An Experience Report on Applying Passive Learning in a Large-Scale Payment Company

Wieman, Rick; Aniche, Maurício; Lobbezoo, Willem; Verwer, Sicco; van Deursen, Arie

DOI
10.1109/ICSME.2017.71

Publication date
2017

Document Version
Accepted author manuscript

Published in
Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017

An Experience Report on Applying Passive Learning in a Large-Scale Payment Company

Rick Wieman[1,2], Maurício Aniche[1], Willem Lobbezoo[2], Sicco Verwer[1], Arie van Deursen[1]
{m.f.aniche, s.e.verwer, arie.vandeursen}@tudelft.nl, {rick.wieman, willem.lobbezoo}@adyen.com
[1]Delft University of Technology - The Netherlands, [2]Adyen B.V. - The Netherlands

Abstract—**Passive learning techniques infer graph models on the behavior of a system from large trace logs. The research community has been dedicating great effort in making passive learning techniques more scalable and ready to use by industry. However, there is still a lack of empirical knowledge on the usefulness and applicability of such techniques in large scale real systems. To that aim, we conducted action research over nine months in a large payment company. Throughout this period, we iteratively applied passive learning techniques with the goal of revealing useful information to the development team. In each iteration, we discussed the findings and challenges to the expert developer of the company, and we improved our tools accordingly. In this paper, we present evidence that passive learning can indeed support development teams, a set of lessons we learned during our experience, a proposed guide to facilitate its adoption, and current research challenges.**

***Keywords*-passive learning, experience report, dfasat.**

I. Introduction

The use of log data to analyze the real behavior of a software system in production can provide useful insights to software development teams, such as conformance checking or anomaly detection. However, performing such analysis on a large scale can be challenging: the log entries that are of interest (i.e., log entries pointing towards some anomaly in the system) may be hidden among 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. Clearly, learning the behavior of a system by analyzing each execution trace by hand is simply impossible in large systems. Thus, the use of an automated approach becomes a necessity.

The goal of *passive learning* techniques is to infer graph models on the behavior of the system from large trace logs [34]. Such graph models could then be inspected for different reasons: model checking, error finding, et cetera. Researchers have been working on approaches that would enable passive learning to be used even in large scale, such as selecting representative subsets of log data from large log files [9], and reducing the size of the generated graph by merging similar states [6], [5], [14].

Indeed, a common belief is that these techniques can help companies to identify, among others, whether the behavior of a system in production is correct, or whether the behavior in a test environment matches what happens in production. However, there is a lack of empirical knowledge on how such techniques would behave inside the software development life cycle of a large company and how successful such techniques would be in finding errors.

To that aim, we conducted a research project at a large company in the payment industry. 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 more than 250 payment methods, the company delivers frictionless payments across online, mobile, and in-store channels. With offices all around the world, the company serves more than 4,500 businesses [2].

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 the point-of-sale (POS) solution, which is developed in-house. 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".

We spent nine months performing action research [25], [10] and introducing passive learning techniques in a large-scale system. In this paper, we present our experience report. More specifically, we first present five real examples where passive learning was crucial in providing the team with errors and useful insights. Thereafter, we present the lessons that we learned throughout this period. We then finish the paper by providing a guide for companies to apply passive learning.

The main contributions of this paper are:

1) Concrete evidence of the usefulness of passive learning techniques in a large industry system (Section IV).
2) Lessons learned derived after nine months of action research focused on applying passive learning in a company (Section VI).
3) A guide to facilitate the adoption of passive learning by other companies (Section VII).
4) A list of research challenges that should be tackled by researchers (Section VIII).

II. Background: Passive Learning

Passive learning tools infer state machines from log data [34]. From an input sample of event sequences (logs), they construct an automaton model (state machine) that can produce those sequences.

One of the main challenges faced by such techniques is to produce the smallest possible state machine. However, the problem is NP-hard [13], and inapproximable [22]. Thus, several techniques have been proposed for solving it in practice. The techniques that we use in this research make use of two different techniques: a greedy state-merging method [21], [15] and the k-tails algorithm [6].

