate the events taking the edge, and how often that event leads to respectively an accepting or a rejecting end state. Note that we did not use any rejecting traces, and that the latter is therefore always 0.

Listing 3.6: Example log files 1-4 in a format for DFASAT, with log 1 and 4 having 5 occurrences.

```
12 5
1 4 A B C D
1 4 A B C D
1 4 A B C D
1 4 A B C D
1 4 A B C D
1 3 A C D
1 3 A D E
1 5 A B E C D
1 5 A B E C D
1 5 A B E C D
1 5 A B E C D
1 5 A B E C D
```

3.2.4. Summary

We refer to Table 3.1 for a brief overview of the different tools. Note that most of the tools are quite similar in terms of features from the ‘user’ perspective. Each of the tools produces a graph model, but the meaning of the graphs varies slightly: Synoptic puts log events in the nodes, whereas InvariMint and DFASAT put them on the edges. For InvariMint, the nodes have no meaning, for DFASAT they contain the number of traces that go through them. Both Synoptic and InvariMint can process log data in any format, whereas DFASAT requires some preprocessing to a fixed format. DFASAT hides infrequent paths (i.e., ignores rare traces and ends them in sink nodes), whereas Synoptic and InvariMint put all flows in the graph.

	Synoptic	InvariMint	DFASAT
input format	any logs	any logs	formatted traces
meaning of nodes	events	none	number of traces
meaning of edges	number of traces	events	events
ignores infrequent paths	no	no	yes

Table 3.1: Feature comparison of the three inference tools under investigation.

3.3. Comparing Synoptic, InvariMint, and DFASAT

To get a feeling for how the different tools perform, we did a small experiment using representative datasets for our use cases. In this section, we briefly describe how we compared the tools, and we show some of the results in terms of runtime performance and complexity.

3.3.1. Methodology

As the tools allow for custom configurations that are likely to influence their performance, we first had to make a few configuration decisions for each of the three tools, to limit the scope of this experiment. We use the default values, except for the following: For Synoptic, we switch the edge labels to display absolute counts. For InvariMint, we pick k-tails with $k = 1$ and enable minimization of intersections. For DFASAT, we pick the overlap driven heuristic with non-randomized greedy preprocessing and state count $t = 10$; we disable the use of the SAT solver. Throughout this thesis, we always use these settings, unless stated otherwise.

We use four different transaction log datasets, all originating from production terminals of a large merchant: a relatively small set of 15 minutes, a slightly larger set of one hour, a set of one day and a large set of one week. As Adyen patches software versions relatively regularly, and we focus on one particular software version (to eliminate inconsistencies in logging between the different versions), one week of logs is sufficient, especially for identifying recent issues. Data that is too specific, such as transaction-specific identifiers, transaction amounts and card brands, was stripped from each dataset.

The numerical details of the datasets were listed earlier in Table 2.1. We consider each dataset both as full set and as set of unique traces, thus we have eight datasets in total. As the tools infer *behavior* from the logs, we do not expect to see significant differences between the unique set and the full set (other than the trace counts on the edges), as the full set only holds more traces describing identical behavior. We are curious how introducing more identical traces affects the performance of the tools.

Using the various datasets (transformed into a format that the tools can read), we analyze the performance of the three tools. For each data set, we run each tool ten times to eliminate fluctuations. Ultimately, we answer the first research question:

How do different passive learning tools behave when dealing with extensive log data?

3.3.2. Findings

We categorize our findings in two groups: runtime performance and output complexity. For each of them, we show the results and observations we make about these results.

Runtime performance. The average runtimes of the ten executions are shown in Table 3.2, which is visualized in Figure 3.4. From these figures, it is easy to conclude that DFASAT significantly outperforms the other tools in terms of runtime. We conjecture two possibilities for this: on one hand, its heuristic is more efficient and greedy, and on the other hand, DFASAT is implemented in C++ as opposed to Java.

All results indicate the time it takes to import the dataset and to process the output. For all tools, this includes the time Graphviz[5] (open source graph visualization software) needs to process and output the resulting graph in PNG format, as Synoptic and InvariMint have this functionality embedded.

[5]`http://www.graphviz.org`

		Synoptic	InvariMint	DFASAT
15 minutes	all logs	1884 ms	1435 ms	222 ms
	unique logs	921 ms	770 ms	78 ms
one hour	all logs	17555 ms	3185 ms	343 ms
	unique logs	3382 ms	1258 ms	108 ms
one day	all logs	–	21064 ms	1410 ms
	unique logs	32411 ms	2474 ms	180 ms
one week	all logs	–	202668 ms	4383 ms
	unique logs	729623 ms	3980 ms	451 ms

Table 3.2: Runtime comparison of the three tools, running on the different datasets. Runtimes are indicated in milliseconds, as the average of ten sequential runs. Note that Synoptic did not (always) complete on the full day and week datasets.

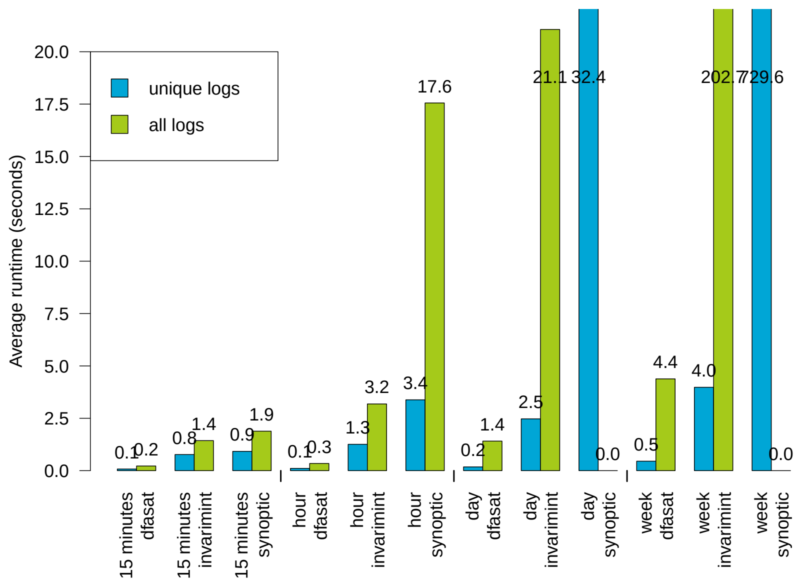


Figure 3.4: Plot of the runtime comparison of the three tools, running on the different datasets.

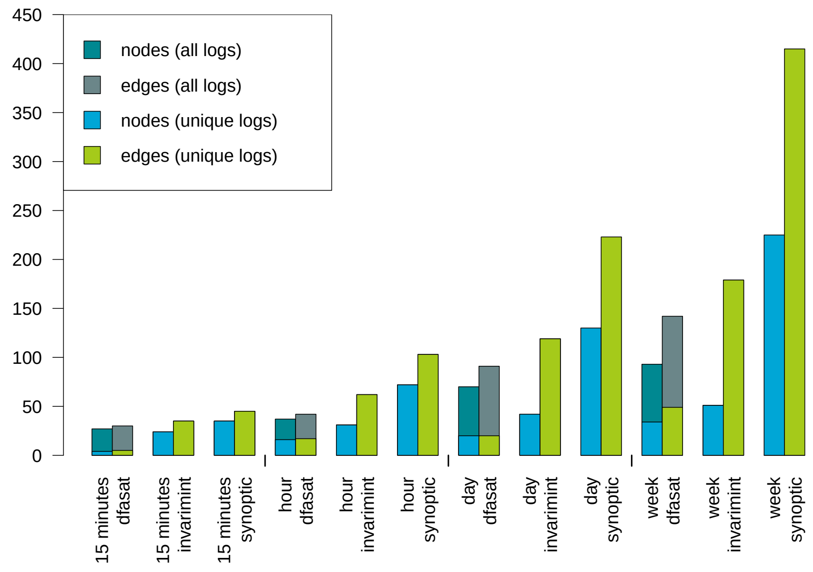


Figure 3.5: Plot of the complexity comparison of the three tools, running on the different datasets. It contains the number of nodes and edges in the graphs for both the unique sets and the full sets. The number of cycles and the cyclomatic complexity are omitted for clarity.

One important finding is that Synoptic was not able to always complete for the largest datasets (all logs for one day and one week) without running out of memory. However, if we only consider the unique logs, Synoptic is able to produce a model without running out of memory. The other tools always complete for all datasets.

Key finding 1: DFASAT has the best runtime performance, followed by InvariMint. Synoptic runs out of memory for large datasets.

Output complexity. We restrict ourselves to a few graph characteristics, namely the number of nodes (N), edges (E) and cycles (C) it contains. Furthermore, we compute the cyclomatic complexity (CC), a commonly used complexity metric in computer science, as $E - N + 2P$ [28]. In this case $P = 1$, as the graph is always one connected component. Although complexity is not an unambiguous metric, it does allow us to compare the graphs numerically. Furthermore, a higher graph complexity usually makes understanding such graph more difficult, whereas we want to analyze the graphs by hand. Figure 3.5 shows a comparison of the number of nodes and edges for each of the datasets from the different tools.

The exact numbers are listed in Table 3.3, which also includes the number of cycles and the cyclomatic complexities. From this table, we can draw several interesting observations. For instance, all tools have increasing numbers as the dataset gets larger. Synoptic has a large number of nodes and edges, but InvariMint has twice as many cycles. The results from DFASAT are clearly influenced by the uniqueness of a trace, as the numbers are much larger for sets of all traces. We can relate this to the significance parameters: for example, in a unique set, a certain state might occur only once and might therefore not be considered. However, in the related full set, that same state might occur multiple times, so that it actually *is* being considered and thus influencing the graph.

Key finding 2: Based on the number of nodes, edges, cycles and cyclomatic complexities of the graphs, DFASAT generally produces the least complex outputs.

3.4. Developers' Perceptions of Passive Learning Tools

How do developers perceive the usefulness of such tools and techniques? Understanding what they want from these techniques would enable us to improve them. To that end, we therefore conduct a survey among

		Synoptic				InvariMint				DFASAT			
		N	E	C	CC	N	E	C	CC	N	E	C	CC
15 minutes	all logs	35	45	0	12	24	35	0	13	27	30	0	5
	unique logs	35	45	0	12	24	35	0	13	4	5	2	3
one hour	all logs	72	103	0	33	31	62	2	33	37	42	5	7
	unique logs	72	103	0	33	31	62	2	33	16	17	2	3
one day	all logs	–	–	–	–	42	119	9	79	70	91	6	23
	unique logs	130	223	8	95	42	119	9	79	20	20	0	2
one week	all logs	–	–	–	–	51	179	28	130	93	142	23	51
	unique logs	225	415	14	192	51	179	28	130	34	49	19	17

Table 3.3: Complexity comparison of the tools, based on the result of the different datasets. The columns list the number of nodes (N), the number of edges (E), the number of cycles (C) and the cyclomatic complexity (CC).

ten developers from the company. In the following sub-sections, we detail the methodology, research questions, and findings.

3.4.1. Methodology

We show the developers one graph from each of the tools (in a varying order), where each graph originates in the same data set from Table 2.1: the full set for one hour. Based on our experience, one hour of logs is enough for these tools to come up with a graph of reasonable size. Besides introducing the developers to the tools, this allows us to perform comparisons between tools.

We asked for their opinion on understandability, possible improvements and suitable purposes for the tools (i.e., for which purpose the tools can be used). Using this survey, we ultimately aim to answer the second research question:

How do developers perceive passive learning tools and techniques?

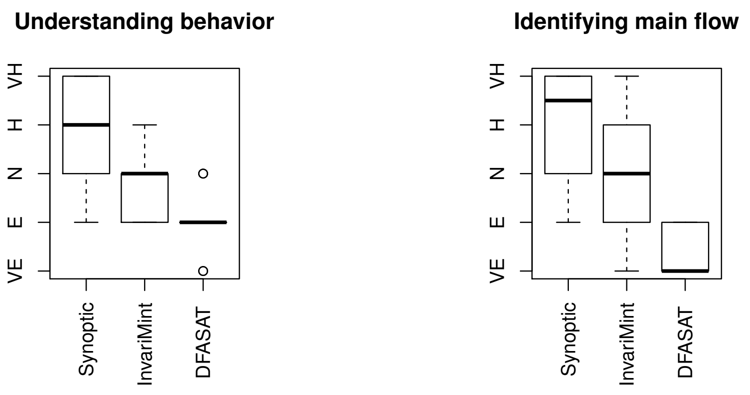
We received, in total, 10 responses to the survey. Before sending it to the developers mailing list, we validated the survey with one developer, which lead us to several improvements – but unfortunately, this made his submission irrelevant (the questions changed significantly). Of the ten responders, six work as developers on the payment system itself. Two of them develop a .NET library that allows for integration with the system, one similarly develops an iOS integration, and one is the test automation engineer who tests this particular system. On average, each of them is employed at this company for roughly one year, and seven of them indicate that they make use of the transaction logs we have used here, on a daily basis.

