Improving Model Inference in Industry by Combining Active and Passive Learning

Nan Yang*, Kousar Aslam*, Ramon Schiffelers*†, Leonard Lensink‡,
Dennis Hendriks‡, Loek Cleophas*, Alexander Serebrenik*
* Eindhoven University of Technology, The Netherlands
† ASML, The Netherlands
‡ ESI (TNO), The Netherlands

***Abstract*—Inferring behavioral models (e.g., state machines) of software systems is an important element of re-engineering activities. Model inference techniques can be categorized as active or passive learning, constructing models by (dynamically) interacting with systems or (statically) analyzing traces, respectively. Application of those techniques in the industry is, however, hindered by the trade-off between learning time and completeness achieved (active learning) or by incomplete input logs (passive learning). We investigate the learning time/completeness achieved trade-off of active learning with a pilot study at ASML, provider of lithography systems for the semiconductor industry. To resolve the trade-off we advocate extending active learning with execution logs and passive learning results.**

We apply the extended approach to eighteen components used in ASML TWINSCAN lithography machines. Compared to traditional active learning, our approach significantly reduces the active learning time. Moreover, it is capable of learning the behavior missed by the traditional active learning approach.

***Index Terms*—reverse engineering, model inference, passive learning, active learning, runtime logs, equivalence oracle**

I. Introduction

Unlike the traditional software development process, Model-Driven Software Engineering (MDSE) uses models as the main software artifacts. MDSE promises that presence of models will facilitate early verification of correctness and hence earlier defect detection, reducing development cost [12].

In order to benefit from the promises of MDSE, existing software systems have to be migrated. To tackle this problem, model inference techniques have been proposed in the literature. These techniques infer behavior using a running system, a so called SUL (system under learning), rather than modeling from scratch. The inferred models (e.g. state machines) can then be verified, simulated, transformed or used to generate new code.

Model inference techniques can be categorized into active and passive learning techniques. *Active learning* techniques [2], [16], [26] are based on the query-response mechanism. They iteratively interact with a running system by sending inputs (queries) and observing outputs (responses), and infer hypothesized models based on the interactions. Such techniques guarantee to learn the complete behavior under the assumption that the counterexamples differentiating the hypothesized model from the system can be found via conformance testing. However, as discussed by Vaandrager [34], the required number of test sequences grows exponentially with the size of the system. Executing such a large set of test sequences is very time-consuming. In practice, the learning process has to be stopped at some point. In such a case, one can never be sure whether the learned model represents the complete behavior of the running system. Hence, application of active learning in practice induces a trade-off between efficiency and behavioral coverage.

Passive learning techniques [7], [37], [42] infer models from a set of execution logs. Since the logs correspond to a limited number of use cases, the learning results are also incomplete [11], [40]. Moreover, these techniques introduce overapproximation [8], [18], [19] making it hard to learn the complete behavior of the system.

To get a better understanding of how testing hinders active learning to scale in real settings, we conducted an exploratory pilot study at ASML, provider of lithography systems for the semiconductor industry. We applied active learning to 218 components from the TWINSCAN lithography machine and observed that when the total active learning time increases, the learning is dominated by the testing time. This observation confirms the theory of Vaandrager [34]. Moreover, we also find that it is particularly hard to learn the complete behavior of systems where earlier choices restrict the behavior much later on. We name the early choice behavior problem as *far output distinction behavior*. Indeed, active learning requires counterexamples that distinguish the hypothesized model from the system, to achieve completeness. For systems with the far output distinction behavior, the counterexample is a long sequence of inputs capable of reaching the system states where different outputs can be observed. It requires a lot of time for the conformance testing algorithms to explore all possible input sequences of a certain length and find the counterexamples. The far output distinction behavior is one of the reasons why in our pilot study active learning could not finish learning all the 218 components within 1 hour. By inspecting these components we find that the missing behavior occurs frequently during system execution. Artifacts created during system execution, such as logs, and subsequently passive learning results obtained from them, can thus be expected to contain this missing behavior. Hence, additional information derived from logs or passive learning results is expected to speed up finding the counterexamples, improving the efficiency of active learning.

Based on the pilot study we explore *whether active and passive learning can be combined to improve the efficiency of learning, while guaranteeing a certain minimum behavioral coverage*. From the passive learning perspective, active learning can be used to find the exceptional behavior that is not captured by the execution logs. For active learning, the logs and passive learning results can be used to learn the behavior efficiently. A certain minimum behavior coverage can be guaranteed; the observed behavior captured by the execution logs will be included in the learned result. To evaluate whether combining active and passive learning can improve the efficiency of active learning, we applied the combined approach to 18 components from 218 components of our pilot study. We observe that active learning finishes significantly faster and results in complete behavior. In particular, the combined approach helps to distinguish states that were hard to distinguish with the existing setup, without exhaustively exploring all combinations of input actions state by state.

The main contributions are the investigation of the scalability of active learning and the causes of time-consuming testing (the pilot study), and an improved active learning technique that integrates execution logs and results of passive learning.

Outline. After discussing the background in Section II, we present the pilot study in Section III. Then we introduce the combined approach in Section IV and evaluate it on the industrial case study in Section V. Finally, we discuss related work and conclude our paper in Sections VI and VII.

II. Background

Most model inference techniques learn state machines. Several algorithms [2], [7], [16], [42] have been proposed over the years to implement active and passive learning. Since we focus on the conceptual weaknesses of active and passive learning, we introduce only the generic concepts underlying the algorithms. For a more detailed explanation of different algorithms, the reader is referred to the paper by Vaandrager [34] for active learning, and the one by Stevenson et al. [32] for passive learning. Below we introduce the necessary concepts and definitions that are used in the paper.

A. State machines

Definition 1 (Mealy machine): A Mealy machine is a tuple $\mathcal{M} = \langle S, \Sigma, \Omega, \rightarrow, \hat{s} \rangle$, where S is a set of states, Σ is a set of input actions, Ω is a set of output actions, $\rightarrow \subseteq S \times \Sigma \times \Omega \times S$ is a transition relation and $\hat{s} \in S$ is the initial state.

Definition 2 (Deterministic Finite Automaton): A Deterministic Finite Automaton is a tuple $\mathcal{DFA} = \langle S, \Sigma, \rightarrow, F, \hat{s} \rangle$, where S is a set of states, Σ is a set of input actions, $\rightarrow \subseteq S \times \Sigma \times S$ is a transition relation, $F \subseteq S$ is a set of accepting states, and $\hat{s} \in S$ is the initial state.

Given a set of input traces, a prefix tree acceptor $\mathcal{PTA}$ is a tree-like $\mathcal{DFA}$ (a.k.a. trie automaton) where each input trace in the set is represented by a path from the initial state to an accepting state, and no state has multiple incoming transitions.