State-merging methods start by representing the input sample as a large tree-shaped automaton, called a prefix tree. Every state in this model corresponds to a unique input prefix from the input sample. They then iteratively combine (merge) state pairs by moving all input and output transitions from the second state to the first, and subsequently merging the targets of non-deterministic choices. Only consistent states are merged. When the data is labeled, consistency means that the resulting automaton still assigns the correct label to every sequence from the input sample as in the Blue-Fringe algorithm [15]. When the data is unlabeled, the consistency check is typically replaced with a statistical test whether the probabilities of future event sequences are similar in every pair of merged states as in ALERGIA [8]. State-merging ends when no more consistent merges can be performed. The result is a small automaton that is consistent with the input sample.

The k-tails algorithm also looks at states and future sequences, but only up to a given depth k. Thus, for $k = 1$, it requires only that the immediate events occuring after the combined (merged) states result in the same label (or have similar distributions). For $k = 2$, this holds for futures of length 2, et cetera. The obvious advantage of limiting the consistency test to fixed length futures is that the learning problem can be solved more quickly. The pioneering work that introduced this algorithm even contained a formulation of the problem in constraint programming [6], solving the problem exactly for small k. Decades later a similar formulation was proposed for the full problem (with $k = \infty$) as a satisfiability problem [14]. The disadvantage of using a small k however is that the resulting automaton can be an overgeneralization of the (software) process that generated the data.

We briefly list our selection of three techniques that come with open source implementations: Synoptic, InvariMINT, and DFASAT. There are more tools available that do similar jobs, but that do not fit our datasets. For instance, CSight (short for concurrent insight) [4] analyzes logs from a *distributed* system. As we do not deal with distributed systems, in the sense that all logs are individual and sequential, we do not discuss that tool. Walkinshaw et al. [35] introduce MINT (or EFSM(Inference)Tool). MINT considers data incorporated in execution traces during inference, as events might be related to certain measurements. 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. [20] 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.

Synoptic. Beschastnikh et al. [5] present Synoptic, a tool that aims to make state machine 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. After mining invariants from this graph, it first merges all nodes, and then splits them again so that the newly obtained graph satisfies the invariants. Thereafter, it tries to find the smallest possible automaton by again merging nodes until the invariants are violated. Synoptic models the log events as the nodes of this automaton. Although slightly unusual for state machines, this makes little difference from a representational point-of-view since the log labels can occur in multiple states. The learning algorithm is non-traditional in the sense that it learns from invariants instead of the data sample directly. These invariants are geared towards finding patterns that occur freqently in software development.

InvariMint. The authors of Synoptic also created InvariMint [3]. InvariMint aims at improving understandability of inference algorithms, by describing an approach to model inference algorithms declaratively. Other than Synoptic, InvariMint models log events on edges, and the nodes are empty (a model with hidden state). It ships with a few different algorithms following the declarative approach, among which Synoptic's algorithm and k-tails [6].

DFASAT. DFASAT is a novel tool based on the work from Heule and Verwer [14]. In its core lies a greedy merging algorithm, based on Blue-Fringe [15]. DFASAT takes traces (fixed format) as input, which contain the different events as well as the trace *type* (i.e., accepting or rejecting). It outputs an edge-labeled automaton, just like InvariMint. A key technique used by DFASAT is hiding infrequent paths (i.e., ignoring rare traces and directing them to sink nodes). In contrast, Synoptic and InvariMint use and show all flows in the automaton. In our experience, hiding these paths results in easier to understand models. One key property of DFASAT is its flexibility to model different heuristics and model types. We used the *overlap driven* heuristic, which is based on ALERGIA [8] and was used in DFASAT during the Stamina competition [34].

III. Research Methodology

The *goal* of this study is to evaluate the application of passive learning techniques in a large system and use this knowledge to support both companies that would benefit from such techniques and to provide researchers with a future research agenda on the topic.

To that aim, we spent nine months introducing passive learning techniques, learning from the results and discussing with Adyen's expert developers, and re-iterating. We position this study as *action research* [25], [10]. According to Reason and Bradbury [25], action research is a participatory and democratic process that seeks to bring together action and reflection, theory and practice, in participation with others, in the pursuit of practical solutions.