3.4.2. Findings

We use both Likert-scale questions and open questions, in which we ask for the developer's perception of the graphs. For an industrial adoption, understandability of the results is very important; if developers do not understand the graphs, they will probably not use them. Quantifying understandability is hard, as there is no real metric. Therefore, for each graph, the developer needs to indicate on a scale from 1 to 5 how easy it is to understand the behavior of the system using the graph, and how easy it is to identify the main flow in the system from the graph — 1 meaning very easy and 5 meaning very hard. This second question will make the understandability more concrete, as the developers need to identify something in the graphs. Furthermore, we leave space for other remarks on understandability, as well as for remarks on what needs to be improved.

The outcome of both Likert questions is shown in Figure 3.6. We clearly see that DFASAT's results were considered the easiest ones to be understood by developers. We tested the results using the Wilcoxon signed-rank test, and we find that $p < 0.05$ for *all relations* except for identifying the main flow in Synoptic and InvariMint (in this case, difference was marginally significant, $p = 0.05658$).

Most of the surveyed developers indicate that Synoptic's diagrams are too complex/dense. However, they do think it might get more useful when the dataset is split, or when the graphs are used to zoom in on specific areas. For DFASAT, the sinks raise questions, as they are not defined clearly, but the graphs themselves are much easier to understand. These can be clearly seen as points for improvements in the tools.



(a) Understanding behavior of the system (b) Identifying the main path/flow

Figure 3.6: Boxplots summarizing the developers' perceptions on the tools. VE=very easy, E=easy, N=neutral, H=hard, VH=very hard.

Some other recurring remarks are made about the visualization itself. As the tools rely on Graphviz, the visualization is rather minimalistic, tends to have crossing arrows and does not allow for any interactions (such as collapse/expand, highlighting flows, et cetera). Furthermore, multiple developers suggest the introduction of colors, which can help to visually separate groups of traces. Where we decided to switch to absolute numbers in Synoptic, one of the developers suggested the use of *relative* numbers (for DFASAT), to get a better understanding how often a certain line is logged.

Key finding 3: DFASAT produces a graph that is the easiest to understand. However, a graph from Synoptic could be used to zoom in. Visualization is the main issue.

We asked the developers to rank the eight purposes below, as we expect that the tools can (and should) be useful from these perspectives. Furthermore, we want to identify the likelihood of developers to use the tools for these purposes.

- To understand the system from a high-level view;
- To understand a specific part of the system;
- To understand the interaction between the user and the system;
- To find unexpected paths (i.e., bugs) in the system;
- To share knowledge about the system's behavior among developers;
- As documentation about the system;
- To compare different versions of the system;
- To verify the system against a specification.

In Figure 3.7, we show an overview of the distribution of the listed purposes. From this, we see that there is one clear purpose that comes out as 'most likely': to find unexpected paths (i.e., bugs) in the system. Most of the developers rank 'understanding the interaction between the user and the system' and 'comparing different versions of the system' as second purposes. According to them, the graphs are apparently not very useful as documentation of the system.

As our list is probably not exhaustive, we also asked them to come up with other types of analyses. Their suggestions are varying in the sense that there are no real similarities between responses. We therefore selected three suggestions that differ the most and that we find the most promising:

1. Use these graphs for detecting possible paths that are actually *not being taken*, i.e., if the system supports specific features, why are those not being used?

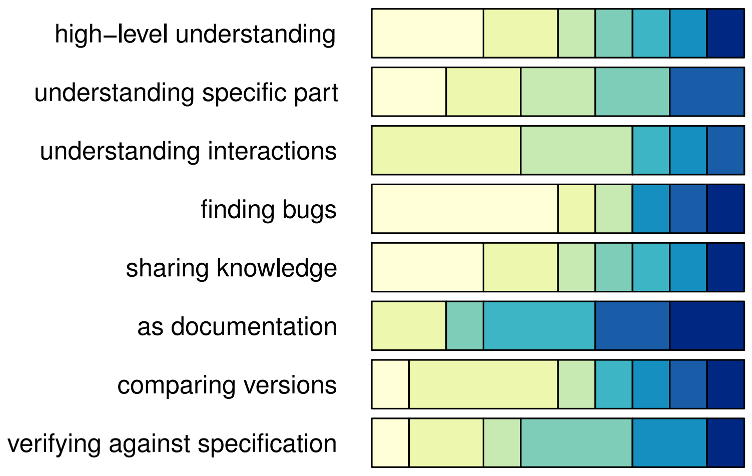


Figure 3.7: Overview of purpose rankings. Yellow colors (on the left) indicate how many responders are likely to use the tools for that purpose, and similarly blue colors (on the right) indicate how many responders are very unlikely to use the tools for that purpose.

2. Relate these graphs to the *code paths* that are responsible for the different paths.
3. Use this for *real-time monitoring* of the system, e.g., if the distribution of the taken paths shifts, or if there occurs a new path, this might indicate some anomaly.

Key finding 4: According to developers, passive learning can help in finding bugs, understanding interactions in the system, and for comparing different versions; however, they do not see it as a way to document software.

3.5. Discussion: Are tools and techniques ready for adoption in industry?

Based on our experiences, we argue that passive learning can be used for analyzing log data, especially if the system produces a large amount of logs that always follow a similar format. However, our study shows that none of the tools are out-of-the-box ready for large industrial adoption. To reach their full potential, the existing tools probably need to be adjusted to suit the particular use case. This is not necessarily very complicated, but it might require some programming knowledge and some knowledge about the field.

Based on our findings, DFASAT is the most promising tool in terms of speed, complexity and understandability. However, it seems to be focused more on academic interests than on an industrial application, making it hard to get the most out of it. On the other hand, Synoptic seems to be the most targeted on industrial applications (it does not require any preprocessing, for example), but for our log data it unfortunately suffers some performance issues, and its outputs become complex as the dataset grows. InvariMint improves in both of those fields, but is not as promising as DFASAT.

Regarding data input, Synoptic (and InvariMint) can already be applied on any log format (as long as there is a regular expression that can describe it), but DFASAT requires some preprocessing. For industrial adoption of such a tool this should be integrated, or part of the toolchain. The fact that DFASAT explicitly allows for customization works in two directions: on the one hand, it allowed us to make the graphs more useful. On the other hand, for industrial use, this is probably undesirable. The source code is not documented, nor does the tool provide clear errors if something fails. Someone without background knowledge on this field probably cannot implement this efficiently.

Another important remark to be made is that, from a business perspective, in general the ‘happy’ path (i.e., the most common flow in the graph) is the most important. However, from a development perspective, the *uncommon* paths are even more important, as those might reveal in which cases the system behaves anomalous, and thus may indicate a bug. This distinction should be kept in mind while using the tools.

3.6. Threats to Validity

In this section, we discuss possible threats to the validity of this study and how we mitigate each one of them.

Internal validity concerns external factors we did not consider that could affect the variables and the relations being investigated.

1. To design the survey presented to developers in the qualitative study, we randomly selected a 1-hour dataset from our data. A different dataset would have resulted in a different output, which could have been evaluated differently by developers. However, our goal was not to identify the best tool among all, but to introduce developers to the existing tools and techniques, and collect their perceptions.
2. We made use of three tools that we found after reading the related literature. There might be other tools available which we did not find. Our findings are still valid as they show what any similar tool should have in order to be useful for practitioners.

External validity concerns the generalisation of results.

1. We conducted the survey with 10 developers from the POS team. They represent all developers we had available for this research. Definitely, we did not achieve data saturation. Still, our goal was to explore ideas rather than generalize the findings.
2. Our study was conducted a single company that deals with a specific market (i.e., payments). In this case, our findings helped the company. Nevertheless, further research needs to be conducted in order to confirm that ideas in this thesis are valid to other companies. We provide a deep discussion about the topic in Section 4.4.3.

3.7. Conclusion

To get a feeling for passive learning tools and techniques, we selected three open source tools: Synoptic, InvariMint, and DFASAT. We analyzed their performance in terms of runtime and complexity, and can conclude that (based on our measurements) DFASAT is the best performing tool; it produces the least complex graphs in a short period of time. However, all tools struggle with increasing data set sizes, especially in terms of graph complexities.

Furthermore, we asked ten developers for their perceptions on the graphs originating in the three tools. They confirm that the graph from DFASAT was the most understandable one. In addition, they think that these tools can be used best for finding bugs, and not for documentation purposes.

4

Valuable Findings At Our Industry Partner

"It seems crucial to be closely involved with the people of the organization itself to carry out a meaningful analysis."
— W.M.P. van der Aalst et al.

In this chapter, we share six real examples of different types of analyses we were able to do using passive learning at our industry partner. These analyses evolved from an iterative process, where we sat down with a company expert on a weekly basis. We were able to identify incorrect and potentially undesired software behavior, (un)common flows in the system, behavioral differences when comparing against a specification, behavioral differences when comparing between different contexts, and slow transitions in the system.

4.1. Introduction

For roughly four months, we were able to perform interesting analyses and improvements in the existing passive learning tools. We met an expert from the company on a weekly basis, and we presented new results and approaches. Before meetings, we printed out the graphs, to make it easier to analyze, and to annotate. We limited ourselves to only a few datasets from two different merchants, and we applied the new approaches on these sets. By adapting to his observations and by coming up with fresh ideas, we iteratively improved the analysis. Figure 4.1 shows a schematic overview of the data flow used for this chapter.

The goal of this chapter is to illustrate what type of analyses (that are useful for the company) can be done using passive learning, and how to improve them. With this, we answer the third research question:

What types of anomalies do passive learning tools identify in a system of our industry partner?

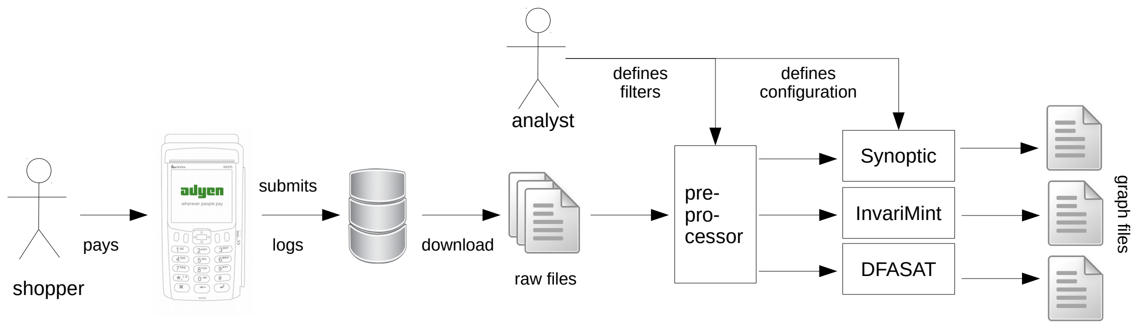


Figure 4.1: Schematic overview of the data flow. A shopper pays, after which the logs are submitted to the platform. We download a set of these logs, and apply preprocessing (and filtering) to put the logs in the format for the tools. Then, we analyze the different results.

4.2. Findings

We present six different examples of what we believe are scenarios where passive learning can reveal useful insights:

1. discovering a bug in the testing environment;
2. checking conformance (i.e., comparing inferred models to a specification);
3. revealing undesired behavior in the system (in a production environment);
4. identifying how inferred models can help to reveal the most common flow(s) in a system;
5. comparing two models from a specific *context*, in this case card brands;
6. identifying slow transitions in the system.

We illustrate all these examples with subgraphs from the related inferred graphs, as the full graphs are too large to print (they would only obfuscate the essence of the examples). For each finding, we also describe the impact for the company.

4.2.1. Unexpected Behavior Caught In A Testing Environment

Our first experience with passive learning was applying it on logs from the testing process, where a new firmware version for V/OS was being tested. Specifically, these tests focused on swipe transactions, where PIN entry (as opposed to putting a signature) is required.

Recall from Section 2.4 that the V/OS application logs explicit state transitions (i.e., the system goes from state *A* to *B*). We took 20 logs for this proof of concept, and applied Synoptic on them; we only had to define a regular expression to capture *B* as log event. The resulting graph is shown in Figure 4.2.

From this graph, the test automation engineer was already able to discover a bug that he did not detect earlier: of the 11 transactions that reach `PIN_ENTERED`, only 10 transactions continued to `PRINT_RECEIPT`. One transaction proceeds with `ASK_SIGNATURE` instead; this is indicated by the red arrow in the graph. This should not have happened for this selection of test cases.

Impact. One of the developers confirmed that this was indeed a bug in the firmware of the terminal. The issue was fixed in the next firmware version.