Example 1: The Mealy machine in Fig. 1 implements functions a_i and b_i with return values a_o and b_o, respectively. The $\mathcal{PTA}$ corresponding to the set of execution traces $\{a_i a_o a_i a_o b_i b_o b_i b_o, a_i a_o b_i b_o\}$ is shown in Fig. 2.

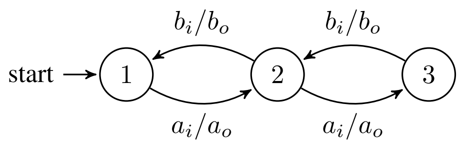


Fig. 1: A Mealy machine of a SUL

B. Completeness of learning results

Completeness requires that the learning results contain all the behavior that is allowed by the SUL and nothing more than that. Completeness can hence be violated by overapproximation or underapproximation. Overapproximation means that the learned model allows behavior that is not allowed by the SUL. Underapproximation indicates that some of the behavior of the SUL is absent from the learned model.

Example 2: The model in Fig. 3(a) overapproximates the SUL from Fig. 1. In fact, this model allows any sequence of inputs. The model shown in Fig. 3(b) underapproximates the SUL. It misses the occurrence of input sequence $a_i a_i b_i b_i$ which is allowed by the SUL. The model shown in Fig. 3(c) both overapproximates and underapproximates the SUL. This model allows input sequence $a_i b_i b_i$ which is not allowed by the SUL, while it disallows $a_i a_i b_i b_i$ present in the SUL.

C. Active learning

In 1987, Angluin proposed the L* algorithm which implements a well-known active learning framework [2]. The original L* algorithm was designed to construct DFAs and was later adapted to learn Mealy machines, enabling learning for reactive I/O systems [21]. The active learning framework (Fig. 4) assumes the presence of a *teacher*, which consists of the SUL and an equivalence oracle. It also assumes that the SUL can be represented by a regular language.

When learning, the learner iteratively executes two steps. In the *first* step, the learner asks membership queries (MQs) to the SUL to obtain output sequences in response to input sequences. For example, for the SUL represented in Fig. 1, output sequence $a_o b_o$ is the response for input sequence $a_i b_i$. The learner then proposes a hypothesis model in the form of a Mealy machine, based on the output sequences. In the *second* step, the learner verifies the correctness of the derived hypothesis model by posting the hypothesis as an equivalence query (EQ) to the equivalence oracle; the oracle uses test queries (TQs) to check the equivalence with respect to the SUL. A TQ, similar to a MQ, checks whether the system's response to an input sequence agrees with the response expected from the hypothesis. If there is a mismatch between the responses to the TQ from the hypothesis and the SUL, then the input sequence is considered as a counterexample. Based on the counterexample, the learner refines the hypothesis with further MQs. The learning process continues until the equivalence oracle cannot find a counterexample anymore to distinguish the hypothesis from the behavior of the SUL.

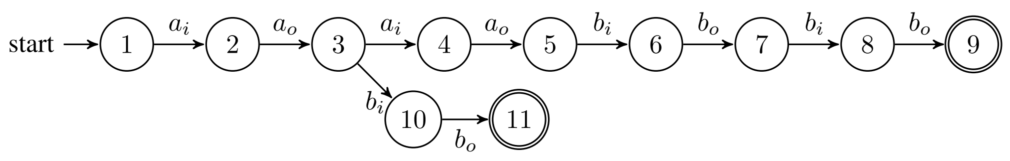
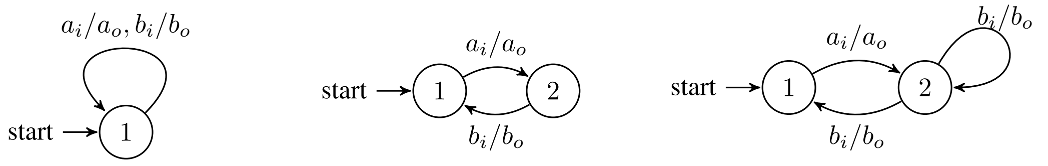


Fig. 2: PTA for a set of traces $t = \{a_i a_o a_i a_o b_i b_o b_i b_o, a_i a_o b_i b_o\}$



(a) Overapproximated model (b) Underapproximated model (c) A model containing both overapproximation and underapproximation

Fig. 3: Model (a) overapproximates the SUL from Fig. 1, (b) underapproximates it and (c) both overapproximates it and underapproximates.

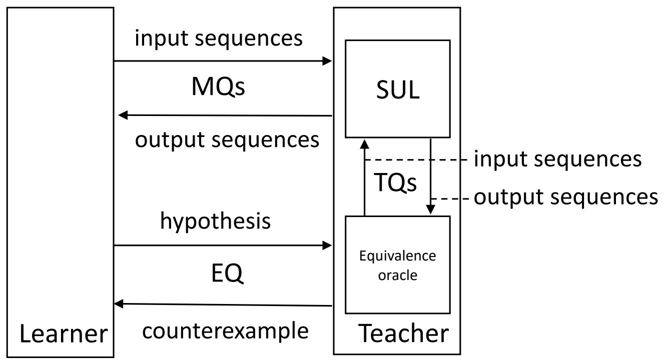


Fig. 4: Active learning framework

The learning algorithm guarantees the completeness of the learned model under the assumption that the equivalence oracle always returns a counterexample, given a counterexample exists. Peled et al. [24] proposed to use conformance testing to approximate the equivalence oracle. The partial W-method (Wp-method) [10] is a conformance testing technique that, given an upper bound m on the number of states of the target model, constructs a set of TQs to find the difference between the hypothesis and the SUL. Hence, if such a bound m is known for the given SUL, the equivalence oracle based on the Wp-method guarantees to find a counterexample if one exists.

However, in practice finding m is non-trivial as it requires a comprehensive understanding of the SUL [30]. Furthermore, the number of TQs increases exponentially in $m - n$ where n is the number of states of the hypothesis [34]. Underestimation can cause incompleteness while overestimation can cause scalability issues. Limited by time, in practice one has to stop testing the hypotheses at some point, sacrificing completeness. The last hypothesis, proposed before stopping, can overapproximate the SUL, underapproximate it or both overapproximate and underapproximate it for different parts of the behavior.

D. Passive learning

Different from active learning, passive learning algorithms learn a model based on a provided set of traces. The majority of passive learning algorithms is based on state merging [17], [33]. Many algorithms, such as RPNI [22], expect both positive traces (i.e., traces accepted by the SUL) and negative traces (i.e., traces rejected by the SUL). Below we explain the concept of state merging with a well-known RPNI algorithm. Many algorithms were later developed on top of it.

