Software and Systems Modeling (2020) 19:1519–1540
https://doi.org/10.1007/s10270-020-00809-2

THEME SECTION PAPER

Interface protocol inference to aid understanding legacy software components

Kousar Aslam[1] · Loek Cleophas[1] · Ramon Schiffelers[1,2] · Mark van den Brand[1]

Received: 22 June 2019 / Revised: 28 May 2020 / Accepted: 31 May 2020 / Published online: 28 June 2020

Abstract
High-tech companies are struggling today with the maintenance of legacy software. Legacy software is vital to many organizations as it contains the important business logic. To facilitate maintenance of legacy software, a comprehensive understanding of the software's behavior is essential. In terms of component-based software engineering, it is necessary to completely understand the behavior of components in relation to their interfaces, i.e., their interface protocols, and to preserve this behavior during the maintenance activities of the components. For this purpose, we present an approach to infer the interface protocols of software components from the behavioral models of those components, learned by a blackbox technique called active (automata) learning. To validate the learned results, we applied our approach to the software components developed with model-based engineering so that equivalence can be checked between the learned models and the reference models, ensuring the behavioral relations are preserved. Experimenting with components having reference models and performing equivalence checking builds confidence that applying active learning technique to reverse engineer legacy software components, for which no reference models are available, will also yield correct results. To apply our approach in practice, we present an automated framework for conducting active learning on a large set of components and deriving their interface protocols. Using the framework, we validated our methodology by applying active learning on 202 industrial software components, out of which, interface protocols could be successfully derived for 156 components within our given time bound of 1 h for each component.

Keywords Active automata learning · Interface protocols · Learning framework · Equivalence oracles

1 Introduction

Large-scale software systems are inherently complex, with complexity caused by a large number of constituent components and the interactions between them [54]. The software also changes over time due to maintenance as a result of evolving requirements, emerging technology trends and hardware changes [38]. To deal with this ever-increasing complexity of high-tech software systems, different software development methodologies emerged with the goal of raising the abstraction level of software development. In the late 1960s, component-based software engineering (CBSE) [9] started becoming popular, which presented the idea of componentization and the consequent focus on interface specification. The components communicate with each other through the interfaces [6]. The communication follows the software specification which states the order and timings of the messages [43].

As a step forward in increasing abstraction of software development, in the 1980s, model-driven engineering (MDE) [50] started coming into play. MDE represents software structure and behavior in terms of models, which are better suited for formal verification and are closer to the problem domain. Hence, MDE-based software is expected to be easier to understand and maintain by the domain experts [20]. For building complex software, high-tech companies are now

Communicated by Federico Ciccozzi, Antonio Cicchetti and Andreas Wortmann.

✉ Kousar Aslam
k.aslam@tue.nl

Loek Cleophas
l.g.w.a.cleophas@tue.nl

Ramon Schiffelers
r.r.h.schiffelers@tue.nl; ramon.schiffelers@asml.com

Mark van den Brand
m.g.j.v.d.brand@tue.nl

[1] Eindhoven University of Technology, Eindhoven, The Netherlands

[2] ASML, Veldhoven, The Netherlands

adopting MDE techniques [25,41], yet a considerable amount of software exists that was developed by manual coding. For the components that are built with MDE, usually good interface protocols are available. In the general case, however, for traditionally developed software components only some rudimentary interface descriptions can be found.

To achieve the same benefits for traditionally developed software as those offered by MDE, such as formal verification, easier understanding and maintenance, interface protocols need to be constructed for this software. Manually creating models for large legacy software systems is very time-consuming and often infeasible. In this paper, we present the research that we have conducted to support an automated and cost-effective transition from existing software to models. We try to achieve this by developing techniques which use information available in source code, execution traces and/or other software documentation to infer interface protocols [5,67]. The interface protocols, once learned, can serve as the starting point for several engineering and maintenance activities, which is discussed in Sect. 2.

Several techniques exist in the literature about the reverse engineering of existing software components. These techniques can be broadly categorized as static or dynamic analysis techniques. Static software analysis methods examine the code without actually executing it. This provides an understanding of the code structure and components constituting the software [23]. Dynamic software analysis techniques examine the actual execution of the software, either by processing execution traces—passive learning [37] or by actively interacting with the software components—active automata learning or active learning [52].