4.2.2. (Non-)conformance With The Specification

The company expert wants to verify the behavior of the system against a specification. To investigate the usability of passive learning for this, we took the EMV specification [18] (developed by EMVCo, a payment authority), and compared one part of the inferred graph to the related part of the specification. Essentially, this specification consists of a list of steps the system needs to execute sequentially, as we discussed earlier in Section 2.2.2.

During the (manual) comparison, we noticed a different order of events in the logs, i.e., two steps were switched. Provided that the desired order of events is known, this is easy to see in the graph: if the correct order would be first *A*, then *B*, there are edges containing *B* before *A* instead. An example of this is shown in Figure 4.3. We learned from the specification that this was actually allowed, so in this case, this was not a bug. However, it illustrates how passive learning could be used for conformance checking.

One might argue that this could also be determined using one log file. However, passive learning could help to verify that the order is correct in *all* log files. An important remark here is that, unfortunately, not all eleven steps are listed in the logs. Thus, in order to truly verify a correct order of events, those logs need to be adjusted.

Impact. Based on the available information, we concluded that the software indeed matches the specification (i.e., all steps are performed in an allowed order). Thus, there was no impact, other than increased confidence in the system.

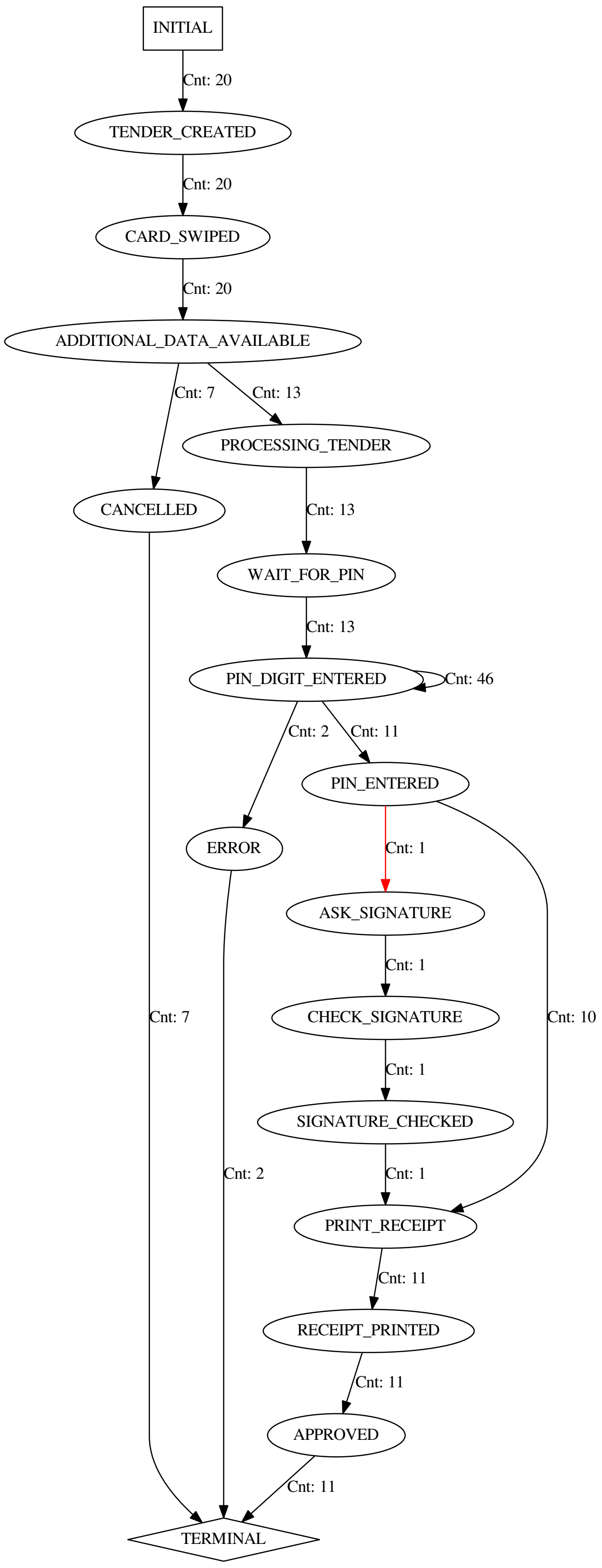


Figure 4.2: Result from Synoptic, using 20 transaction logs from the test environment. The red arrow indicates erroneous behavior, as the PIN was already entered (in these tests, one cardholder verification method (CVM) is sufficient).

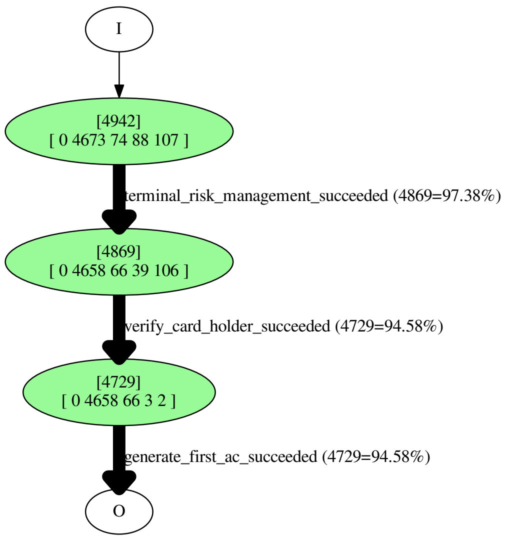

I
[4942]
[0 4673 74 88 107]
terminal_risk_management_succeeded (4869=97.38%)
[4869]
[0 4658 66 39 106]
verify_card_holder_succeeded (4729=94.58%)
[4729]
[0 4658 66 3 2]
generate_first_ac_succeeded (4729=94.58%)
O

Figure 4.3: Subgraph of the result from DFASAT, using a random sample of traces from 5000 transaction logs. This subgraph only shows a part of the common flow (alternative flows are hidden, and as a result, the numbers in the figure do not add up) – node *I* illustrates the prior part for this flow, node *O* the remainder of the flow. According to the EMV specification, card holder verification is step 5, terminal risk management is step 6, and the generated card cryptogram is a result of step 7 and 8. In this subgraph, it becomes clear immediately that the firmware uses a slightly different order. Note that this particular case is actually allowed by the specifications.

4.2.3. Undesired Behavior

Sometimes, certain calls to the online platform might fail. This should never happen, especially for terminals using a wired network connection. However, when we inferred a model from production logs of one merchant, the company expert immediately learned that some terminals sometimes fail to make such important calls to the platform. This is easy to identify by analyzing the flow in the graph, as this will result mainly in declined or cancelled transactions at some point.

The relevant cut-out of the graph is shown in Figure 4.4. Note the blue color of the node that follows `validate_-1`, which shows that from there, most transactions end up in a cancelled state, which in itself can also be an indicator of misbehavior.

Impact. In a later firmware version, a retry mechanism was added to lower the probability of `validate_-1` (and other connectivity related issues) – for this newer version, we were able to confirm that the frequency indeed lowered significantly.

4.2.4. Most Common Flows Through the System

Passive learning can also be used to identify which flows occur the most frequently in the system, provided that the inferred model holds information about the relative frequency of an edge. If this information is present, it is possible to (for example) change the line width of the edges accordingly, which visually helps to identify the most common flows, especially in larger graphs. This is visible in Figure 4.4, where the `validate_0` transition is part of the main flow.

One important finding is that if the logs of successful transactions (i.e., traces that end in an 'approved' state) are included, this renders the changed line width useless for the largest part of the graph. The main reason for this is that the largest part of the transactions in general ends in an approved state, which results in one bold arrow depicting this 'approved' flow, and thin arrows for all other transitions. In contrast, this bold flow will still help to identify the deviations from the most common flow(s), and at which point those deviations occur.

Impact. This finding did not have any direct impact on its own. However, this finding did influence the other

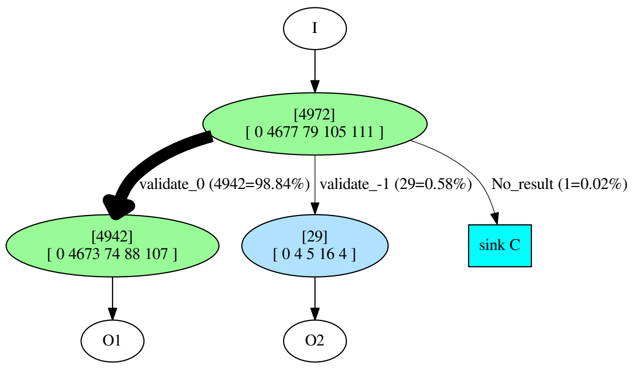


Figure 4.4: Subgraph of the result from DFASAT, using a random sample of traces from 5000 transaction logs. Node *I* illustrates the prior graph for this part, the nodes *O* the remainder of the graph. Each node holds the total number of traces through the node on the first line, and the number of respectively unknown, approved, declined, cancelled and error traces on the second line. The `validate_-1` indicates an undesired failure. After such event, 16 of the 29 transactions end up in a cancel, hence the blue color of the node.

findings, as it improves the visibility of potential issues that occur frequently.

4.2.5. Behavioral Difference Between Two Different Card Types

This time, we took a large transaction log dataset from one merchant on production. From this set, we extracted two different sets representing the erroneous behavior for two different card brands (i.e., we stripped traces that end up in an approved state).

From the two graphs (subgraphs of both are shown in Figure 4.5), we could almost immediately discover several differences. For example, if we look at the cancel and error rates for this particular part of both graphs. Type *B* ends up in more than twice as many cancels and errors as type *A*. There are also behavioral differences (albeit minimalistic ones), as type *B* shows an edge that does not exist in type *A* and vice versa.

Impact. As the company does not have any direct influence on the card brands, this finding did not have any impact on the firmware. However, the information was shared with the merchant(s) to whom this might be of concern, as an explanation for lower authorization rates.

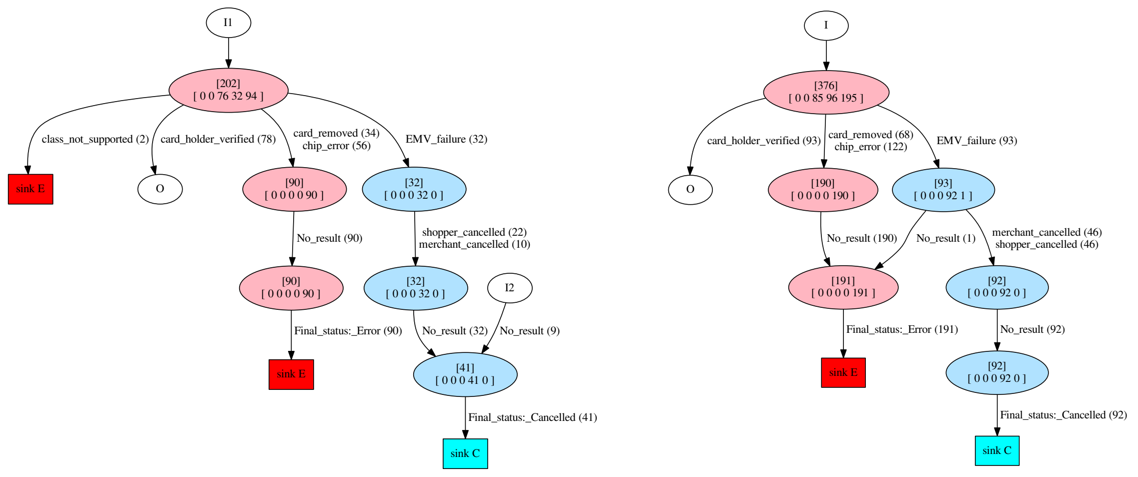
4.2.6. Identifying Slow State Transitions

Similar to the line width in Section 4.2.4, it is possible to highlight the edges with a long duration, to make it easier to identify which transitions need more time. This is specifically useful to find the time-related bottlenecks in the system. Note that displaying this information on the edges can significantly increase the graph size (in terms of space needed for printing all labels), depending on the number of desired metrics.

Figure 4.6 shows the same graph as Figure 4.4, with the timings added to edges. We have discussed earlier that `validate_-1` is undesired, as it is a potential indicator for failed calls to the platform. By just looking at the timings, one might also conclude that `validate_-1` is unwanted, as an average duration of 22.5 seconds for one step of a payment is simply undesired. Ultimately, a payment would be completed as fast as possible, so that the shopper can move on.

Note that when looking at timings in a system like this, there needs to be a distinction between time needed by the system, time needed by the attendant (or merchant), and the time needed by the shopper. Only the first of those could potentially be improved, whereas the others fully rely on other actors.

Impact. With this finding, we identified a few bottlenecks after performing more than 200 benchmark tests on one test robot (repeatedly executing two or three happy flow test cases). Those bottlenecks were resolved by the developers in a later version, improving the total time needed for a payment (in some cases, the gain was more than 4 seconds).