RPNI starts from positive traces to build up a PTA. Next the algorithm iteratively merges pairs of states. The merge might cause non-determinism which is then removed by merging additional states. For example, given the PTA in Fig. 2, RPNI might decide to merge states 3 and 7, hypothesizing $a_i a_o (a_i a_o b_i b_o)^* b_i b_o$. Next, the validity of the merge is checked: merges that accept negative traces are disallowed to avoid overgeneralization. This process continues until no further merges are possible. Upon termination, the algorithm has learned a model that accepts all positive traces and rejects all negative traces. The language of the PTA, representing the exact behavior of the set of positive traces, is a subset of the language of the model resulting from passive learning.

The passive learning algorithms guarantee to learn a complete model given a complete set of traces. Notions of completeness for a trace set differ for different algorithms: e.g., RPNI requires that the positive trace set visits every state and transition in the behavior of the SUL, and the negative trace set distinguishes every pair of states in that same behavior.

In practice, execution logs consisting of traces are used as the inputs to passive learning algorithms [20]. The execution logs usually only cover limited use cases, and contain only positive traces [36]. This means that the negative traces are practically absent [20], although such traces are needed to avoid overgeneralization as proven by Gold [11]. In the absence of negative traces, heuristics are typically used to prevent over-generalization [17]. These heuristics can vary, but typically still lead to overapproximation of some parts of a system. Together with the incompleteness of the logs, the passive learning result presents the same drawback as the active learning result, that is, both overapproximation and underapproximation can exist in the learned models.

III. Pilot study

To understand the scalability challenges of active learning and the role of the testing phase, we conduct a pilot study at ASML, provider of lithography systems for the semiconductor industry. Our approach to be presented in Section IV is inspired by the findings of the pilot study.

Smeenk et al. [29] applied active learning to learn an industrial component and reported that they did not learn the complete behavior of the component, having used more than 263 million queries over a learning period of 19 hours. Their study evidently supports the claim that the low learning efficiency reduces the scalability of active learning. However, they did not study *(1) to what extent testing is the bottleneck in the active learning process* and *(2) which distinguishing behavior between hypothesis and SUL is time-consuming to find via testing*. Answering these two questions is a prerequisite to developing more scalable solutions. Hence, we design a pilot study to answer these questions.

A. Study Design

a) Study objects: To apply active learning as described above, we need to identify the upper bound on the number of states m. In general, this step requires profound knowledge of the SUL and precise estimation of the upper bound. Therefore, we opt for components that originally were developed using traditional engineering practices and later manually migrated to MDSE, while preserving the functionality. Since these components are MDSE-based, we can use the number of states from the behavior of the MDSE models (i.e., reference models) as m. Moreover, since the components were first developed in a traditional way, they can be expected to exhibit a level of control-flow complexity comparable to other components developed using a traditional software engineering approach. Based on these criteria, we select the logistics controller of ASML's TWINSCAN lithography machine. This controller is in charge of scheduling the logistical process within a wafer scanner, making sure that each wafer is processed according to a specified recipe. In 2012, it was manually redesigned using an MDSE technology called Analytical Software Design (ASD) [38]. Over the years, 28 developers have performed more than 1,500 commits to the master branch of the version control repository of the controller. The resulting software consists of 218 communicating ASD components. Each component is modelled as a Mealy machine. The number of states in the Mealy machines varies between 1 and 18,229. The generated code consists of more than 700 KLOC.

b) Active learning setup: We opt for the state-of-the-art active learning algorithm TTT [16] together with the Wp-method for testing. Active learning consists of two steps, learning or refining of the hypotheses using MQs and testing these hypotheses using EQs and TQs. For each component, we separately measure the learning time and the testing time. We run the learning process with a timeout of 1 hour. In case of a timeout, we consider the last hypothesis constructed by the learner as the learned model.

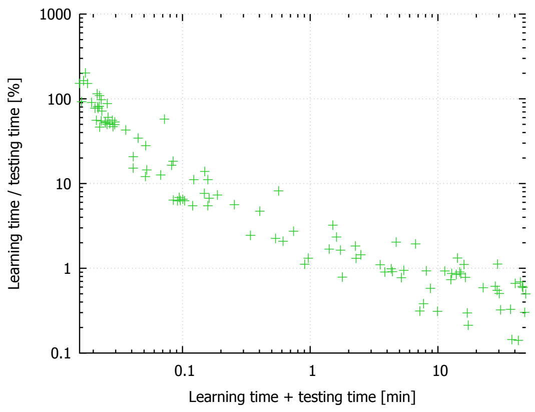


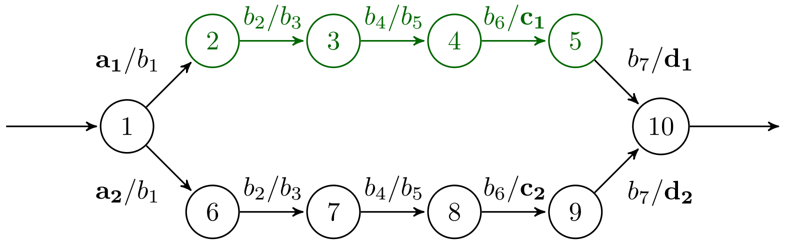
Fig. 5: Learning time and testing time in active learning

c) Model comparison: Following Aslam et al. [4] we consider a learned model to be complete if it holds a certain formal relation with its corresponding reference model. If a mismatch exists between the learned model and its reference model, we use the structure-based comparison method of Walkinshaw et al. [41] to identify the differences. We then analyze why the differences between the hypothesis and the SUL are hard to find via conformance testing, and what improvements might be beneficial to make this process efficient.

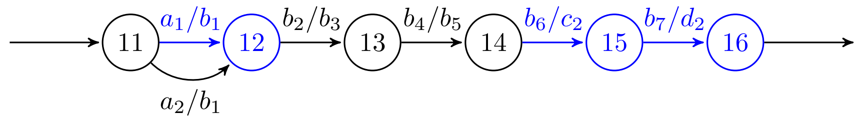
B. Results

Among the 218 components, 112 have been learned within 1 hour. For these components, when the total learning and testing time is small, learning is responsible for more than half of the total time (see Fig. 5). However, as the total time increases, the ratio drops: when the total time exceeds 1 minute, the learning time is less than 3% of the testing time.