In this work, we have explored the active learning technique. Active learning extracts the software behavior by alternating between two phases, learning and testing. Learning builds a hypothesis, and testing checks the equivalence between the hypothesis and the system under learning (SUL). Learning a behavioral model from an existing software system using active learning has been an area of quite some interest. In this context, the technique was first used in 2010 to learn a network protocol for analyzing *botnets* [12]. Later, it has been used for learning formal models of bank cards [2], reverse engineering a smartcard reader for bank cards [11], learning big control software component from Océ printers [53] and learning legacy code at Philips [51][3]. These studies show active learning as a promising approach to construct behavioral models from existing software, for different purposes.

The scope of previous active learning studies is, however, limited to a single (part of a) software component, or at most a few components. Furthermore, all of the existing case studies performed with active learning mostly focus on learning the current behavior of software components. To the best of our knowledge, there is no prior published work on inferring interface protocols for software components based on active learning. Also, in practice, active learning cannot learn the correct and complete model of the SUL. This is because the number of queries posted to the SUL is finite and limited for practical reasons (Sect. 4.2), making it typically impossible to guarantee that the active learning result is a true representation of the behavior of the legacy software components [59]. Therefore, the correctness and completeness of learned results cannot be ensured.

To address the above-mentioned challenges, we propose a two-step methodology to infer the interface protocols for software components. In the *first* step, we apply active learning to learn the behavior of software component. For reducing the uncertainty about the learned models, we apply active learning on MDE-based components. These components are based on models, which we use as reference models to formally check the equivalence with our active learning result. This is an excellent step toward building confidence in the technique and validating its correctness before applying it to legacy components with unknown behavior. Our chosen components only possess control flow behavior and do not have data flow behavior. This is because learning data flow behavior is still an open research question in the active learning field [59]. There have been efforts to make steps in this direction [1,10,24], but there is not enough evidence yet, both in terms of fundamental research and required tool support to learn data flow behavior of industrial-scale software components.

In the *second* step of our methodology, we infer interface protocols from the active learning results of a desired interface by abstracting away (i.e., hiding) the actions coming from other interfaces. We validate the interface protocols by verifying that the original behavioral relations, which existed between the original interface specification and the implementation, are preserved between the derived interface protocol and the implementation. This validation ensures that the inferred interface protocols are correct. Our methodology provides the basis for inferring interface protocols of legacy software components, for which indeed no reference models are available.

To validate our approach, we present an automated framework that can be used to perform active learning on a number of software components. Using our framework, we applied active learning on 202 MDE-based software components from our industrial partner ASML's lithography machines software. Testing is known to be the bottleneck of active learning process [59,67]. To achieve best results, and benefiting from our automated framework, we applied active learning using several testing methods. In this way, we are able to infer interface protocols for a large number of software components out of 202 software components under learning. We derived interface protocols for the 156 components

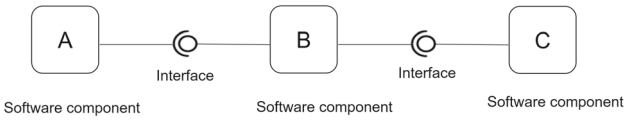


Fig. 1 A simple CBSE-based system with components *A*, *B* and *C*, interacting with each other over the interfaces

learned by active learning and verified that behavioral relations are preserved between learned and reference models.

This paper is an extension of our previous work [5], where we already presented the methodology and a small example as proof of concept. The added contributions in this paper include:

1. an automated framework to apply our methodology on a large number of software components.
2. a large-scale industrial evaluation as validation of our methodology, using several testing methods.

Outline of paper Section 2 discusses a few use cases of the derived interface protocols as motivation of our work, Sect. 3 summarizes the related work, Sect. 4 gives the background information necessary to understand the paper, Sect. 5 presents our methodology to infer interface protocols, and an example is presented in Sect. 6 to illustrate the working of our approach. Section 7 explains our framework for conducting active learning experiments, and Sect. 8 presents the evaluation of our approach. We finally conclude and present future work directions in Sect. 9.