During nine months, we performed several work iterations. In each iteration, we discussed possible improvements to the use of passive learning at the company with the expert developer, worked on the improvements, and analyzed the new results. In many occasions, different members of the software development team (composed by 20 developers) were also involved in the discussion. We adapted Lewin [16]'s three steps on how to perform action research, which we describe in the following:

1) **Planning.** We had a weekly meeting with the company's expert developer. The expert developer has 15 years of experience as a software developer in this field. These meetings were divided in two phases. In the first phase, we presented the results of our previous interaction, both from the company's point of view (i.e. presenting what we found on their data) and from the research's point of view (i.e. presenting what we learned on passive learning). Commonly, we presented large printed graphs generated by the tools and asked for the expert to help us interpret it. Thereafter, we discussed the next steps. The next steps were also discussed from the two perspectives. From the company's perspective, in many cases, we focused on better understanding a possible problem found; from the research's perspective, we focused on how to make the techniques better and more accurate.
2) **Action.** Most of the times, executing what was planned meant improving or tuning the existing passive learning tools. In particular, after a few experiments, we chose DFASAT as the tool to customize. All our changes and improvements are available as open source [36].
3) **Results.** In our case, results mean revealing some new useful information to the development team; something that was hidden in the log data before. In many cases, the developers responsible for the system actually deeper investigated our finding into their source code bases, and improved their systems accordingly.

We made sure to take notes about our meetings with the expert as well as about all our lessons learned. At the end of our journey, we grouped everything we learned and observed into the five main sections of this paper: in Section IV, we present the real cases in which passive learning provided new information to the team; in Section V, we present the improvements we made to one of the current state-of-art passive learning tools; in Section VI, we discuss the lessons we learned while applying passive learning; in Section VII, we present a guide to facilitate the adoption of passive learning; and finally, in Section VIII, we discuss research challenges that should be tackled by researchers.

IV. Evidence of the usefulness of passive learning

In this section, we present five different real case examples in which the use of passive learning was instrumental in revealing useful information to the software development team:

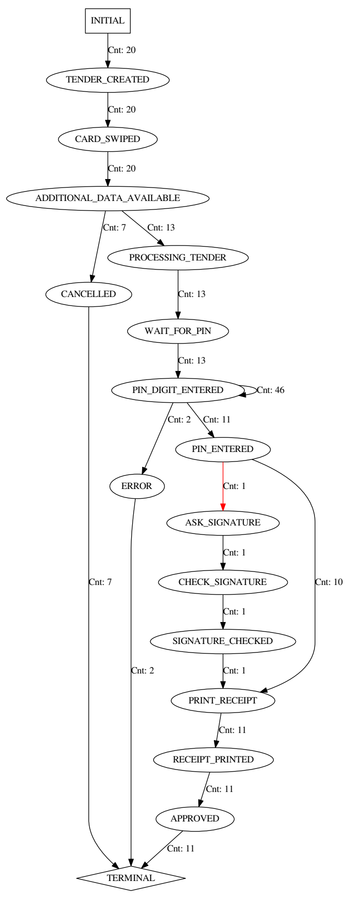


Fig. 1: Graph model from a new firmware version. The red arrow indicates erroneous behavior, as the PIN was already entered (in these tests, one *CVM* is sufficient).

1) Discovering a bug in the testing environment (Section IV-A)
2) Finding non-conformant behavior when compared to the official specification (Section IV-B).
3) Revealing undesired behavior in the system (Section IV-C).
4) Comparing the same state machine in different contexts, e.g., payment over different card brands (Section IV-D).
5) Identifying slow transitions in the system (Section IV-E).

A. *Unexpected Behavior Caught In A Testing Environment*

We applied passive learning on logs from the testing process, where a new firmware version was being tested. Specifically, these tests focused on swipe transactions, where *PIN* entry (as opposed to putting a signature) is required. The resulting graph is shown in Figure 1.

From this graph, one of our test automation engineers was 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.

B. (Non-)conformance with the Specification

The company expert wanted to verify the behavior of the system against a specification. To investigate the usability of passive learning for this, we took the *EMV* specification [11] (developed by EMVCo, a payment authority), and compared 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.

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. 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.

C. Revealing Undesired Behavior

Certain calls to the online platform should never fail. 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 can be identified 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 2. 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.