Next, we take a closer look at one of the remaining 106 components. The reference model for this component is a Mealy machine with 14 states and 144 input actions. Fig. 6 presents only the states that show the structural differences between the last hypothesis constructed for this component and the component itself (SUL). The reference model in Fig. 6(a) shows that output actions c_1 and d_1 can only occur in the upper path, following the input action a_1 (as shown in bold). Similarly, output actions c_2 and d_2 can only follow the input action a_2 in the lower path (also shown in bold). However, the learner only successfully learned the path starting with input action a_2, as shown in Fig. 6(b). It also tried to explore input action a_1 from state 11, but failed to distinguish the paths because no different output action was immediately observed. As a result, output actions c_2 and d_2 can follow input action a_1 (as shown in blue). This means that the input sequence $a_1 b_2 b_4 b_6$ was not explored, as otherwise the counterexample that distinguishes state 2 from state 6 would have been found. Searching for such a counterexample requires the equivalence oracle to test all combinations of the input actions, to explore the behavior from state 2 to state 5 (in green) with sequences that consist of 3 input actions. Recall that the component



(a) Reference model



(b) Learned model

Fig. 6: Example showing the far output distinction behavior problem in the testing part of the active learning process

is much bigger than suggested by Fig. 6, which focuses only on the structural differences between the last hypothesis constructed and the component itself; the component has 14 states and 144 input actions. Exploring sequences of 3 input actions requires exploring 144^3 (i.e., 2,985,984) combinations in the worst case, while only one of them is relevant and can refine the model. Furthermore, the equivalence oracle not only considers behavior between states 2 and 5, but between other pairs of states as well. Even with a timeout of 8 hours, those two paths could not be distinguished.

We inspected other unfinished learning cases. The far output distinction behavior appears often in the reference models. Discussing this observation with the developers of the logistics controller we learned that such behavior is common in the controller due to non-stop parallel processing in TWINSCAN systems: for maximum accuracy and productivity a TWINSCAN measures one wafer while imaging another one. Hence, the controller has to, for instance, schedule the movement of two chucks which hold the measured and imaged wafers respectively. This system requirement is reflected by different control outputs given similar input sequences in the components of the controller. However, the far output distinction behavior is typically missing from the learned models.

C. Conclusions

This pilot study confirms that testing is the bottleneck of active learning as previously discussed in literature [29], [34]. It also shows that most of the testing time is spent on finding counterexamples that distinguish the states differentiated by an output action on a (far) future state.

In the pilot study, we took the advantage of MDSE components for which the number of states is known, and hence the Wp-method can be suitably configured. However, in practice users usually know little about a legacy SUL that was developed using traditional engineering practices. The underestimation of the upper bound m removes the guarantees the Wp-method provides, i.e., the learned models might be incomplete, even though we run the learning till it ends. Given the difficulty of estimating the upper bound m and the large amount of required testing time, obtaining a model that is far from complete is not unusual.

Furthermore, Fig. 6 suggests that if one can find the counterexample $a_1b_2b_4b_6$ faster, then the model can be completely learned in a shorter amount of time. Since ASML developers recognize far output distinction behavior as part of their regular system behavior, we expect that artifacts produced during system execution, such as logs, capture this far output distinction behavior. Hence, next we design the *sequential equivalence oracle* extending the Wp-method conformance testing to generate counterexamples based on execution logs and passive learning results.

IV. Sequential Equivalence Oracle

We start by presenting the overall architecture of our sequential equivalence oracle which has been briefly sketched [3], and then focus on its individual components.

A. Architecture

The idea behind the sequential equivalence oracle is similar to the idea behind hierarchical memory in computer architectures, i.e., expensive operations are used only when necessary. In computer architectures, different levels of caches are serving as staging areas, to reduce the needs of visiting relatively slow main memory and disk, when the CPU searches for data.

We can compare the traditional active learning approach to a computer architecture without caches. In the traditional active learning approach, when the learner requests a counterexample using an EQ, the Wp-method searches for the counterexample by asking the generated TQs to the SUL. This is an expensive operation as shown in the pilot study. To reduce the frequency of this expensive operation, we insert "caches", i.e., cheaper oracles, into the active learning process, as shown in Fig. 7.

Acting as the first "cache", the Log-based oracle starts searching for a counterexample when the hypothesis arrives. It returns a counterexample if found, otherwise the hypothesis is forwarded to a PL-based oracle. Similarly, the PL-based oracle returns a counterexample if it finds one, otherwise the hypothesis is forwarded to the Wp-method oracle. The role of the Wp-method oracle is comparable to that of main memory and disk where the data can always be fetched if it exists, at the price of time. We do not modify the Wp-method oracle; it works in the same way as in traditional active learning.

B. Log-based oracle

The Log-based oracle is based on the observation that logs represent actual behavior of the system. Hence, counterexamples can be found by identifying traces present in the log that cannot be generated by the hypothesis model H. To implement the Log-based oracle we collect execution logs for the SUL and construct a PTA, M_{PTA}, from these logs. Then, we compute the difference automaton for $M_{PTA} \setminus H$. If the resulting automaton has at least one accepting trace, which shows the language is not empty, the Log-based oracle generates a trace and returns it as a counterexample. Otherwise H is forwarded to the PL-based oracle.

C. PL-based oracle

Most passive learning algorithms ensure the inclusion of the input logs in their learned models. Hence, the execution logs are accepted *both* by the result of a passive learning algorithm and by the hypothesis H, that is, they are accepted by the *intersection* of the DFA representing the result of passive learning and H. The behavior represented by this intersection is more likely to belong to the SUL than behavior represented solely by the result of passive learning (but not H) or solely by H (but not the result of passive learning).

The PL-based oracle is based on the *conjecture* that passive learning generalizes the logs, potentially overapproximating the behavior, and that the behavior learned solely by passive learning (but not H) might contain valid generalization that belongs to the SUL.

As opposed to the Log-based oracle, the PL-based oracle is based on a *conjecture*. This is why in addition to computing the difference automaton for the hypothesis DFA and the DFA representing the passive learning result, and generating a trace from that difference, we need to check whether the generated trace is a valid counterexample or not. The oracle posts the generated trace as a TQ to the SUL. If the trace is accepted, it is valid and should be included in the behavioral model, so the trace is sent to the learner as counterexample to refine the learning. If the trace is rejected, it is excluded from the passive learning result as invalid generalization. This process continues until a counterexample is found or all generated traces are examined. The hypothesis is then forwarded to the Wp-method oracle if no counterexample can be found anymore.

D. Implementation

The sequential equivalence oracle is developed on top of LearnLib [25], an open-source framework providing the implementation of several active learning and testing algorithms.

As our target model is a Mealy machine, we use the learning algorithms provided by LearnLib for learning Mealy machines. We choose Mealy machines to represent behavioral models because Mealy machines are a good representation for reactive systems as they fit seamlessly with function calls

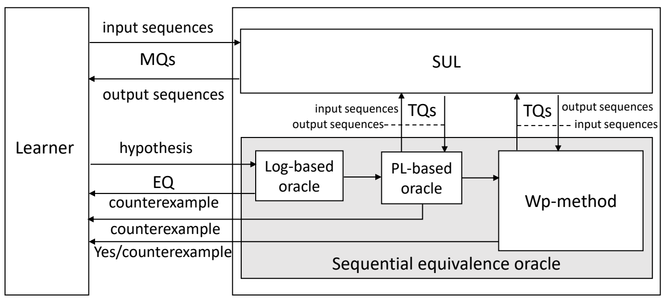


Fig. 7: Active learning with sequential equivalence oracle

and return values. When implementing the Log-based and PL-based oracles, we first convert hypotheses represented as Mealy machines to DFAs, and only then compute difference automata. The worst case complexity of the conversion and subsequent DFA operations is $O(n^2)$, where n is the number of states of the Mealy machine. Since LearnLib does not include means of computing the difference between two automata, we compute the intersection of one of the automata with the complement of the other. The resulting automaton is then minimized using a standard Hopcroft minimization [14].