(a) Type *A*

(b) Type *B*

Figure 4.5: Comparison between two different card types for one particular part of the flow. Nodes *I* illustrates the prior graphs for this part, node *O* the remainder of the graphs. Both are cutouts of the full graphs (produced by DFASAT), originating in a random sample of traces from 5000 transaction logs, from which we stripped the approved transactions. Each node holds the total number of traces through the node on the first line, and the number of respectively unknown, approved, declined, cancelled and error traces on the second line.

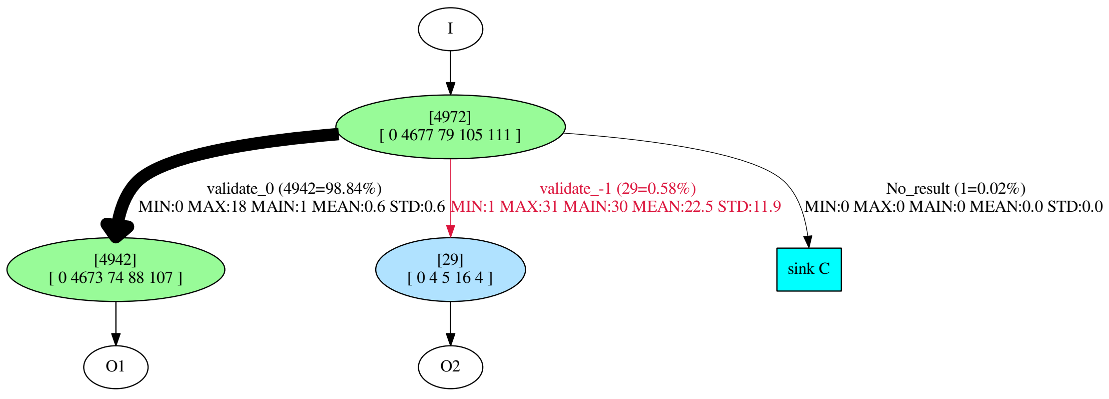


Figure 4.6: Subgraph of the result from DFASAT, that indicates the timings recorded for the validate call, based on a random sample of traces from 5000 transaction logs. Node *I* illustrates the prior graph for this part, the nodes *O* the remainder of the graph. The edges list five metrics: the minimum (MIN), the maximum (MAX), the most common (MAIN), the average (MEAN), and the standard deviation (STD). Thus, on average, a successful call takes 0.6 seconds, with a standard deviation of 0.6; in contrast, a failed call takes on average 22.5 seconds, with a standard deviation of 11.9. In this case, slow transitions (average duration larger than 10 seconds) are highlighted in red.

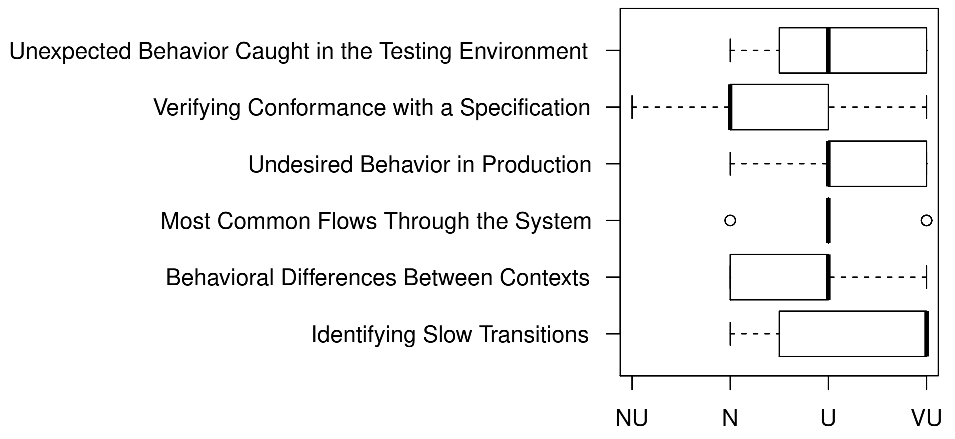


Figure 4.7: Usefulness of the findings, as indicated by developers. All findings are deemed useful, with identifying slow transitions being the most useful, and conformance checking being the least useful. None of the developers rated any of the findings 'not useful at all'. NU=not useful, N=neutral, U=useful, VU=very useful.

4.3. Developers' Perceptions on the Findings

To identify the extent to which the presented findings are useful for the development team, we again conducted a survey with the developers. Key questions in this survey are 'How useful is this finding for you?' and 'Please rank the findings'. We followed a methodology similar to Section 3.4, but we intentionally chose to ask mainly Likert-scale questions this time. However, as the developer with whom we verified the survey did not have enough space to fill in some remarks, we added additional room for motivations.

This time, we received 11 responses in total. Half of this group participated in the first survey, the other half did not. Six of the respondents work on the payment system, two develop libraries (one for .NET, one for Java) for integrating with the system, one is a backend developer, and two are test automation engineers.

Based on Figure 4.7, we can conclude that the developers deem all findings useful, with Identifying Slow Transitions being the most useful, and Verifying Conformance with a Specification being the least useful. While looking at the ranking provided by developers, we learn that identifying unexpected/undesired behavior, either in test or in production, is the most important for them, as can be seen in Figure 4.8.

To the question "Would this graph help you to find the root cause of the issue?" (which we asked for both the unexpected behavior in test and the undesired behavior in production), most of the developers reply either neutrally or slightly positively. One of them indicated that these tools would be better suited for *signalling* a potential problem than for solving that problem.

From remarks in most of the responses, it again becomes clear that a user interface would contribute significantly to the usefulness. For example, as it could allow for changing the abstraction level, and for viewing the actual logs that caused certain paths. One of the developers suggests to also include logs from the library software, to make the picture even more complete. He also envisions using these tools for diagnosing implementation problems on the merchant side.

In one of the final questions, we asked how useful the respondent finds passive learning tools in general. Half of the group replied 'useful', the other half 'very useful'.

4.4. Discussion

From all these findings, we can learn several lessons. In this section, we will group and discuss these lessons. First, we list the changes we made to DFASAT in order to infer most of the graphs in this chapter. Then, we explain why considering the contexts from the logs can add significant value, and the steps that can be taken to adopt passive learning in another setting.

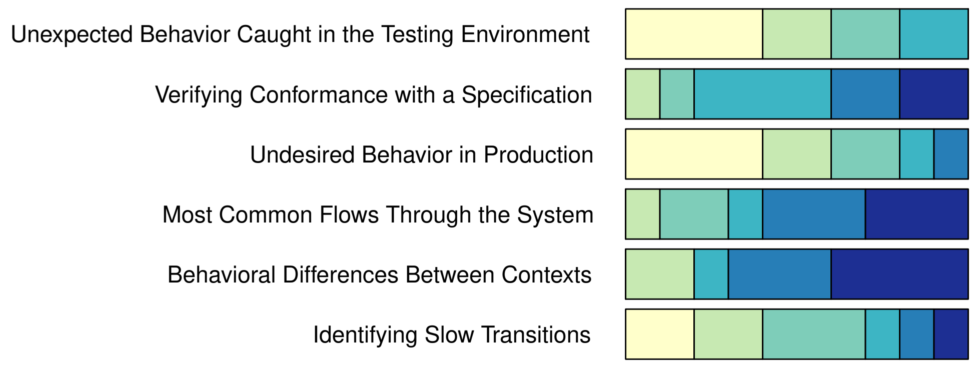


Figure 4.8: Ranking of the findings, as indicated by developers. Yellow colors (on the left) indicate how many responders rank the finding high (important), and similarly blue colors (on the right) indicate how many responders rank the finding low (less important). One of the developers “boldly refused to rank” the findings (as they are all important to him), so this figure illustrates only 10 responses.

4.4.1. Improvements Made to DFASAT

During the iterative analysis, we came up with different improvements to DFASAT[1]. These can be categorized into the addition of trace types, and the addition of more data in the resulting graphs.

Adding Trace Types

To the best of our knowledge, all tools that infer state-machines only have the notion of *accepting* and *rejecting* traces (if they even distinguish between the two). However, in a system with multiple final state types, this does not make sense, as there might be multiple final states that are (un)desired. For example, should a final state ‘Cancelled’ be classified as accepting or as rejecting? This might totally relate to the performed type of analysis. To eliminate this debate, we simply consider each final state as its own type.

Implementation. We exploit the modularity of DFASAT to add our own ‘heuristic’ that contains the changes. That heuristic is based on the *overlap driven* heuristic. As DFASAT only supports two trace types, we start by increasing the number of possible types.

Note that the trace types are embedded in the input file for DFASAT. Thus, if we want to distinguish the types, we need to change our preprocessing logic to put the actual type number (instead of just ‘1’) in front of the trace. This is exactly what we did; the preprocessor maps final states to type numbers. We do not use the ‘rejecting’ type for our input traces, but we fully rely on the newly defined types, to not break the existing heuristics.

Furthermore, we extend the consistency checks to take these new types into account and to make sure that traces of different types do not end up in the same sink. The sinks also indicate the different types, including a unique color per type for an easier distinction. Additionally, we put inside the different nodes the number of traces that reach the node, and we color each node according to the largest type.

The new heuristic produces graphs similar to the underlying heuristic. However, the explicit final states (and their colors) add more details to the graph. The nodes now contain the number of traces that end up in the different final states, as opposed to only the total number of traces. The sinks also have a true meaning, namely ‘all traces in this sink end up in the same final state *x*’.

Adding More Data

Depending on the chosen tool, certain information that is contained in the traces (or that can be computed during the inference process), is not shown in the resulting graphs. For example, some tools choose to only show absolute counts, whereas others only show relative counts. Based on our analysis, we chose to add more information to the graphs: relative frequencies (based on the total number of traces) and event durations.

[1]Available from `https://bitbucket.org/RickWieman/dfasat/branch/overlap4logs`; we use commit 15c13f4.

Implementation. It is fairly easy to compute the relative frequency of a certain event on an edge; the root node of the graph holds the total number of traces in the graph, so that we can easily compute the relative frequency based on that: $\frac{\text{number of traces on the edge}}{\text{total number of traces in the graph}} * 100\%$. Alternatively, it is possible to compute it with the total number of traces in the *source node*; this can be useful to analyze the distribution of the outgoing edges of a node.

Using the relative frequency (based on the total number of traces in the graph) we can change the penwidth of the edges accordingly; edges that occur more often in the data, are thus thicker than edges that occur seldom. Developers can now instantly see where certain final types are more likely to occur, and how many traces end up in a certain final state from one particular point in the graph.

Furthermore, we calculate the time needed to take a transition during the preprocessing, by subtracting the timestamp of the previous event from each event. This assumes that the log lines are sorted chronologically. The first event in the log always gets a duration of 0. After calculating these durations, we add them as *data* to the respective events in the traces. We then use this data in DFASAT to compute for each edge how long the process takes (on average, at least, at most, et cetera).

4.4.2. Taking Context into Consideration

To make the most of the passive learning tools, one needs to tailor the input data. We identified two key approaches that significantly improved our analyses: splitting the data by its context and filtering out data of successful executions.

Splitting by context

One can provide a full set of logs to an inference tool to obtain the ‘total’ model. However, this model might not show certain details, as this is then a ‘general’ overview. For example, if we would take the logs from one application with two different configurations, it might be the case that configuration A has no errors in a certain path, whereas configuration B has many errors on the same path. From this overview graph, one can only conclude that ‘errors occur’ in this path, but by applying *contextual splitting*, it is possible to expose these details. We define contextual splitting as splitting data from a dataset, based on some values residing in the data or related to the data.

Implementation. During the preprocessing (recall Figure 4.1), we are already stripping some information that is too specific. However, some of this information is very suitable for contextual splitting. Examples of such information can be different merchants (i.e., companies that use their payment gateway), or different acquirers (i.e., credit card companies or banks).

Thus, after selecting the information to split, our preprocessor provides different files to be analyzed by the passive learner. The outcome is basically a (potentially) smaller graph which we can then compare among others. Note that we were already applying contextual splitting for our original dataset, as we (1) took the data from one merchant and (2) split the data on different firmware versions.

The key analysis that this contextual split makes possible is comparing paths between different graphs from the same context. For example, is the error frequency for graph A similar to the error frequency in graph B? And from certain points in the graph, are the paths similar, or is the behavior different?

Filtering successful traces

One of the experts from the company pointed out that for him, storing (and analyzing) “success logs” (in our case, logs that confirm a successful transaction) does not make sense, as he is not interested in the normal flows; he only cares about the situations where the system was failing. Thus, we came up with the idea of stripping those success logs from the dataset, so that we can focus entirely on all undesired traces.

Implementation. The implementation is as simple as omitting the approved traces from the input file during the preprocessing phase. This could even go one step further by also filtering non-approved traces that end up in an explanable final state. For example, if the shopper does not have enough balance, the transaction will end up in a decline, but this does not necessarily indicate a system failure.