D. Behavioral Difference Between Two Different Card Types

We took a large transaction log dataset from one merchant on production with the goal of understanding whether there was a different behavior between two different card brands. We then generated two state machines, one for each card brand.

From the two graphs (subgraphs of both are shown in Figure 3), we could almost immediately discover several

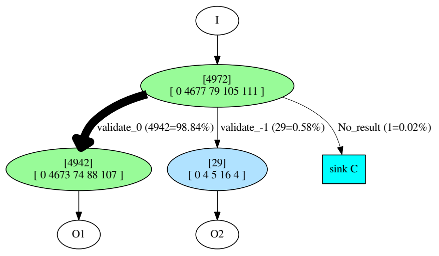


Fig. 2: Example of revealed undesired behavior in the system.

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.

E. Identify Slow Transitions

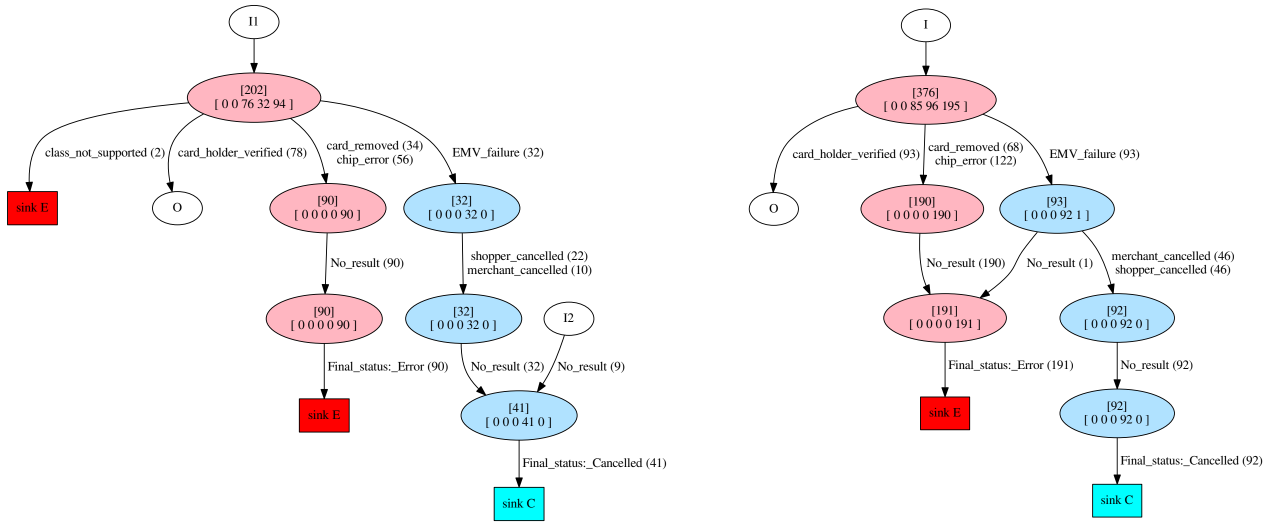
Some parts of the system can take more time to execute than others. In a state machine, this is represented by a slow transition between two states. Highlighting the edges with a long duration makes it easier to identify which transitions need more time. This is specifically useful to find the time-related bottlenecks in the system.

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). Figure 4 shows the same graph as Figure 2, with the timings added to the 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.

Impact. 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).

V. Modifications to DFASAT

During our 9 month study, we made several modifications to DFASAT that improved its applicability. DFASAT allows the user to create custom heuristics and state machine types by adding a single file to the code and recompiling. We used the overlap driven heuristic as a base class, and added a new heuristic using 150 lines of code, and another 150 for new vizualization routines. Our modifications are available as open source, we list the four most influential ones below.



(a) Type A

(b) Type B

Fig. 3: Comparison between two different card types for one particular part of the flow. Nodes I illustrate the prior graphs for this part, nodes O the remainder of the graphs. 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.

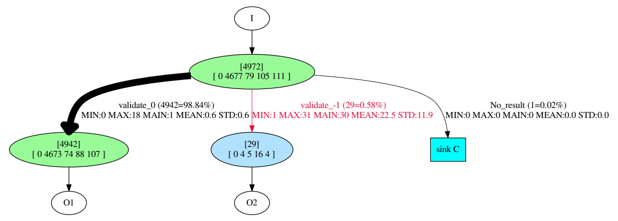


Fig. 4: Example of a slow transition in the system.