In both the Log-based and the PL-based oracles, traces are generated from the difference automata by applying a breadth-first search until an accepting state is reached.

V. Evaluation of proposed Approach

In this section, we present the case study for the evaluation of our approach. We report our study according to the guideline proposed by Runeson et al. [27].

A. Research questions

The goal of this case study is to evaluate whether, and to what extent, our approach can improve the efficiency of active learning in an industrial setting. We refine our goal further to the following research questions:

RQ1: *To what extent does our approach reduce the time for learning a SUL?* Although our approach was inspired by particular observations about far output distinction behavior, we expect our approach can reduce the time for learning any SUL, given the behavior of the SUL is deterministic and not influenced by data parameters.

RQ2: *How do the Log-based oracle and the PL-based oracle individually contribute to reducing the testing time together with the Wp-method?* We conjecture that both the Log-based oracle and the PL-based oracle contribute to the improvement. However, it is possible that one outperforms another, or one of them does not contribute significantly. Answering this question can help us to improve the architecture of the equivalence oracle, as well as assist in making the trade-off between efficiency and computational complexity for the approach.

RQ3: *Does combining different oracle components help?* We combine Log-based and PL-based oracles with the Wp-method oracle as we conjecture that these two oracles contribute different behavior to the learning. However, it is possible that using only one of them with the Wp-method can already achieve the same or relatively comparable performance.

TABLE I: Features of 18 MDSE-based industrial components

	#states	#inputs		#states	#inputs		#states	#inputs
A	14	144	G	47	103	M	80	98
B	9	18	H	27	152	N	11	52
C	6	64	I	37	115	O	9	123
D	9	62	J	30	102	P	14	13
E	14	99	K	17	102	Q	14	159
F	2	3	L	17	102	R	37	102

B. Case study selection

We selected cases from the 218 ASD components studied in our pilot study. As they are MDSE-based components, we know the size of the behavior of the components, and can correctly configure the Wp-method oracle. Furthermore, we can evaluate the industrial applicability of our approach.

We applied the following criteria to select the components:

1. Logs should be available. Our approach needs logs for the Log-based and PL-based oracles. Unavailability of logs makes the approach inapplicable. At ASML, software execution is logged during both normal machine production and software testing, if logging is enabled for the component.

2. The behavior of the selected components must be deterministic and not influenced by data parameters.

Limited by the availability of logs, we obtain 18 components (from 218) as study objects. We name the components A to R; A is the component discussed in the pilot study. Table I shows the number of states and input actions of these components.

C. Experiment setup

1) Equivalence oracle setup: We conduct experiments with different equivalence oracle settings. We have four equivalence oracle settings in total. The equivalence oracle setting with the Wp-method alone is the control group. For the experiment groups, we have three additional equivalence oracle settings: the sequential equivalence oracles consisting of 1) Log-based, PL-based and Wp-method oracles (**EO1**), 2) Log-based and Wp-method oracles (**EO2**), 3) PL-based and Wp-method oracles (**EO3**). For each group we measure the testing time and the total active learning time. We configure a timeout of 1 hour for all experiments.

2) Logs and passive learning results: Log-based and PL-based oracles require logs and passive learning results as inputs, respectively. The used logs are collected from the execution of unit and integration tests, containing the interactions (i.e., input and output actions) between components and their system environment. Some post-processing, such as parsing and renaming, is conducted for fitting the passive learning tools. The passive learning results are obtained using the Alergia algorithm (with a configured bound of 10) [23] provided by FlexFringe [39].

3) Hardware setup: We executed all our experiments on a HP Z420 workstation, a desktop PC with an Intel Xeon E5-1620 v2, a quad core CPU consisting of cores running at 3.70 Ghz with hyperthreading, 32 gigabytes of memory, and running the Microsoft Windows 7 SP1 x64 operating system.

D. Statistical analysis

a) RQ1: To answer RQ1 given an oracle o (EO1, EO2, EO3) we formulate the following hypotheses:

$H^o_{0_1}$*: There is no statistically significant difference between the total learning time with the Wp-method alone and with the equivalence oracle o.*

$H^o_{a_1}$*: The total learning time with the Wp-method alone is greater than with equivalence oracle o.*

We formulate the alternative hypothesis as a directional alternative since testing the correctness of the hypothesis constructed by the learner is known to be the most expensive step in the active learning process. We expect our approach to reduce the testing time of the active learning process.

b) RQ2: We formulate the following hypotheses:

$H^o_{0_2}$*: There is no statistically significant difference between the total learning time with EO2 and with EO3.*

$H^o_{a_2}$*: The total learning time with EO3 is less than with EO2.*

The rationale behind this alternative hypothesis is that it is possible that EO3 outperforms EO2 as the passive learning results include the behavior shown in the log and potentially some other valid generalized behavior, which might further reduce the required testing time.

c) RQ3: Given an oracle o (EO2, EO3), the following hypotheses are formulated:

$H^o_{0_3}$*: There is no statistically significant difference between the total learning time with EO1 and with o.*

$H^o_{a_3}$*: The total learning time with EO1 is less than with o.*

This alternative hypothesis is a directional alternative since we expect that EO1, which is the combination of EO2 and EO3, results in shorter total learning times than when each one of the oracles is used separately.

d) Analysis technique: To test the hypotheses we perform pairwise tests (RQ1: Wp-method vs. EO1, Wp-method vs. EO2, Wp-method vs. EO3; RQ2: EO3 vs. EO2; RQ3: EO1 vs. EO2, EO1 vs. EO3). Next we adjust the p-values [5] to control the false discovery rate. Since answering RQ1, RQ2 and RQ3 involves comparing the same values, we adjust the six p-values together. Finally, if the difference is observed to be statistically significant, we report the effect sizes.

E. Results

Table II presents the results of learning components A to R with different equivalence oracle settings. Using the Wp-method oracle alone, 12 out of 18 components were fully learned within 1 hour. The learning for components A, G, K, L, M and R remained unfinished. In contrast, by applying EO1, EO2 or EO3, all components are fully learned within 1 hour. In particular, we completely learned component A, which exhibits far output distinction behavior, within 13 mins. This seems to be a promising result. Next, we further analyze the data to answer our research questions.