The result is a graph where each path indicates an anomaly. There are no longer nodes with the ‘success’ color.

4.4.3. A Guide to Adopt Passive Learning

We applied the passive learning tools on the logs of a single company. The results were positive enough that we conjecture that these tools and techniques can as well be applied in any other domain, as long as the system and its logs possess several necessary characteristics and it is possible to follow certain strategies:

1. *The system should provide the full state of a single operation.* In our system, each set of log lines (i.e., every 20 lines) represents a single transaction. Thus, we are able to see the details of a specific transaction, from its beginning to its end. To make these passive learning tools work, it is important that each log file at least follows a similar order of events. If the logs are not sequential nor consistent, the graphs might be complex or even useless for analysis.
2. *Manipulate/Transform the logs.* Some of the tools require a specific format, whereas others can read logs in any format. Nonetheless, be ready to apply some transformations on the input logs to improve the results of the tools. Strip information that is too specific, such as identifiers. Note that some of the information that gets stripped might be useful for splitting by context; use this to compare different graphs in the same context. If the logs originate in different software versions, splitting by version can be a good starting point.
3. *Identify the different final states in the system.* In general, passive learning tools only support accepting and (in some cases) rejecting traces. However, as we have seen, real systems can have more than just these two state types. Taking all different types into account adds significant value to the graphs. However, this will probably require some changes in the selected tool – depending on the tool, this can be relatively easy or difficult.
4. *Pick the tool that fits the logs and suits the purpose.* There are many different tools that all infer in a slightly different way and that produce different graphs. For example, if the number of unique events is small, Synoptic might produce useful graphs. Or when the logs incorporate measurements, MINT [44] could be a good fit. For our logs, DFASAT was the best fit due to its extendability and its fast performance.
5. *Be ready to implement new features in the tools.* Unfortunately, there are no out-of-the-box tools that work immediately for any case. DFASAT is even designed to require some adjustments in order to produce the best results. Furthermore, keep in mind that most of the tools are developed in an academic setting, and thus in general lack essential documentation.

4.5. Threats to Validity

In this section, we discuss possible threats to the validity of this study and how we mitigate each one of them.

Internal validity concerns external factors we did not consider that could affect the variables and the relations being investigated.

1. Most of the findings originate in an iterative process, where we sat with a company expert to analyze and improve the produced graphs. During this process, we modified DFASAT based on the new insights. Thus, the graphs produced at the beginning of this process differ from the graphs produced near the end. Similarly, the maintainers of DFASAT made some modifications themselves. However, the aim was to identify the potential and the possibilities of passive learning, which we think can include tweaking the tools within the boundaries of passive learning.
2. We used different samples of similar data sets throughout the process; different data sets yield different graphs. However, we did stay with the same merchant and the same major version for both terminal platforms, to minimize the impact.
3. The approach we used was exploratory. If we would do the same research with a different expert, the results could be different, as different experts might have different opinions on what is useful (and what

not). Again, the aim was to identify the potential of passive learning; we might not have identified all possibilities yet, but that would not invalidate our current findings.

External validity concerns the generalisation of results.

1. We focused mainly on DFASAT, as this tool had the largest potential in terms of speed and complexity. Applying contextual splitting and removing successful logs could be used for the other tools as well, and the introduction of colors might also be possible. Naturally, stripping traces decreases the input size and will thus improve the performance of the tools; we did not verify the performance impact for the tools.

2. The results presented in this chapter are specific for the Adyen system that produced the logs. However, the *type* of finding (e.g., a bug, a performance issue, the most common flow(s), et cetera) may be generalizable for other systems.

4.6. Conclusion

Based on our findings and experiences, we argue that passive learning tools can indeed help to identify different types of anomalies in a large-scale system, such as bugs, common flows, slow transitions, or differences between contexts. However, there are a few remarks that should be taken into account, as stipulated in Section 4.4.3. The most important one is that the data should be filtered. Specifically, analyzing the data in its context seems to add value.

Furthermore, to reach the full potential of passive learning, the input is as important as the tools themselves. If the logs do not contain the right level of detail, the inferred models might as well lack the detail to be useful for developers or analysts.

5 Comparing Graphs

"The goal is to turn data into information, and information into insight."

— C.C. Fiorina

As we have seen, the graphs inferred using passive learning tend to become huge, especially when the number of unique sequences of events increases. Even for graphs that are still manually analyzable, this poses a new challenge when using these graphs for monitoring or evaluation purposes. For example, to answer the question 'Did the system behave differently last week in comparison to the week before that?', one needs to manually compare the graph from the previous week to the graph from two weeks ago, hoping to notice all differences by hand.

In this chapter, we present a comparison algorithm, which in essence is a recursive depth-first search algorithm. Furthermore, we show some examples of what type of graph differences the algorithm detects and how they will be shown in the resulting graph. We finish this chapter with some examples of graphs inferred from real data: the company's transaction logs.

5.1. Introduction

To allow for comparing graphs in a more automated fashion, we developed a tool to compare different graphs. Not only does this tool facilitate automated graph analysis, it will especially support manual analysis. The idea is simple: it produces another graph that visualizes where the differences occur exactly. Although the approach presented here is centered around the graphs originating in DFASAT, it can probably be applied on graphs in general (albeit with some minor modifications). Essentially, the tool takes two Graphviz dot files and produces another dot file.

Ultimately, this comparison tool helps with monitoring the performance of the system in production; it can potentially remove some load from the person responsible for monitoring the system, as this tool will only highlight the differences. In addition, it might also help to actually notice all differences. With this, we answer the fourth research question in this chapter:

Does a graph difference tool help to identify possible anomalies at our industry partner?

5.2. The Algorithm

The comparison tool basically performs two different steps: the first step is to identify the differences between a *reference* graph and an *observed* graph, in terms of diverging paths. We define the reference graph as the graph that serves as the baseline (intuitively, this can either be some specification graph, or a graph depicting 'previous' behavior), and the observed graph as the graph that shows the 'new' behavior. The second step is then to highlight those paths in the desired output graph, which we refer to as the *difference graph*. Simply put, the tool compares the observed graph to the reference graph, and highlights the identified differences in another graph.

The main advantage of splitting the identification and the highlighting in two steps, is the fact that it allows for printing out (and storing) the affected paths, and the fact that it allows (rather easily) for highlighting those paths in any graph. Typically, this will be the observed graph, but it could also be a combined graph, for example a graph that represents both the reference and the observed graph.

In this section, we describe both steps. First, we discuss the algorithms to compare the graphs, both for structural divergence and numerical divergence. Second, we briefly describe the highlighting of the detected paths in a graph.

5.2.1. Finding Divergent Paths

The intuitive idea behind finding divergent paths is to walk all the paths in the observed graph, and identify whether they exist in the reference graph. This is the *structural* comparison. If a path occurs in both graphs, we can compare the relative number of occurrences in both graphs, to determine whether the frequency changed significantly. This is the *numerical* comparison.

Using this, we can identify the following four types of divergent paths. The first two types are indicators for structural differences in the graphs, and the latter two types are indicators for numerical differences in the graphs. Those differences are detected by two separate algorithms: Algorithm 1 detects type 1 and 2, and Algorithm 2 detects type 3 and 4.

1. **Introduced paths.**
   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.
2. **Missing paths.**
   Paths that exist in the reference graph, but that are missing in the observed graph. This potentially indicates removed functionality, or resolved bugs.
3. **Less frequent paths.**
   Paths that occur in both graphs, but for which the frequency in the observed graph is *lower* than in the reference graph.
4. **More frequent paths.**
   Paths that occur in both graphs, but for which the frequency in the observed graph is *higher* than in the reference graph.

Both algorithms operate on a reference node and an observed node: the root nodes in both graphs are the starting point. Each algorithm will then recurse into all overlapping children; children that do not overlap, indicate a structural difference (either an introduced or missing path) and are stored as such – numerical differences cannot be computed for non-overlapping children. At some point, the intersection $E_{ref} \cap E_{obs}$ will be empty (when a leaf node is reached), which stops the recursion, and thus the algorithm.

Algorithm 1 identifies paths with structural changes (i.e., paths that have newly introduced or missing edges). If the observed node has outgoing edges that are missing in the reference node, these edges are transformed into a path with type Introduced. If the observed node is missing outgoing edges that are present in the reference node, these edges are transformed into a path with type Missing. For outgoing edges that occur in both nodes, the algorithm is called recursively on their destinations. After the recursion, the traversed edge is prepended to the obtained paths, to ensure that the predecessors of a node are listed in the divergent path.

One important observation is that, due to the fact that introduced/missing edges are not traversed, divergent paths go as far as the diverging edge. This is helpful to identify at which point the path diverged, but it prevents the divergent path to be used as a 'full' observation on its own (i.e., the divergent path does not describe the full path from the beginning to the end, but from the beginning to the point of divergence).

Algorithm 2 identifies paths where the (relative) frequency diverges significantly: the edges for which this holds are transformed into a path with type Less Frequent or More Frequent. Thereafter, the algorithm again applies these checks recursively on the (mutual) children. Note that the algorithm relies on the data in the

Algorithm 1 Find structural graph differences starting from a reference node and an observed node.

```
1:  function findStructuralDifferences(Node ref, Node obs)
2:      $result \leftarrow \emptyset$
3:      for all edge $E$ in $E_{obs} - E_{ref}$ do          ▷ Edges that are introduced in the observation
4:          add new DivergentPath($E$, INTRODUCED) to $result$
5:      end for
6:      for all edge $E$ in $E_{ref} - E_{obs}$ do          ▷ Edges that are missing in the observation
7:          add new DivergentPath($E$, MISSING) to $result$
8:      end for
9:      for all edge $E$ in $E_{ref} \cap E_{obs}$ do          ▷ Recursively process mutual children
10:         $childPaths \leftarrow$ findStructuralDifferences(ref.getChild($E$), obs.getChild($E$))
11:         for all $P$ in $childPaths$ do
12:             add $E$ in front of $P$
13:         end for
14:         add all $childPaths$ to $result$
15:     end for
16:     return $result$
17: end function
```

Algorithm 2 Find numerical graph differences starting from a reference node and an observed node.

```
1:  function findNumericalDifferences(Node ref, Node obs, double threshold)
2:      $result \leftarrow \emptyset$
3:      for all edge $E$ in $E_{ref} \cap E_{obs}$ do
4:          $\Delta_{diff} \leftarrow \Delta_{obs} - \Delta_{ref}$
5:          if $|\Delta_{diff}| >$ threshold then          ▷ If the difference is significant enough
6:              if $\Delta_{obs} > \Delta_{ref}$ then
7:                  add new DivergentPath($E$, FREQUENCY_INCREASE, $\Delta_{diff}$) to $result$
8:              else
9:                  add new DivergentPath($E$, FREQUENCY_DECREASE, $\Delta_{diff}$) to $result$
10:             end if
11:         end if
12:         $childPaths \leftarrow$ findNumericalDifferences(ref.getChild($E$), obs.getChild($E$), threshold)
13:         for all $P$ in $childPaths$ do
14:             add $E$ in front of $P$
15:         end for
16:         add all $childPaths$ to $result$
17:     end for
18:     return $result$
19: end function
```

Type	Example	Description
Structural	Introduced edge	The introduction of a new edge in the observation; an edge that did not exist in the reference.
	Missing edge	The removal of an edge in the observation; an edge that did exist in the reference.
	Introduced cycle	The introduction of a new cycle in the observation.
	Different ordering	A swapped order of events in the observation.
Numerical	More frequent	Frequency increase of an edge in the observation.
	Less frequent	Frequency decrease of an edge in the observation.
	Conflicting frequencies	Special case where an edge in the observation has both an increase and a decrease in frequency.

Table 5.1: Overview of all discussed examples. Those seven abstracted examples cover all possible differences between graphs.

graph; if the graph does not contain the number of (relative) occurrences, it simply cannot work. The difference between the two is also stored in the path, so that it can be used as extra information during the highlighting. In this case, to print it on the edges.

Although both algorithms address different types of differences, it is possible to run them sequentially to obtain a set of both structural and numerical divergent paths. This would allow for highlighting all differences between the graphs, instead of just one difference type (i.e., either structural or numerical). However, we think that the difference graphs are easier to analyze if they only address one difference type.

5.2.2. Highlighting Divergent Paths

During the highlighting, all divergent paths are walked on the chosen graph (by default, that is the observed graph), and affected edges are colored accordingly. We call the resulting, colored graph the *difference graph*. If edges from the paths are missing in the difference graph, those are added during the coloring process (i.e., the difference graph is augmented). As this algorithm is straight-forward graph traversal, we do not discuss its implementation details here.