2 Motivation

As already discussed, deriving interface protocols is part of understanding legacy software components. Models such as interface protocols can be used like models for components developed using MDE, e.g., they can be used for formal verification and are closer to the problem domain, facilitating understanding of the legacy components. The interface protocols coming from the complex software of high-tech systems can be utilized in a variety of ways, which we discuss below. We explain each use case using a small system example, shown in Fig. 1. The system consists of three software components *A*, *B* and *C*, interacting with each other over the interfaces depicted between them.

2.1 Observer and armor

2.1.1 Observer

Observer and monitor are two terms used interchangeably to refer to the activity of observing software execution [44,62]. By monitoring the externally observable behavior of a system, the observer determines if it is consistent with a given specification. Software can be observed for different purposes such as performance analysis, software fault detection, diagnosis and recovery [14]. This increases the confidence in the implemented system.

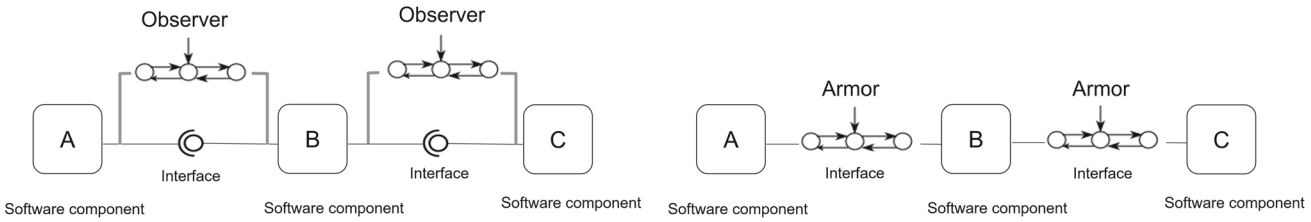
For CBSE-based software systems, the observer monitors the calls to the interface functions and responses made to such calls. In this way, it detects unexpected behavior [69]. The goal of the observer is to collect the actual interactions by observing the communication between the components. The ability of an observer to detect unusual behavior is determined by its specifications. For instance, an observer may be specified to check whether the number of processes for one software component is as specified, or how many events are produced by a software component during a certain timespan. Observers find applications in several domains, e.g., Steinbauer et al. [55] used observers for detecting failures in the control software of autonomous robots. Diaz et al. [15] propose a system in which distributed systems can be verified with respect to the specifications of a previously defined model.

The interface protocols contain the information of software components and their relationships. They can monitor the externally observable behavior of software components and compare it to the specifications and raise a flag in case of a discrepancy. In this way, a separate observer will not need to be implemented. This is shown in Fig. 2a.

2.1.2 Armor

The functionality of an armor is one step forward compared to that of the observer. It blocks any call to the interface that does not obey the specifications of the component, thus protecting the software from illegal behavior. With armoring, it is easy to distinguish failures caused by protocol violations from failures caused by (functional) errors. Armoring reduces the risk of process crashes or hangups. In the presence of armoring, the analysis of failures caused by protocol violations becomes easier. An armor component separates all the error handling code from the main functionality of the system, thus facilitating separation of concerns.

Armoring facilitates automatic program recovery by detecting illegal behavior. The idea gained a lot of popularity in recent decades [35]. The Adaptive Reconfigurable Mobile Objects of Reliability (Armor) middleware architecture [30] was developed and used in different case studies to detect and recover from failures at runtime. Kalbarczyk et al. [31] have developed a tool called Chameleon which uses the concept of armors to provide an infrastructure for runtime-adaptable fault-tolerant software. Stramaglia [56] applied armoring to reduce the probability of memory errors affecting the operating system and causing a reboot.



(a) Observer **(b)** Armor

Fig. 2 Interface protocols serving as observer and armor for software components

For MDE-based components, armors can be automatically constructed from formal specifications and then integrated at proper places in the program. Figure 2b shows the use of an interface protocol as armor.