1) RQ1: To what extent does our approach reduce the time for learning a SUL? Fig. 8 shows the total learning times for different oracles. While it clearly suggests that *overall* the Wp-method oracle is by far the slowest, this does not

TABLE II: Experiment results (testing times and total learning times) for 18 industrial components using Wp-method oracle, Sequential equivalence oracle, Log-based oracle and PL-based oracle.

Component	Wp-method		Seq. equiv. oracle (EO1)		Log-based oracle (EO2)		PL-based oracle (EO3)	
	Testing [s]	Total [s]	Testing [s]	Total [s]	Testing [s]	Total [s]	Testing [s]	Total [s]
A	timeout	timeout	778.990	788.202	816.013	826.804	765.306	772.956
B	2.786	3.295	1.804	3.125	2.098	3.537	1.530	4.780
C	8.676	9.328	1.230	1.820	1.120	1.704	0.814	1.407
D	5.813	6.671	4.615	5.326	4.286	4.965	4.570	5.268
E	245.650	249.421	97.083	100.448	90.290	93.340	897.460	931.330
F	0.040	0.106	0.134	0.186	0.122	0.173	0.097	0.146
G	timeout	timeout	971.513	1007.606	1007.947	1038.270	984.008	1006.132
H	1789.083	1808.736	26.357	52.127	23.578	43.365	27.089	50.341
I	2768.578	2783.944	719.338	735.648	718.815	733.797	718.674	734.401
J	221.367	229.185	6.779	16.552	6.319	15.823	7.009	16.728
K	timeout	timeout	3.450	7.951	3.089	7.504	4.095	8.336
L	timeout	timeout	4.231	9.781	3.772	8.957	3.834	8.593
M	timeout	timeout	54.812	92.438	45.388	83.794	44.147	79.708
N	190.460	192.575	6.565	9.685	5.782	8.456	3.554	6.414
O	319.283	322.333	131.301	134.582	130.041	132.940	111.614	115.060
P	27.433	29.687	8.098	10.603	9.407	11.167	61.263	90.950
Q	2030.369	2039.024	1800.076	1811.001	1774.798	1787.820	1784.561	1792.723
R	timeout	timeout	16.528	35.635	14.817	35.234	18.456	34.503

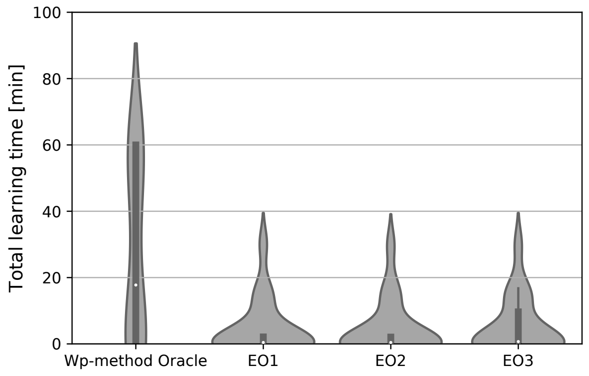


Fig. 8: Violin plots of the total learning times with different oracles

necessarily mean that *for each* individual component, the Wp-method oracle is slower than the other oracles. To answer this question we performed pairwise tests. Since the distribution of the learning times is skewed (cf. Fig. 8) we opt for the pairwise Wilcoxon rank sum tests, and for Cliff's delta as the effect size measure. We interpret the Cliff's delta according to the guidelines of Cohen [9].

We observe that for all pairs of oracles $H^o_{0_1}$ can be rejected in favour of $H^o_{a_1}$ ($p \simeq 7.6{*}10^{-6}, 1.9{*}10^{-5}, 1.6{*}10^{-3}$ for EO1, EO2, EO3, respectively). Furthermore, the effect of replacing the Wp-oracle with EO1 or EO3 is medium ($\delta \simeq 0.47, 0.46$, respectively) and with EO2 it is large ($\delta \simeq 0.48$).

2) RQ2: How do the Log-based oracle and the PL-based oracle individually contribute to the time reduction? We find that $H^o_{0_2}$ cannot be rejected ($p \simeq 0.62$).

3) RQ3: Does combining different oracle components help? The p-values are 0.96 and 0.93 for EO2 and EO3 respectively. Therefore, $H^o_{0_3}$ cannot be rejected.

F. Discussion

Our results show that the sequential equivalence oracle and its simplified versions, the Log-based and PL-based oracles, significantly improve the performance of active learning, while we could not observe a significant difference in performance of the Log-based and PL-based oracles. The conclusion one would like to derive is that integrating log data (either in the form of a log, or in the form of a model inferred from the log using passive learning techniques) in active learning is beneficial. However, it is not yet clear whether enhancing active learning with passive learning is always beneficial or not.

Next we perform a closer inspection of the performance of the different oracles on individual components. To this end we show a bar chart (Fig. 9) of the ratios of the total learning time with other oracles with respect to the total learning time with the Wp-method alone. For the cases where learning was not finished within 1 hour with the Wp-method, we use 1 hour as the total learning time. As expected EO1, EO2 and EO3 perform better than the Wp-method oracle alone, for most of the cases. Particularly, the total learning time for relatively large components H, K and L were significantly reduced. This suggests that our approach can potentially address the challenges of learning large systems. However, some exceptions exist. For example, for component F, the Wp-method oracle alone results in the shortest total learning time. This is because component F has only two states and three input actions (as shown in Table I). In such cases, the computation required in the Log-based and PL-based oracles costs more time than sending very few TQs with the Wp-method oracle. Moreover, for components B, E and P, EO3 costs more time than the Wp-method oracle. This likely indicates the presence of a significant amount of invalid generalization in the passive learning result; therefore, extra time was spent on validating the generated traces even though no counterexample was eventually found. Based on the observations, we can expect

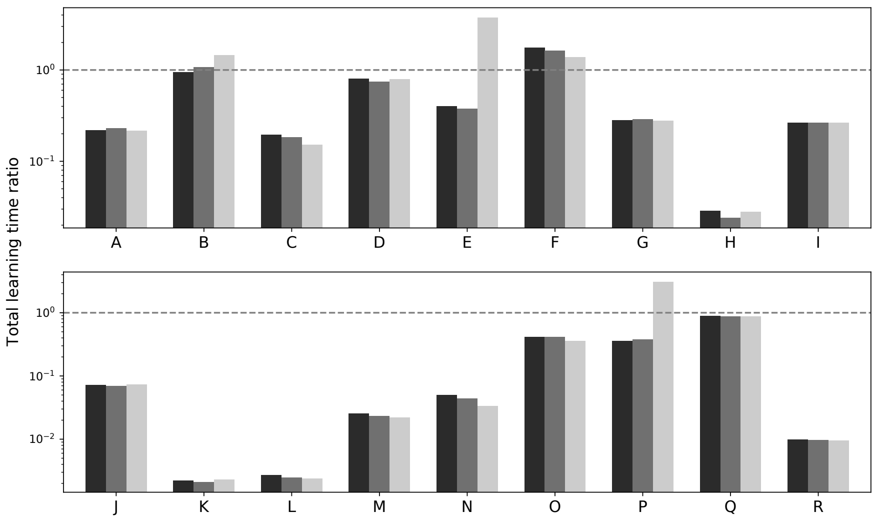


Fig. 9: Ratios of the total learning time with 1) EO1 (black), 2) EO2 (dark grey) and 3) EO3 (light grey) with respect to the total learning time with the Wp-method alone (shown with dashed lines)

that the performance of the PL-based oracle is influenced by the used algorithms and heuristics that generalize the logs in different ways. The quality of the logs also influences the performance of both the Log-based and PL-based oracles.