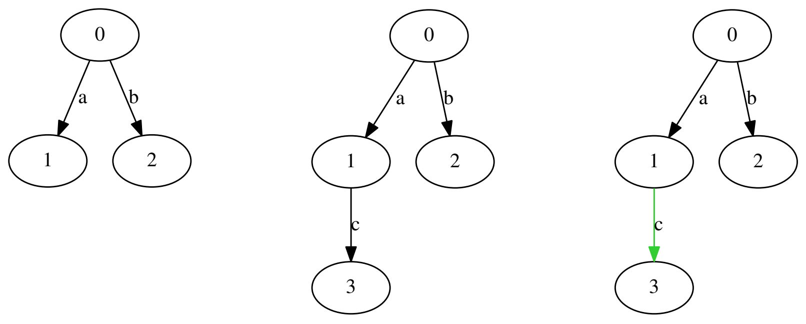
All subsequent edges of an introduced edge are colored, as these all indicate new paths. Recall that divergent paths only go as far as the diverging edge. As a result, only the first diverging edge can be augmented for paths that are missing in the observation. For numerical differences, only the affected edges are colored. Additionally, the difference is added to the label of the edge, as an indicator how large the difference actually is.

Optionally, to obtain a smaller difference graph, all paths that are not affected can be hidden. To be more specific, all paths that contain only black edges from the beginning to the last node are then ignored and thus not printed. Depending on the number of affected edges, this can significantly reduce the size of the difference graph.

5.3. Evaluation with Synthetic Data

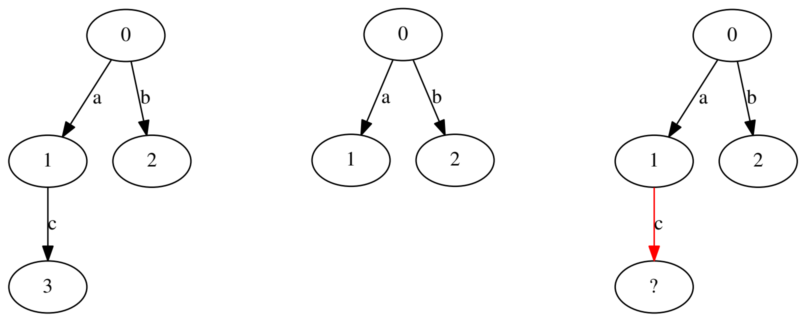
To illustrate the outcome of the tool for different scenarios, we present seven synthetic examples in this section. Each example shows three graphs: the reference graph, the observed graph, and the difference graph. These seven examples cover all possible differences between graphs in an abstract way; any difference that is not directly covered by one of these examples, is very likely covered by a combination of those examples. Table 5.1 shows an overview of all examples, including a brief description of the addressed difference(s).

Note that we did not hide unaffected paths, as these examples are already quite small. Further, showing the full graphs makes the examples easier to comprehend. The numbers inside the nodes of the examples have no meaning for the algorithm. First, we show the identification of structural differences between the graphs. Second, we show the identification of numerical differences, i.e., the differences in frequencies on edges.



(a) Reference (b) Observation (c) Difference

Figure 5.1: Synthetic example of the introduction of an edge. The observation has an additional the possibility of taking *c* after *a*, which is marked with a green arrow in the difference graph.



(a) Reference (b) Observation (c) Difference

Figure 5.2: Synthetic example of a missing edge. The observation misses the possibility of taking *c* after *a*, which is marked with a red arrow in the difference graph.

5.3.1. Structural Differences

There are essentially two types of structural differences: introduced paths and missing paths. We will illustrate those in the first two examples (Figure 5.1 and Figure 5.2). A special case is the introduction of a cycle, which is shown in the third example (Figure 5.3). The disappearance of a cycle is similar (basically a combination of the cycle example and the missing edge example), and is therefore not discussed separately. Another special case is detecting a changed order of events, which is shown in the fourth example (Figure 5.4). All other possible scenarios for structural differences are encapsulated by these four examples. For example, if in the observation the possibility exists to bypass a certain node, this is covered by the introduction of an edge.

Introduced edge. Suppose that a path was introduced in the observed graph: the reference graph only has a path *a* and a path *b* in the starting node, but the observation has an additional *c* after *a*, a path that does not exist in the reference graph. Then, it simply colors this edge (and all its children, if applicable) green, as can be seen in Figure 5.1.

Missing edge. If we swap the reference and the observed graph from this example, we basically get a scenario where the observation is missing a path that existed in the reference. When that happens, it colors the missing edge red. This is illustrated in Figure 5.2. However, as a divergent path goes as far as the affected edge, it cannot show what happens in the reference graph after taking that edge. Simply put, all children that existed after a missing edge, are not printed. This can be seen more clearly in the next example.

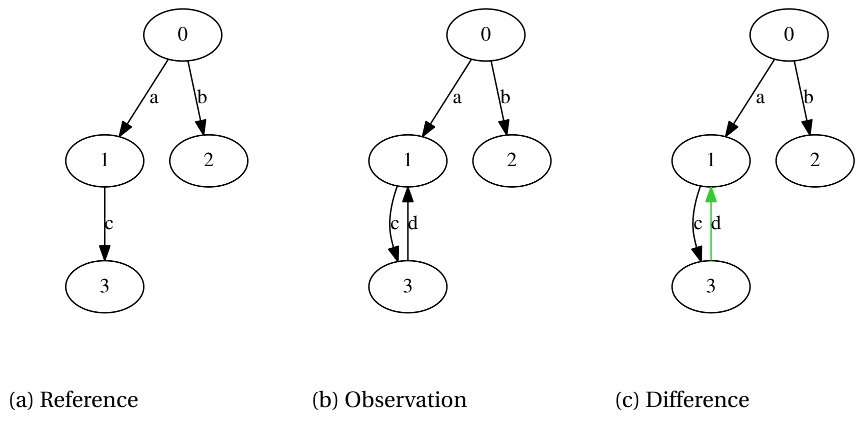


Figure 5.3: Synthetic example of the introduction of a cycle: the observation has an additional d after $a \rightarrow c$. Note that edge c is intentionally not being colored, even though it points to a child.

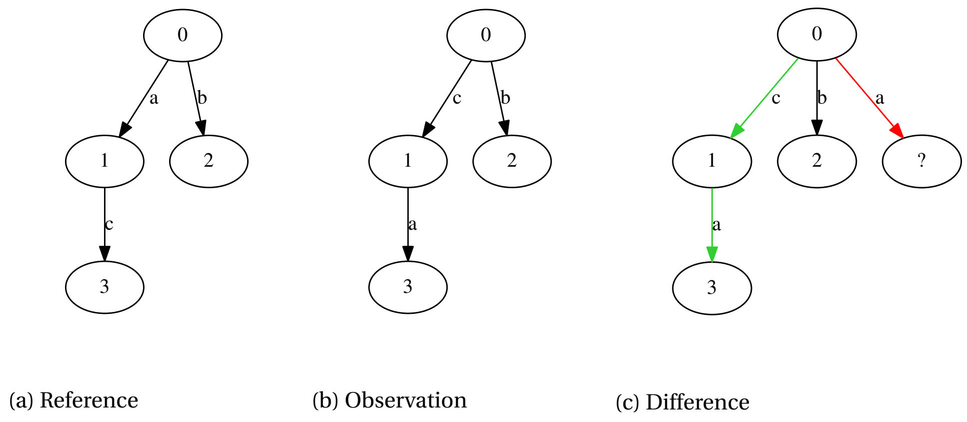


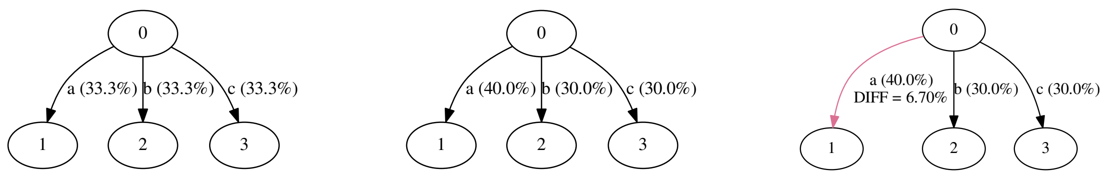
Figure 5.4: Synthetic example of a swapped order. The new order is fully marked green, whereas the missing order is marked red. Note that the red arrow does not have any children by design, as this path is no longer possible.

Introduced cycle. A special case of an introduced edge is an edge that introduces a cycle. In essence, this is just processed like any another introduced edge. This is shown in Figure 5.3. However, in this case, we do not want to highlight the children of the introduced edge, even though it is arguable that all subsequent paths are newly introduced. The main reason is that, if a graph has a cycle to a node close to its root, a large part of the graph might be colored in green, which potentially obscures other highlighted structural differences.

Different order of events. Suppose that we introduce a different order in the observation. For example, the reference has a path $a \rightarrow c$, and the observation has a path $c \rightarrow a$ instead. Then, we basically have two structural differences: we miss the possibility to immediately take a, and we introduce the possibility to immediately take c. Thus, the algorithm will color the a red, and the path $c \rightarrow a$ green, as can be seen in Figure 5.4. The algorithm does not try to find swapped transitions and mark them as a different type (i.e., not introduced nor missing), thus if this would happen in a large graph, there is a chance of getting many green arrows.

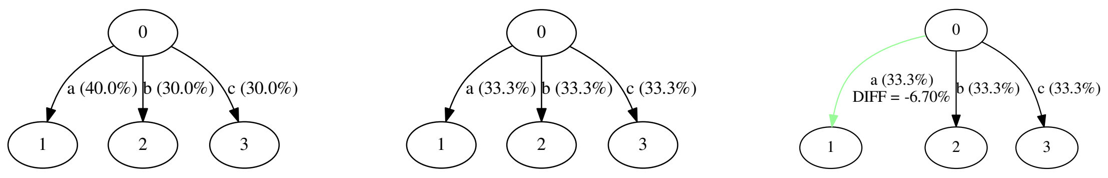
5.3.2. Numerical Differences

There are two types of numerical differences: a path can occur *more* or *less* frequent. This will be covered by the first two examples (Figure 5.5 and Figure 5.6). However, the possibility exists that in the reference graph, two (almost similar) paths occur in different branches, whereas they are merged in the observed graph (or the other way around). The highlighting then might introduce a conflict to the difference graph, which we show in the third example (Figure 5.7). For each example, we set the significance threshold to 5.0%.



(a) Reference (b) Observation (c) Difference

Figure 5.5: Synthetic example of the frequency increase of an edge. The observation now has a higher probability of taking a, which is marked with a red arrow in the difference graph. Note that the frequency difference is also added to the edge label.



(a) Reference (b) Observation (c) Difference

Figure 5.6: Synthetic example of the frequency decrease of an edge. The observation now has a lower probability of taking a, which is marked with a green arrow in the difference graph. Note that the frequency difference is also added to the edge label.

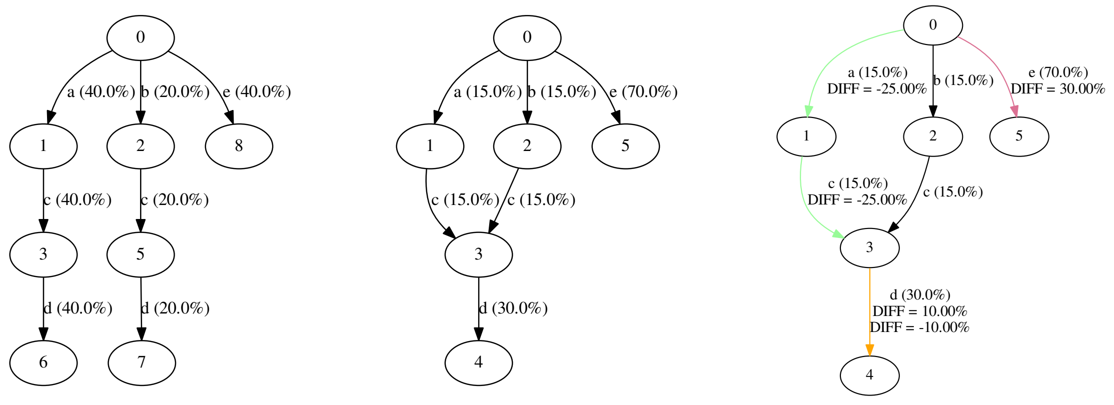
More frequent path. Suppose that the reference graph has a root node, from which a, b, and c can be traversed. In the reference, the respective frequencies are equally distributed (i.e., 33.3%). If we shift the frequencies in the observation in such a way that the difference for at least one of the edges is larger than 5.0% (e.g., 40%/30%/30%), that edge will be colored red. Furthermore, the edge will have an additional label showing the difference (i.e., 40.0% − 33.3% = 6.7%). This is illustrated in Figure 5.5.