2.2 Re-factoring and re-engineering

As discussed in Sect. 1, large-scale software systems typically undergo repeated changes over time due to evolving requirements, hardware changes and emerging technology trends. During this evolution phase, the software documentation may not be regularly updated and the developers initially working on the project may no longer be available, as discussed by Lehman [38]. These factors turn the software into so-called legacy software which becomes harder to understand and costly to maintain. Legacy software usually implements crucial domain logic which cannot be discarded or easily replaced. However, for different reasons, such as technology changes or performance issues, the legacy components may need to be re-factored or re-engineered. To support such activities, the implicit domain logic (behavioral models) of these components and interface protocols implemented between the components need to be extracted and learned. While re-factoring or re-engineering, the external behavior of the component needs to be preserved so that the whole system operates externally in the same manner as before the intervention.

Figure 3 shows the scenario where a component B is to be replaced by B'. After replacement, B' is expected to perform the same interactions with the surrounding components A and C. As these interactions are meant to be specified in interface protocols, these protocols can serve as the starting point by specifying requirements for re-factoring, re-engineering, code modernization or software rejuvenation. For legacy components, therefore, it is necessary to derive the interface protocols so that maintenance activities can be facilitated.

2.3 Supervisory control synthesis

Supervisory controllers coordinate the control of the individual machine components. The traditional practice of

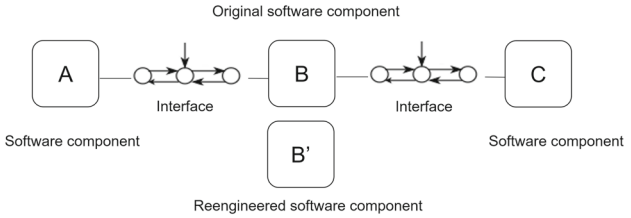


Fig. 3 Replacement of software components by a re-factored/re-engineered component

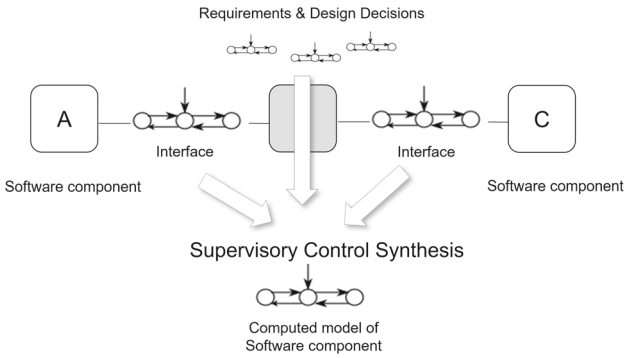


Fig. 4 Supervisory control synthesis by integrating software requirements and interface protocols

developing supervisory controllers is to code them manually, based on control requirements. Using supervisory control theory (SCT) [65], an uncontrolled system and the requirements are formally specified in terms of an automaton (a.k.a. the plant). Then, from these models, the supervisor is derived. SCT synthesizes the models of the supervisors such that the correctness of these models is predetermined.

Such supervisor synthesis has been studied by van Beek et al. [60] in the context of the Compositional Interchange Format (CIF),[1] a modeling language and accompanying toolset based on automata that is used to specify discrete-event, continuous-time and hybrid systems. Loose [39] synthesized supervisory controllers for multiple MDE-based components and compared these with manually created supervisory controllers, enabling verification of the synthesized ones. The use

[1] http://cif.se.wtb.tue.nl/index.html.

of (formal) models for controller design allows validation and verification of controllers long before they are implemented and integrated into the system, leading to fewer defects and reduced costs.

Figure 4 shows the idea for supervisory control synthesis where component models can be computed using SCT based on interface protocols and the requirements. The interface protocols describe the uncontrolled system. The *requirements and design decisions* are the requirements to be implemented by the control component. Using the interface protocols and these requirements, with SCT one can automatically generate the controller component.

3 Related work

A considerable amount of work for inferring the internal (control) behavior of existing software with static and dynamic analysis (passive and active) techniques can be found in the literature.

Static analysis Software reusability is facilitated by using frameworks and libraries through Application Programming Interfaces (APIs). Using advanced and complicated APIs is often challenging due to hidden assumptions and requirements. Research has been conducted lately to retrieve undocumented properties of APIs. An overview of static analysis techniques and tools for automated API property inference is given in [48].