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? We therefore treat each final state as its own type.

New sinks and colors. Besides extending the consistency checks to take these new types into account, we also implemented new sinks based on the final state types. Whenever a state is reached only by traces of a certain type, it is replaced by the corresponding sink node. In the resulting dot file, we color the sinks differently for the different types, making visual distinction easier. Similarly, we color each node according to the most frequent type. Figures 2, 3, and 4 show the result of these modifications.

Additional consistency check. The first months of our study, we almost continuously learned models in order to fine-tune the parameter settings. We noticed however that DFASAT sometimes performed merges that we consider to be wrong. It can create self-loops when a logged event does not influence the future behavior. For instance, given inputs $abccc$ and $accc$, it may conclude that the behavior after a and ab is similar, and merge the states reached by a and ab. This introduces a self-loop with label b to the state reached by a. Although this may be correct in theory, visually it gives the wrong impression that multiple b events can occur after an a event. We added a check that avoids creating such loops.

Adding more data. We added more information to the dot files: relative frequencies and event durations. The relative frequencies are computed using the number of traces on the edges divided by total number of traces in the graph. Using the relative frequency, we modify the width of edges, such that those that occur more often in the data are thicker. Furthermore, we calculate the time needed to take a transition during the preprocessing, by subtracting the timestamp of the previous event from each event (assuming log lines are stored chronologically). We then compute for each edge how long it takes on average and vizualize this in the dot file.

VI. Lessons Learned

In this section, we present four lessons we learned during our nine months:

1) Different tools present different results (Section VI-A).
2) Tools need customization before being applied in real settings (Section VI-B).
3) Take the context into consideration (Section VI-C).
4) Developers want to mostly focus on finding bugs (Section VI-D).

	# 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 I: Overview of log sizes per time interval, including the number of unique sequences and events

A. Lesson 1: Different tools present different results

As tools allow for custom configurations that are likely to influence their performance and output, we wanted to understand how each tool behaves in different datasets.

We experimented with the three tools on four different log datasets, all originating in 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 the company 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 are listed in Table I. 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.

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.

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.

Runtime performance. The average runtimes of the ten executions are shown in Table II, which is visualized in Figure 5. We 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[1] (open source graph visualization software) needs to process and output the resulting graph in PNG format, as Synoptic and InvariMint have this functionality embedded.

[1]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 II: Runtime comparison of the three tools, running on the different datasets. Runtimes are indicated in milliseconds, as the average of ten sequential runs.

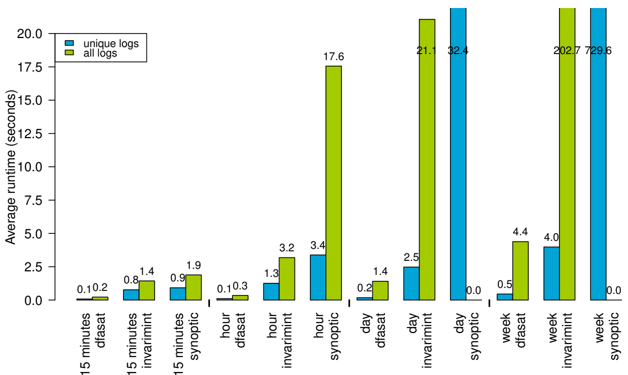


Fig. 5: Plot of the runtime comparison of the three tools, running on the different datasets.

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.

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 $E - N + 2P$ [19]. 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 graphs more difficult, whereas we want to analyze the graphs by hand. In Table III and in Figure 6, we show a comparison of the number of nodes and edges for each of the datasets from the different tools.

The exact numbers listed in Table III also include 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,

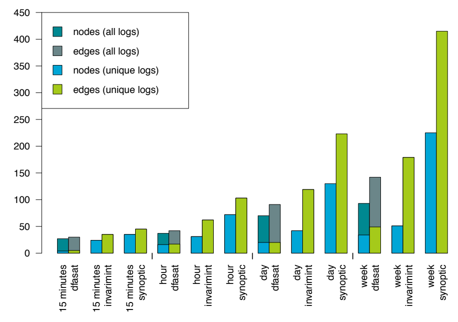


Fig. 6: Complexity comparison between Synoptic, InvariMint and DFASAT

that same state might occur multiple times, so that it actually *is* being considered and thus influences the graph.