G. Threats to validity

a) Limitation: The main limitation of our evaluation is that we applied our approach to 18 components for which logs are available. The unavailability of logs for a larger set of components hindered us to evaluate our approach with a better distribution in the size of components and a wider diversity of component behavior. However, as the preliminary evaluation, the promising results motivate us to evaluate our approach on a larger set of software components in the future.

b) Construct validity: This validity examines whether what we measured can quantify the efficiency of active learning. We measured the total learning time as it intuitively measures the efficiency. However, the efficiency can possibly be quantified by such metrics as the number of MQs and TQs, since time might not be the only costly resource.

c) Internal validity: In order to control the variables of our experiments, we only change the equivalence oracles in each experiment, and keep the remaining settings the same. In this study we do not consider the types of execution logs (e.g., test and production logs), the features of traces (e.g., long or short traces) in logs and the heuristics of passive learning algorithms as variables, although they can greatly influence the performance of the Log-based and PL-based oracles. Moreover, we only run the experiments once, although the time spent in different runs might slightly differ.

d) External validity: This validity questions whether our conclusions are valid in a more general context. We find two threats to this validity. First, as stated, the limited number of case study objects is the main limitation of our work. We expect that the Log-based oracle can help for other systems as well because it finds counterexamples without costing any test query. Second, our case study objects are a group of components used to perform control logic of systems. A further study is required to evaluate our approach on different types of system from different companies.

VI. Related work

The idea of combining different techniques to solve theoretical and practical model learning problems is not new. Some hybrid learning techniques aim at enabling learning on a larger set of systems. Walkinshaw et al. [42] introduced a way to combine data mining techniques with a passive learning algorithm to learn the behavior that is influenced by data parameters. For the same purpose, Howar et al. [15] opened the black-box of the SUL by applying symbolic and static analysis techniques to iteratively refine the active learning result. Different from these approaches, our work combines techniques to improve the efficiency while guaranteeing a certain minimum behavioral coverage for deterministic and non-parameterized systems.

Smetsers et al. [30] combined conformance testing with mutation-based fuzzing methods, to enhance the equivalence oracle in active learning. This work uses a fuzzer to mutate the program tests, monitors the code coverage of the generated tests, and uses the mutated tests as a source of counterexamples. The authors experimentally showed that the hybrid approach can discover states that were not learned by using conformance testing alone. However, making use of the full potential of this approach requires the source code to be available; a 2.5 times efficiency degradation was observed when the source code was not available. Differently, our approach still treats the SUL as a black-box and therefore can be applied independently from the availability of the source code and programming language in which it was written.

To improve the quality of the models resulting from active learning, Smetsers et al. [31] used a metric to measure the distance between hypotheses and the behavior of the SUL. The proposed approach promises that the measured distance does not increase, i.e., the behavioral coverage does not decrease over time. This means that when users stop the learning, the current hypothesis is the best model the active learner has ever constructed, in terms of behavioral coverage. Our approach makes a different promise about the behavioral coverage, i.e., the learned models at least cover the execution logs.

Previous work has also shown that combining different techniques can reduce the need for testing in active learning. Howar et al. [15] adopt partial order reduction to reduce the number of required sequences in testing. This work relies on static analysis to determine mutually independent functions (input actions), and only a single order is constructed for these functions. Instead of using white-box techniques such as partial order reduction, our approach reduces the need for testing by searching counterexamples from the execution logs and the generalization of passive learning results. As stated, finding counterexamples faster is the key to reducing the number of tests. Smeenk et al. [29] used manually crafted counterexamples to reduce the testing time. However, constructing counterexamples manually requires domain knowledge which is not necessarily available. We use logs and passive learning results instead, and do not rely on domain knowledge.

Several approaches use logs to refine learning. Smetsers et al. [35] claimed their approach ensures that learned models cover the behavior of the logs, yet it is not clear *how* logs were integrated into the learning framework. Their work also suffers from limited evaluation and the use of artificial logs. Differently, we explicitly integrated the logs as an equivalence oracle and evaluated our approach on a larger scale in an industrial setting. Bertolino et al. [6] proposed to build up a framework for updating the hypothesis while a networked systems are evolving. This framework integrates a continuously running monitor that collects system traces at runtime, examines the mismatch between the hypothesis and the target system, and then refines the learning. However, this continuously running mechanism can be very expensive in terms of time, memory resources and infrastructure cost, thus making it less applicable. Our approach integrates pre-collected logs into the learning process. In addition, our approach enables combining logs from different sources (e.g., test execution and production), which can potentially enrich the behavioral coverage, as different logs might represent the execution of completely different use cases of the systems.

VII. Conclusion and future work

In this paper, we started by evaluating the performance of a state-of-the-art active learning setup on a collection of 218 MDSE-based components provided by ASML. We observed that the active learning converged for 112 components in one hour or less. For these components we have observed that as the total learning time increases, active learning becomes dominated by the testing phase. Active learning did not converge for 106 models. By inspecting one of these components, we observed that the lack of convergence can be attributed to the presence of far output distinction behaviour in the SUL. As far output distinction behaviour is part of the regular system behavior we expect to observe it in the execution logs.

To improve the efficiency of active learning we have proposed the sequential equivalence oracle integrating information from the execution logs and passive learning results. The sequential equivalence oracle has been evaluated on 18 industrial components. The results show that our approach can significantly reduce the total active learning time. Evaluation of the individual oracles suggests that using the Log-based oracle with the Wp-method might be sufficient to achieve good efficiency. However, considering other variables, such as the completeness of the logs and the level of generalization introduced by different passive learning algorithms, we suggest to conduct a more comprehensive case study that takes all these factors into account.

An additional advantage of the sequential equivalence oracle is the ease with which additional components based on behavioral evidence can be integrated, further reducing the need for testing. For example, as sub-oracles one can integrate manually crafted models (cf. Smeenk et al. [29]) or use multiple models learned by different passive learning algorithms. Furthermore, similarly to what we did to answer RQ2, users can analyze which sub-oracles contribute more to refining active learning, and adapt the sequential equivalence oracle to their context.