Less frequent path. If we again swap the reference and the observed graph from the previous example, we get a scenario where the relative frequency is decreased in the observation. In that case, the edge gets colored green, and it will again have an additional label stating how large the frequency difference is. In this case, that will be 33.3% − 40.0% = −6.7%, as can be seen in Figure 5.6.

Conflicting frequencies. This example is probably the most complicated one, and is partly related to the results of the passive learning tools (in this case, DFASAT). In some scenarios, the reference graph might contain separate branches (e.g., to maintain consistency), but those paths might be merged in the observed graph (when that does not break the consistency). This is illustrated in Figure 5.7. In that case, the algorithm cannot clearly indicate the differences. For the path $a \to c \to d$, the d has a decreased frequency (above the threshold of 5%), but for the path $b \to c \to d$, the d has an increased frequency (above the threshold of 5%). In those cases, the color will be orange, to indicate the conflict. The differences printed in the edge label indicate why it is a conflict.

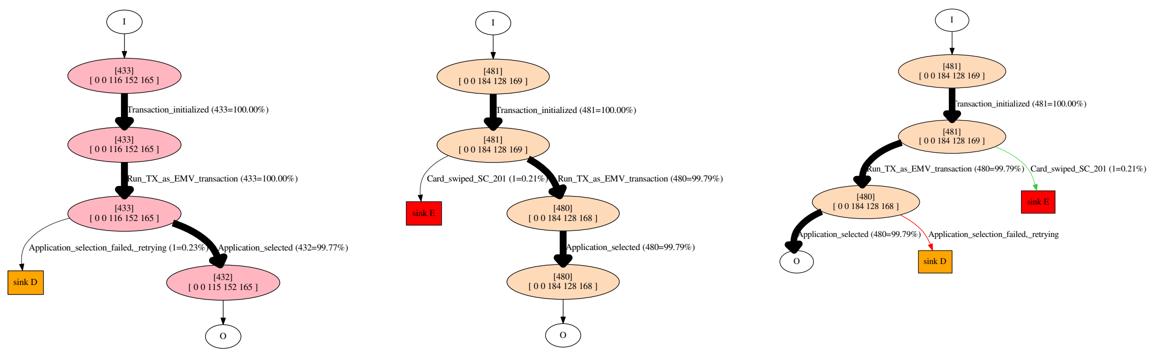
5.4. Evaluation with Real Data

To illustrate the practical usefulness of this tool, we also provide a few examples from real data. As the graphs are still too large to publish entirely, we provide relevant cut-outs instead. In the previous section, we showed examples for all possible differences to demonstrate the tool; in this section, we provide only a few interesting real examples to illustrate the usefulness of the tool, as not all synthetic examples occur in the actual dataset.



(a) Reference (b) Observation (c) Difference

Figure 5.7: Synthetic example of conflicting frequencies. The edge d has a lower frequency for the path $a \to c \to d$, but also a higher frequency for the path $b \to c \to d$. The edge label now holds two differences, which also illustrates the conflict. In addition, the path $a \to c$ has a significant frequency decrease, and the edge e a significant frequency increase. The path $b \to c$ has an insignificant frequency difference (i.e., below the threshold), and is therefore not affected.


I
[433]
[0 0 116 152 165]
Transaction_initialized (433=100.00%)
Run_TX_as_EMV_transaction (433=100.00%)
Application_selection_failed,_retrying (1=0.23%)
Application_selected (432=99.77%)
sink D
[432]
[0 0 115 152 165]
O
[481]
[0 0 184 128 169]
Transaction_initialized (481=100.00%)
Card_swiped_SC_201 (1=0.21%)
Run_TX_as_EMV_transaction (480=99.79%)
sink E
[480]
[0 0 184 128 168]
Application_selected (480=99.79%)
Application_selection_failed,_retrying

(a) Reference (b) Observation (c) Difference

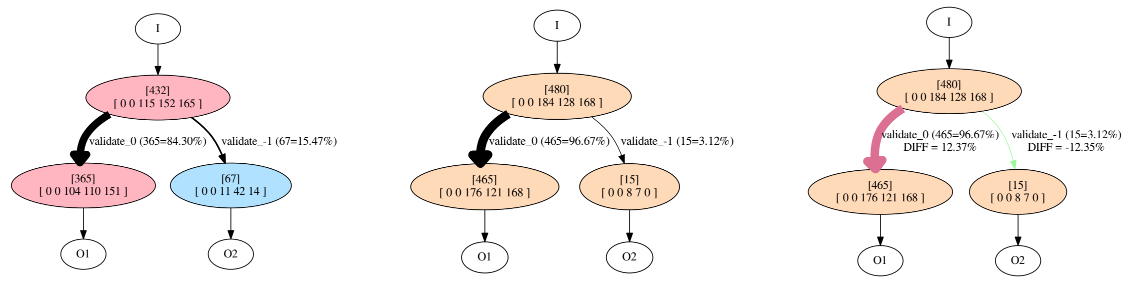
Figure 5.8: Real example of a structural change, of a comparison between two consecutive weeks at the same merchant, for transactions originating in one card type, where approved traces are stripped. The failed application selection disappeared in the observation, but an error caused by a swiped card was introduced.

5.4.1. Structural Differences

Figure 5.8 shows cut-outs of the graphs from a comparison between two consecutive weeks at the same merchant, containing transactions of one card type, on one particular firmware version, where approved traces were stripped from the dataset. One of the transitions disappeared in the observation, and another one was introduced. These are both shown in the difference graph. Although this example is rather trivial, it does illustrate what the difference graph ideally would look like: simple graphs, from which it is easy to derive what changed.

5.4.2. Numerical Differences

Figure 5.9 again shows cut-outs of the graphs from a comparison between two consecutive weeks at the same merchant, containing transactions of one card type, on one particular firmware version, where approved traces were stripped from the dataset. The difference graph clearly shows the decrease in `validate_-1` occurrences, which is actually good (recall Section 4.2.3).



(a) Reference (b) Observation (c) Difference

Figure 5.9: Real example of a frequency change, of a comparison between two consecutive weeks at the same merchant, for transactions originating in one card type, where approved traces are stripped. The number of validate_-1 occurrences dropped significantly (with roughly 12%).

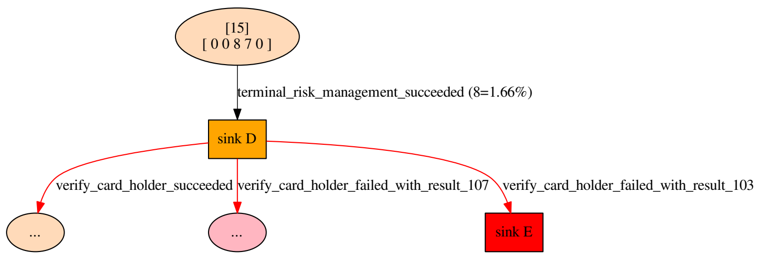

[15]
[0 0 8 7 0]
terminal_risk_management_succeeded (8=1.66%)
sink D
verify_card_holder_succeeded
verify_card_holder_failed_with_result_107
verify_card_holder_failed_with_result_103
...
...
sink E

Figure 5.10: Example where DFASAT introduced a sink in the observed graph. In the reference, there are multiple edges continuing at this point. This difference graph might (wrongly) suggest that verify_card_holder_succeeded never happens in the observed data.

5.5. Discussion: Challenges in Comparing Graphs

We have seen that comparing graphs by hand can be difficult. However, comparing graphs automatically introduces other challenges. The most important remark to make is that the 'difference tool' fully relies on the output from the passive learning tools. Although we like the introduction of sinks in DFASAT, it does not work so well when comparing different graphs. Figure 5.10 shows an example: the reference graph had three branches after `terminal_risk_management_succeeded`, but the observed graph did not. The difference graph might suggest that none of these branches occurs in the observation, although this cannot be said with certainty: we have no indication what happens inside the sinks. It is, in this case, very likely that `verify_card_holder_succeeded` still occurs inside the sink; the other two end up in an error state and will very likely *not* occur inside the sink, as that would make the sink inconsistent.

To solve this issue, it is possible to change DFASAT so that it prints out the full tree hidden in the sink. For this particular example, that would mean that both the reference and the observed graph have the `verify_card_holder_succeeded` edge, but the observed graph would lack the other two edges: only those would then be marked 'missing'. Note that, as a consequence, most graphs produced by DFASAT will be larger. This should have a minimal impact when comparing graphs, especially when the differences between two graphs are minimal.

5.6. Threats to Validity

In this section, we discuss possible threats to the validity of this study and how we mitigate each one of them.

Internal validity concerns external factors we did not consider that could affect the variables and the relations being investigated.

1. We only tried the graph difference tool on a limited set of real data that we selected randomly. How-

ever, by showing the synthetic examples for all possible differences, we argue that we at least cover all scenarios. Perhaps the usefulness of the tool when comparing two totally different graphs decreases, but in such cases, manual analysis would also be difficult. Furthermore, the goal was to *identify* all differences.

2. Although we were looking into the usefulness of such tool, we only evaluated the difference graphs ourselves. To make sure that such tool would really help, a further study (with developers, for example) is required, preferably while using it for a longer time. However, we do have shown conceptually that it would add value to use such tool.

External validity concerns the generalisation of results.

1. To determine numerical differences, the graph should contain information on frequencies. Depending on the chosen passive learning tool/implementation, this frequency might have a different meaning; for us, the frequency is conditional on the *total* number of traces, but it might also be conditional on the number of traces in the *source node*.
2. The approach presented in this chapter only works for graphs that have the information (i.e., symbols and their counts) on the *labels*. To compare graphs similar to the graphs from Synoptic, the algorithm would require some changes, but the conceptual ideas remain the same.

5.7. Conclusion

Based on our findings, we think that a graph difference tool really helps to identify possible anomalies in the system. Especially using manual comparison, one might miss certain differences, that are now highlighted by the tool. We have also seen that as graphs tend to grow with the amount of log data, implementing a graph difference tool resolves a part of the problem of comparing two graphs, provided that the differences are not too large. Finally, we have shown that the algorithms we presented work for all possible differences between graphs (either in terms of structure or in terms of frequencies), and that they also work on our actual data.

6 Conclusions and Future Work

Passive learning has already shown its value in the research field. However, applying it in practice is not as simple as it seems. In this thesis, we presented a case study of the application of passive learning in a large-scale payment system. The main take-away is that tools do not work out-of-the-box; users need to understand their data and manipulate it until the tool is able to generate a useful model. Also, we have seen that current tools contain visualization issues, i.e., they do not offer features that allow users to interact with the results.

We do see a lot of potential for passive learning, and we definitely think that it adds value for Adyen. We will briefly summarize the conclusions and answers for each research question. Furthermore, we provide a list of possible future work, as we still envision much future work to be done.

6.1. Conclusions

What does passive learning bring to Adyen? That was the main question with which we started the thesis. Essentially, we think that it brings many new insights into the performance of their system, as it allows to analyze their transaction logs in a new way. We discovered behavior that the expert was not aware of, and we have shown the potential of passive learning in general. However, we have also seen that adopting passive learning is not as trivial as it may seem, and that there are some hurdles to be taken. To answer the core question more specific, we now answer the four research questions we posed.

RQ1: How do different passive learning tools behave when dealing with extensive log data?

We have seen that the existing open source tools that we selected (namely Synoptic, InvariMint, and DFASAT) struggle to process the amount of data we presented, both in terms of runtime performance and graph complexities. DFASAT has the fastest runtime performance, and produces the least complex graphs. Synoptic runs out of memory for our larger datasets.

RQ2: How do developers perceive passive learning tools and techniques?

From the survey we conducted with ten developers, we learned that they would use passive learning mainly for finding bugs, understanding interactions in the system, and for comparing different versions. However, they do not see it as a way to document software. For all tools, the visualization of the result (i.e., the fact that it is a static image) appears to be the main issue. In addition, they acknowledge what we learned from RQ1, in the sense that they find the grahs from DFASAT the easiest to understand.

RQ3: What types of anomalies do passive learning tools identify in a system of our industry partner?

We identified six types of anomalies that we were able to identify using passive learning, ranging from finding bugs to comparing between contexts, and from most (un)common flows to identifying slow transitions. These findings were very valuable for the company, according to the expert. The main requirement seems to be that the logs follow a certain format, and that they are filtered. Specifically, analyzing the data in its context seems to add value. Furthermore, we noticed that adopting passive learning is not as trivial as one might think, as it requires several (small) improvements to the tools in order to work for a particular use case.

RQ4: Does a graph difference tool help to identify possible anomalies at our industry partner?

We found that a graph difference tool helps a great deal with identifying possible anomalies in the system. Especially as with manual comparison, one might miss certain differences, that are now highlighted by the tool. Specifically, highlighting differences in terms of changed structure and changed frequency appears to be useful, as one can immediately spot the differences (for example, over time, or between contexts).