Developers perceptions. We show the developers one graph from each of the tools (in a varying order), where each graph originates in the full set for one hour, presented in Table I. 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).

We sent the survey to 10 of our developers. 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.

We see that DFASAT's results were considered the easiest ones to be understood by developers. 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.

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.

B. Lesson 2: Tools need customization before being applied in real settings

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.

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 yet well 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.

C. Lesson 3: Take the context into consideration

One can provide a full set of logs to an inference tool to obtain the "full 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.

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*

		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 III: Complexity comparison between Synoptic, InvariMint, and DFASAT. The columns list the number of nodes (N), the number of edges (E), the number of cycles (C) and the cyclomatic complexity (CC).

similar to the error frequency in graph B? or *from certain points in the graph, are the paths similar or is the behavior different?*

Implementation. During the preprocessing, 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.

D. Lesson 4: Developers want to mostly focus on finding bugs

In the same survey, we asked 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 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?
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.

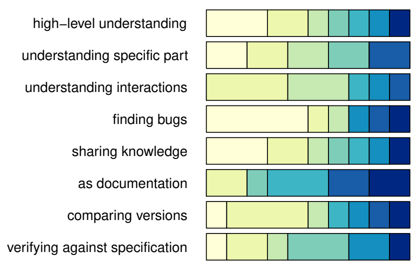


Fig. 7: Overview of purpose rankings. Yellow colors (on the left) indicate how many responders are likely to use the tools for that purpose, and blue colors (on the right) indicate how many responders are very unlikely to use them.

VII. 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:

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.

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.

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.

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 [35] could be a good fit. For our logs, DFASAT was the best fit due to its extendability and its fast performance.

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.

VIII. Research Challenges

During our study, we encountered many points where passive learning tools should be improved in order to be adopted by industry. Out-of-the-box, passive learning tools provide the means to learn models from data, but developers find it often hard to interpret and use the learned models. Models with smaller graph vizualizations are easier to understand. For instance, DFASAT's use of sink states is appreciated by developers. However, developers also mentioned that they did not understand the meaning of sink states, making it again hard to interpret. Why does DFASAT ignore possibly important errors by putting them in low frequency sink states? This is not easy to explain since it is actually good that the state-merging algorithm does not merge these potentially important error states with similar states, potentially losing track of the important error.

This (mis)interpretability of passive learning tools highlights an important challenge for future research. Most passive learning tools are developed from a researchers perspective, and improvements are usually made on the type of models being learned. In this paper we show that very simple state machine models (see, e.g., Figure 1) are already very useful in practice. In essense, there is a mismatch between what the tool developers optimize (building better models) and what the developers want (to find bugs). We believe bad models can also be effective at finding bugs when presented in the right way. In our experience, the key open research challenges for tool adoption are:

1) The development of an interactive vizualization of learned state machines that is useful for developers to locate bugs.
2) Studying how to integrate what developers know (such as code) with the learned models and algorithms in order to improve understandability.

IX. Related Work

Process mining. Similarly to passive learning, process mining [30], [28] 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 [31], a large tool that contains several plug-ins for performing process mining and analyses on that. Process mining has been applied in a variety of sectors and organizations, such as municipalities, hospitals and banks. Van der Aalst et al. [29] did a successful case study on applying process mining at the Dutch National Public Works Department (Rijkswaterstaat). Similarly, Mans et al. [18] 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.

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 [17] 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. [33] 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. [23] present a framework to evaluate how accurate mined specifications of API usage constaints are. Busany et al. [7] 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.

Active learning. Opposed to passive learning, active learning actively queries the system under learning to get the responses to different inputs, and that inferred machines are verified against that system. In recent work, active learning is commonly used for reverse engineering [32], and to verify an implementation against some specification [1], [12]. For example, Ruiter [26] used LearnLib [24] (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. In addition, Smeenk et al. [27] infer the behavior of an embedded control device in a printer, and discover several deadlocks in that system.