We consider several directions as future work. First, we plan to perform a more comprehensive study with the sequential equivalence oracle on a richer set of components (i.e., more components from different types of systems). This will allow us to investigate more variables and provide users a guideline to adapt our techniques in their own context. For example, it would be valuable to study what features of traces can complement the Wp-method better, which passive learning algorithms work best for our PL-based oracle, and in which scenario one oracle set-up performs better than others. Second, while in this study we apply the techniques to MDSE-based software, we plan to replicate our work on legacy software. Moreover, the current approach is limited to the class of systems where values of data parameters do not influence the behavior. Learning data-dependent behavior is still a challenge for model inference techniques in terms of scalability [28]. It might require us to open the black-box of the SUL (cf. Howar et al. [15]) and integrate static analysis techniques. Finally, one can integrate the distance-metric approach of Smetsers et al. [31], [35] with the use of logs advocated in the current work by employing recently proposed log-log and log-model comparison techniques of Gupta et al. [13] or Amar et al. [1].

Acknowledgement

This research was partially supported by The Dutch Ministry of Economic Affairs, ESI (part of TNO) and ASML Netherlands B.V., carried out as part of the TKI-project 'Transposition'; and partially supported by Eindhoven University of Technology and ASML Netherlands B.V., carried out as part of the IMPULS II project.

References

[1] H. Amar, L. Bao, N. Busany, D. Lo, and S. Maoz. Using finite-state models for log differencing. In G. T. Leavens, A. Garcia, and C. S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 49–59. ACM, 2018.

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

[3] K. Aslam, Y. Luo, R.R.H. Schiffelers, and M. van den Brand. Refining active learning to increase behavioral coverage. In *ACM WomENcourage*, 2018.

[4] K. Aslam, Y. Luo, R. R. H. Schiffelets, and M. G. J. van den Brand. Interface protocol inference to aid understanding legacy software components. In *Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems*, pages 6–11. CEUR-WS, 2018.

[5] Y. Benjamini and Y. Hochberg. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300, 1995.

[6] A. Bertolino, A. Calabrò, M. Merten, and B.Steffen. Never-stop learning: Continuous validation of learned models for evolving systems through monitoring. *ERCIM News*, 2012(88), 2012.

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

[8] M. Bugalho and A. L. Oliveira. Inference of regular languages using state merging algorithms with search. *Pattern Recognition*, 38(9):1457–1467, 2005.

[9] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988.

[10] S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on software engineering*, 17(6):591–603, 1991.

[11] E. M. Gold. Language identification in the limit. *Information and control*, 10(5):447–474, 1967.

[12] C. A. González and J. Cabot. Formal verification of static software models in MDE: A systematic review. *Information and Software Technology*, 56(8):821–838, 2014.

[13] M. Gupta, A. Mandal, G. Dasgupta, and A. Serebrenik. Runtime Monitoring in Continuous Deployment by Differencing Execution Behavior Model. In C. Pahl, M. Vukovic, J. Yin, and Q. Yu, editors, *Service-Oriented Computing - 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings*, volume 11236 of *Lecture Notes in Computer Science*, pages 812–827. Springer, 2018.

[14] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman, 2006.

[15] F. Howar, D. Giannakopoulou, and Z. Rakamarić. Hybrid learning: interface generation through static, dynamic, and symbolic analysis. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 268–279. ACM, 2013.

[16] M. Isberner, F. Howar, and B. Steffen. The TTT algorithm: a redundancy-free approach to active automata learning. In *International Conference on Runtime Verification*, pages 307–322. Springer, 2014.

[17] B. Lambeau, C. Damas, and P. Dupont. State-merging DFA induction algorithms with mandatory merge constraints. In *International Colloquium on Grammatical Inference*, pages 139–153. Springer, 2008.

[18] 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.

[19] D. Lo and S. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 51–60. IEEE, 2006.

[20] D. Lo, L.Mariani, and M. Santoro. Learning extended FSA from software: An empirical assessment. *Journal of Systems and Software*, 85(9):2063–2076, 2012.

[21] T. Margaria, O. Niese, H. Raffelt, and B. Steffen. Efficient test-based model generation for legacy reactive systems. In *High–Level Design Validation and Test Workshop*, pages 95–100. IEEE, 2004.

[22] J. Oncina and P. Garcia. Inferring regular languages in polynomial updated time. In *Pattern recognition and image analysis: selected papers from the IVth Spanish Symposium*, pages 49–61. World Scientific, 1992.

[23] Jose Oncina and Pedro Garcia. Identifying Regular Languages In Polynomial Time. In *ADVANCES IN STRUCTURAL AND SYNTACTIC PATTERN RECOGNITION, VOLUME 5 OF SERIES IN MACHINE PERCEPTION AND ARTIFICIAL INTELLIGENCE*, pages 99–108. World Scientific, 1992.

[24] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2002.

[25] 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, 2009.

[26] R. Ronald and S. Robert. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[27] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131, 2009.

[28] L. Sanchez. Learning software behavior through active automata learning with data. Master's thesis, Eindhoven University of Technology, 2018.

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

[30] R. Smetsers, J. Moerman, M. Janssen, and S. Verwer. Complementing Model Learning with Mutation-Based Fuzzing. *arXiv preprint arXiv:1611.02429*, 2018.

[31] R. Smetsers, M. Volpato, F. Vaandrager, and S. Verwer. Bigger is not always better: on the quality of hypotheses in active automata learning. In *International Conference on Grammatical Inference*, pages 167–181, 2014.

[32] A. Stevenson and J. R. Cordy. A survey of grammatical inference in software engineering. *Science of Computer Programming*, 96:444–459, 2014.

[33] B. Trakhtenbrot and Y. Barzdin. Finite automata: Behavior and synthesis. *Journal of Symbolic Logic*, 42(1):111–112, 1977.

[34] F. Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, 2017.

[35] P. van den Bos, R. Smetsers, and F. Vaandrager. Enhancing Automata Learning by Log-Based Metrics. In *International Conference on Integrated Formal Methods*, pages 295–310. Springer, 2016.

[36] W.M.P. van der Aalst. *Process mining: data science in action*. Springer, 2016.

[37] J. M. E. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, and A. Serebrenik. Process Discovery using Integer Linear Programming. *Fundam. Inform.*, 94(3-4):387–412, 2009.

[38] Verum, 2014. http://www.verum.com.

[39] S. Verwer and C. A. Hammerschmidt. flexfringe: A Passive Automaton Learning Package. In *IEEE International Conference on Software Maintenance and Evolution*, pages 638–642, Sept 2017.

[40] N. Walkinshaw and K. Bogdanov. Inferring finite-state models with temporal constraints. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 248–257. IEEE Computer Society, 2008.

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

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