6.2. Future Work

During the nine months of this project, we were able to identify the usability of passive learning tools, how we should apply them, and how they can add value to a large company like Adyen. However, there are still some remaining ideas to make passive learning even more valuable.

6.2.1. System Monitoring

One of the ideas is to implement passive learning in a monitoring solution. This is illustrated by Figure 6.1: There is a PED under test, which submits logs to some database. We currently pull those logs from the database, apply some preprocessing and feed the outcome of the preprocessor to (different) passive learning tools. If we then look at the visualizations of the results, we can draw interesting conclusions on the behavior of the device under test. If we use production logs instead, we are able to get an understanding of the behavior in a real environment.

If we extend this by feeding all results to a knowledge database that detects anomalies in the behavior, the system can then notify a designated user or developer, who annotates the anomaly. Initially, this could build upon the graph difference tool presented in chapter 5. For example, by computing the difference between two weeks, and sending the difference graph(s) to developers.

Ideally, this monitoring solution would use similar concepts as Logness, a tool built by Evers [20] during a similar project at the same company. Logness is a tool to process log stream data to detect anomalies. The idea is that it clusters messages (e.g., exceptions) with the chosen severity (e.g., warning or critical), and notifies development about these messages when they occur too frequently; a developer can then annotate each notification, so that Logness builds a knowledge base on truly problematic issues and issues that can be ignored. Ultimately, it is able to detect issues that would not be detected by manual monitoring and/or measuring ‘spikes’ of errors.

6.2.2. Automated Test Case Generation

Next to the feedback loop to the developer, Figure 6.1 also shows a feedback loop to the test robot. This illustrates another future goal: to generate test cases based on production behavior, so that the test environment gets/remains as close to the production environment as possible. Naturally, this will require a certain level of detail in the log files: transaction details (such as card brand, amounts, performed CVM, et cetera) in the case of Adyen.

6.2.3. Interactive User Interface

We have seen that all selected passive learning tools lack a user interface (they all produce an image/fixed model), which does not allow the user to interactively analyze the graph. All evaluated tools could benefit significantly from such interface, as it would allow for dynamic filtering, adding more information (which can be shown by hovering over, or clicking on elements in the graph), et cetera. Specifically, the ability to directly show the traces (including identifiers et cetera) that were responsible for a certain flow would be very welcome for analysis purposes. The introduction of such a user interface would probably also remove the urgent need for ‘the smallest graph’, as the infrequent paths can then simply be collapsed. Furthermore, we now have to calculate some key metrics on the time needed for a transition, but a user interface could allow

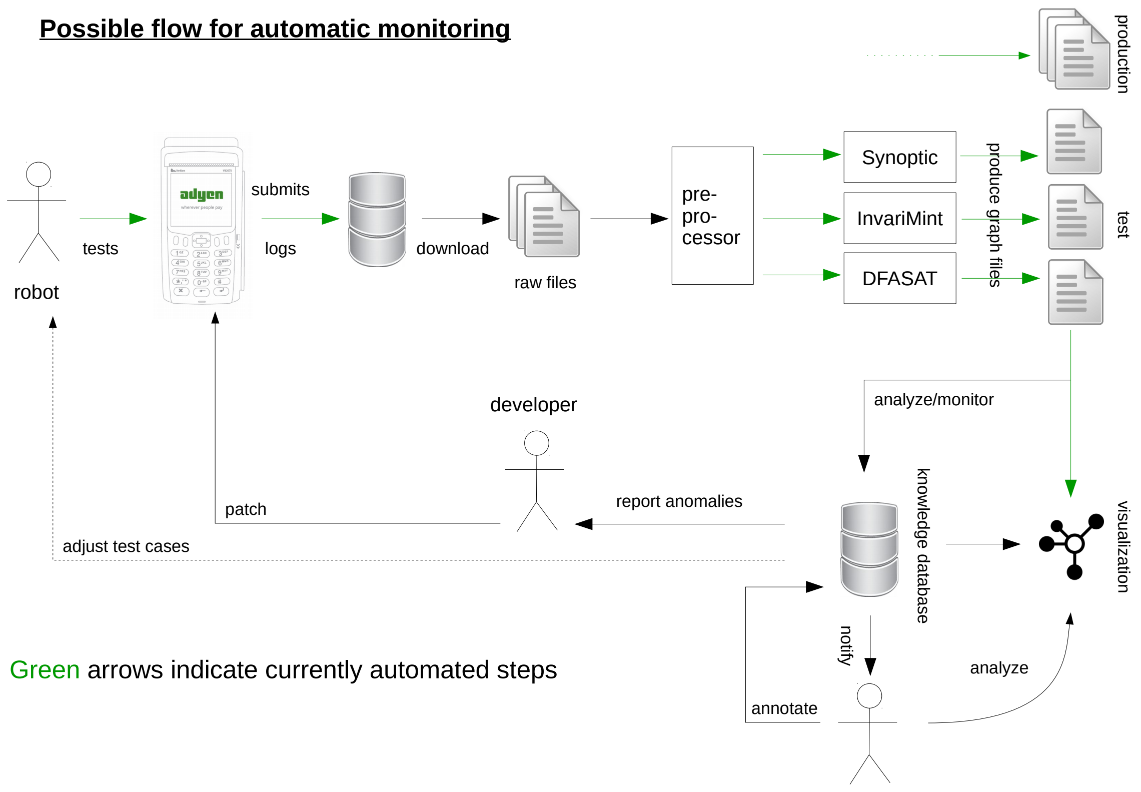


Figure 6.1: Example of how passive learning could be used in a monitoring setup. Log data is processed by passive learning tools, and analyzed by some monitoring application with a knowledge database. Some user can then annotate (possible) anomalies, so that 'accepted' anomalies will not lead to future alerts. Ultimately, the test cases are also adjusted based on the inferred models, to minimize the differences between test and production environments.

for showing the full distribution of those delays. We think that a user-friendly interface would also improve an industrial adoption, as it will likely make the learning curve less steep.

6.2.4. Larger Scale Survey on the Usefulness of Passive Learning

From our surveys with ten and eleven developers respectively, we were already able to draw interesting conclusions on the perceptions and visions of developers. Therefore, we think that performing another survey based on our improvements with a larger group of participants, would add significant value to the research area. Furthermore, it would validate whether our visions and improvements are indeed valuable in other domains as well. This survey could also focus on the usability of the *tools*, i.e., let the respondents interact with the tools themselves.

Acronyms

API	application programming interface
APTA	augmented prefix tree acceptor
ATM	automated teller machine
CVM	cardholder verification method
DCC	Dynamic Currency Conversion
eCom	eCommerce
EMV	Europay, MasterCard and Visa
ICC	Integrated Circuit Card
MKE	manual key entry
MSR	magnetic swipe reader
PED	PIN Entry Device
PIN	personal identification number
POS	point of sale
PSP	Payment Service Provider
SUL	system under learning
XML	Extensible Markup Language

Bibliography

[1] Fides Aarts, Harco Kuppens, Jan Tretmans, Frits W Vaandrager, and Sicco Verwer. Learning and testing the bounded retransmission protocol. In *ICGI*, volume 21, pages 4–18, 2012.

[2] Fides Aarts, Harco Kuppens, Jan Tretmans, Frits Vaandrager, and Sicco Verwer. Improving active Mealy machine learning for protocol conformance testing. *Machine Learning*, 96(1-2):189–224, 2014.

[3] Adyen B.V. Press and media resource page. `https://www.adyen.com/press-and-media`. Accessed 2016-10-18.

[4] Glenn Ammons, Rastislav Bodík, and James R Larus. Mining specifications. *ACM Sigplan Notices*, 37(1): 4–16, 2002.

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

[6] 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.

[7] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D Ernst, and Arvind Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 252–261. IEEE Press, 2013.

[8] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering*, pages 468–479. ACM, 2014.

[9] 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.

[10] Nimrod Busany and Shahar Maoz. Behavioral log analysis with statistical guarantees. In *Proceedings of the 38th International Conference on Software Engineering*, pages 877–887. ACM, 2016.

[11] Adyen B.V. *Functional & Technical Specification C POS*. Adyen B.V., August 2015. Also available on `https://docs.adyen.com/developers/pos-developer-manuals`.

[12] Rafael C Carrasco and José Oncina. Learning stochastic regular grammars by means of a state merging method. In *International Colloquium on Grammatical Inference*, pages 139–152. Springer, 1994.

[13] Hila Cohen and Shahar Maoz. Have we seen enough traces? In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 93–103. IEEE, 2015.

[14] Colin De La Higuera. A bibliographical study of grammatical inference. *Pattern recognition*, 38(9):1332–1348, 2005.

[15] Arianna D'Ulizia, Fernando Ferri, and Patrizia Grifoni. A survey of grammatical inference methods for natural language learning. *Artificial Intelligence Review*, 36(1):1–27, 2011.

[16] EMVCo. EMV Integrated Circuit Card Specifications for Payment Systems - Application Independent ICC to Terminal Interface Requirements, V4.3. Technical report, November 2011.

[17] EMVCo. EMV Integrated Circuit Card Specifications for Payment Systems - Security and Key Management, V4.3. Technical report, November 2011.

[18] EMVCo. EMV Integrated Circuit Card Specifications for Payment Systems - Application Specification, V4.3. Technical report, November 2011.

[19] EMVCo. EMV Integrated Circuit Card Specifications for Payment Systems - Cardholder, Attendant, and Acquirer Interface Requirements, V4.3. Technical report, November 2011.

[20] Peter Evers. Finding errors in massive payment log data. Master's thesis, Delft University of Technology, the Netherlands, 2017.

[21] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Combining model learning and model checking to analyze TCP implementations. In *International Conference on Computer Aided Verification*, pages 454–471. Springer, 2016.

[22] Etienne Gerts. Towards an Improved EMV Credit Card Certification. Master's thesis, Delft University of Technology, the Netherlands, 2007.

[23] Marijn JH Heule and Sicco Verwer. Exact DFA identification using SAT solvers. In *International Colloquium on Grammatical Inference*, pages 66–79. Springer, 2010.

[24] 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.

[25] Martin Leucker. Learning meets verification. In *Formal Methods for Components and Objects*, pages 127–151. Springer, 2007.

[26] David Lo and Siau-Cheng Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *2006 13th Working Conference on Reverse Engineering*, pages 51–60. IEEE, 2006.

[27] RS Mans, MH Schonenberg, Minseok Song, Wil MP van der Aalst, and Piet JM Bakker. Application of process mining in healthcare–a case study in a dutch hospital. In *International Joint Conference on Biomedical Engineering Systems and Technologies*, pages 425–438. Springer, 2008.

[28] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[29] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. Behavioral resource-aware model inference. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 19–30. ACM, 2014.

[30] Michael Pradel, Philipp Bichsel, and Thomas R Gross. A framework for the evaluation of specification miners based on finite state machines. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[31] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. LearnLib: a framework for extrapolating behavioral models. *International journal on software tools for technology transfer*, 11(5):393–407, 2009.

[32] Giles Reger and Klaus Havelund. What is a trace? a runtime verification perspective. In *International Symposium on Leveraging Applications of Formal Methods*, pages 339–355. Springer, 2016.

[33] Joeri de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols*. PhD thesis, Raboud Universiteit Nijmegen, the Netherlands, August 2015.

[34] Wouter Smeenk, Joshua Moerman, Frits Vaandrager, and David N Jansen. Applying automata learning to embedded control software. In *Formal Methods and Software Engineering*, pages 67–83. Springer, 2015.

[35] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. SALSA: Analyzing Logs as StAte Machines. *WASL*, 8:6–6, 2008.

[36] Chris J Turner, Ashutosh Tiwari, Richard Olaiya, and Yuchun Xu. Process mining: from theory to practice. *Business Process Management Journal*, 18(3):493–512, 2012.

[37] Wil Van Der Aalst. Process mining. *Communications of the ACM*, 55(8):76–83, 2012.

[38] Wil MP Van der Aalst and AJMM Weijters. Process mining: a research agenda. *Computers in industry*, 53 (3):231–244, 2004.

[39] Wil MP van der Aalst, Hajo A Reijers, Anton JMM Weijters, Boudewijn F van Dongen, AK Alves De Medeiros, Minseok Song, and HMW Verbeek. Business process mining: An industrial application. *Information Systems*, 32(5):713–732, 2007.

[40] WMP Van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, Heidelberg, Dordrecht, London et. al, 2011. ISBN 9783642193453.

[41] Boudewijn F Van Dongen, Ana Karla A de Medeiros, HMW Verbeek, AJMM Weijters, and Wil MP Van Der Aalst. The prom framework: A new era in process mining tool support. In *International Conference on Application and Theory of Petri Nets*, pages 444–454. Springer, 2005.

[42] Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. Reverse engineering state machines by interactive grammar inference. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 209–218. IEEE, 2007.

[43] 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.

[44] Neil Walkinshaw, Ramsay Taylor, and John Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, 2016.