X. Conclusion

We report on an industrial study of passive learning tools performed at Adyen. Our study indicates that passive learning is very useful in industry. Most importantly, it can be used to discover bugs and undesired behavior in Adyen's point-of-sale devices. Although our study resulted in quite some impact in the point-of-sale development, we identified several shortcomings in all tested passive learning tools that limit their usability in practice. Most importantly, developers found it hard to fully understand the models returned by the tools.

Although we developed modifications to DFASAT that try to overcome some of these shortcomings, they are only a small step towards improving the user experience of passive learning tools. There is much more work to be done. For example the inclusion of developer's knowledge in the algorithm and its output, links to code responsible for certain errors to make debugging easier, improved vizualizations, et cetera. Based on developer interviews, we identified two key open research challenges for the adoption of passive learning techniques by industry. In the near future, we expect to see more industrial studies of passive learning, hopefully making the gap between the tools and industry smaller and smaller.

References

[1] F. Aarts, H. Kuppens, J. Tretmans, F. W. Vaandrager, and S. Verwer. Learning and testing the bounded retransmission protocol. In *ICGI*, volume 21, pages 4–18, 2012.

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

[3] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 252–261. IEEE Press, 2013.

[4] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. 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.

[5] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. 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.

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

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

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

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

[10] J. W. Creswell and V. L. P. Clark. Designing and conducting mixed methods research. 2007.

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

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

[13] E. M. Gold. Complexity of automaton identification from given data. *Information and control*, 37(3):302–320, 1978.

[14] M. J. Heule and S. Verwer. Software model synthesis using satisfiability solvers. *Empirical Software Engineering*, 18(4):825–856, 2013.

[15] K. J. Lang, B. A. Pearlmutter, and R. 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.

[16] K. Lewin. Group decision and social change. *Readings in social psychology*, 3:197–211, 1947.

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

[18] R. Mans, M. Schonenberg, M. Song, W. M. van der Aalst, and P. J. 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.

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

[20] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun. Behavioral resource-aware model inference. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 19–30. ACM, 2014.

[21] J. Oncina and P. Garcia. Identifying regular languages in polynomial time. *Advances in Structural and Syntactic Pattern Recognition*, 5(99-108):15–20, 1992.

[22] L. Pitt and M. K. Warmuth. The minimum consistent dfa problem cannot be approximated within any polynomial. *Journal of the ACM (JACM)*, 40(1):95–142, 1993.

[23] M. Pradel, P. Bichsel, and T. 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.

[24] H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *International journal on software tools for technology transfer*, 11(5):393–407, 2009.

[25] P. Reason and H. Bradbury. *Handbook of action research: Participative inquiry and practice*. Sage, 2001.

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

[27] W. Smeenk, J. Moerman, F. Vaandrager, and D. N. Jansen. Applying automata learning to embedded control software. In *Formal Methods and Software Engineering*, pages 67–83. Springer, 2015.

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

[29] W. M. van der Aalst, H. A. Reijers, A. J. Weijters, B. F. van Dongen, A. A. De Medeiros, M. Song, and H. Verbeek. Business process mining: An industrial application. *Information Systems*, 32(5):713–732, 2007.

[30] W. M. Van der Aalst and A. Weijters. Process mining: a research agenda. *Computers in industry*, 53(3):231–244, 2004.

[31] B. F. Van Dongen, A. K. A. de Medeiros, H. Verbeek, A. Weijters, and W. M. 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.

[32] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse engineering state machines by interactive grammar inference. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 209–218. IEEE, 2007.

[33] N. Walkinshaw, K. Bogdanov, and K. Johnson. Evaluation and comparison of inferred regular grammars. In *International Colloquium on Grammatical Inference*, pages 252–265. Springer, 2008.

[34] N. Walkinshaw, B. Lambeau, C. Damas, K. Bogdanov, and P. Dupont. Stamina: a competition to encourage the development and assessment of software model inference techniques. *Empirical software engineering*, 18(4):791–824, 2013.

[35] N. Walkinshaw, R. Taylor, and J. Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, 2016.

[36] R. Wieman. Improvements to dfasat. https://bitbucket.org/RickWieman/dfasat/branch/overlap4logs.