An annotation assistant tool to infer pre-conditions and post-conditions of methods of Java programs is presented in [17]. Identifying the pre-conditions of functions to infer both data and control flow is discussed in [46]. Buse and Weimer present a tool that learns exception-causing conditions in Java programs [8]. Their approach infers the properties concerned with behavior in erroneous conditions, so they can describe only the pre-conditions that lead to exceptional behavior. For a given API method, the approach infers possible exception types that might be thrown, predicates over paths that might cause these exceptions to be thrown and human readable strings describing these paths. Tillmann et al. developed a tool called Axiom Meister that infers properties, using symbolic execution, automatically from the code [57]. The inferred specifications are human readable and can also be used as input to program verification systems or test generation tools for validation.

Passive learning Data mining techniques are combined with passive learning algorithms to learn data flow behavior of software components [64]. A passive learning algorithm named GK-tail is presented in [40]. The authors show that the behavioral models capturing the constraints on data values and the interactions between software components are learned with this algorithm. Model checking has been used to enhance a passive learning algorithm (QSM) to learn software behavior [63]. Passive learning has been applied for dynamically inferring functional specifications from method calls [66]. They developed a tool called Perracotta that uses heuristics to generalize inferred specifications into regular expressions. These regular expressions can be represented as a DFA.

Active learning The active learning technique has been used for understanding existing software and analyzing current implementations for detecting flaws. Aarts et al. [2] applied active learning to learn formal models for bank cards. Chalupar et al. [11] reverse engineered a smartcard reader for internet banking using active learning and managed to detect a security flaw in its behavior. Smeenk et al. [53] learned a big control software component, the Engine State Manager (ESM), from Océ printers. They used efficient techniques to find counterexamples faster and managed to learn the behavioral model of this complex software. In [51], the authors combined model learning and equivalence checking to gain the confidence in refactoring of legacy components. Equivalence is checked between active learning results from the legacy and refactored component with the mCRL2 toolset. If the models are equivalent, learning ends. Otherwise, models or implementations are refined based on the counterexamples generated by mCRL2. This again is based on the assumption that active learning can learn complete and correct behavior, which does not hold in practice. Recently, Duhaiby et al. [3] discussed the challenges faced and lessons learned by applying active learning to software components of Philips Healthcare systems.

All of the work stated above deals with learning existing software, and the learned results or inferred specifications are not formally validated. We propose that formally validating the learning techniques (active learning in our case) brings greater confidence in the methodology when applied to legacy components. Therefore, we check the equivalence of active learning results with the reference models and formally validate the interface protocols. Equivalence checking guarantees that during learning we preserve the formal relations and do not lose any behavioral information.

To support the software maintenance, we need to infer the interface protocols for communication between software components. To the best of our knowledge, there is no existing work to infer interface protocols using active learning. Our work is a step toward building confidence for applying active learning to derive interface protocols for legacy components as well.

4 Background

4.1 Definitions

First, we present a few definitions that will be used later in the paper.

We choose *Mealy machines* to represent the results of active learning and derived interface protocols. This is because Mealy machines provide the notion of inputs and outputs; therefore, they are good representations for reactive systems.

Definition 1 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 An action that can happen but cannot be directly observed is called an internal action, denoted by τ.

Definition 3 An *interface protocol* is a tuple $\mathcal{IP} = \langle S, \Sigma, \Omega, I, \rightarrow, \hat{s} \rangle$, where

- S is a set of states,
- Σ, Ω, I are three pairwise disjoint sets of input, output and internal actions,
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation, where $Act = \Sigma \cup \Omega \cup I$ and
- $\hat{s} \in S$ is the initial state.

Definition 4 A labeled transition system (*LTS*) is a tuple $\mathcal{L} = \langle S, Act, \rightarrow, \hat{s} \rangle$, where

- S is a set of states,
- Act is a set of actions,
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation and
- $\hat{s} \in S$ is the initial state.

Definition 5 Given a Mealy machine $\mathcal{M} = \langle S, \Sigma, \Omega, \rightarrow, \hat{s} \rangle$, we define the *underlying LTS* as $\mathcal{L}(\mathcal{M}) = \langle S_L, \Sigma \cup \Omega, \rightarrow_L, \hat{s} \rangle$, where S_L and $\rightarrow_L$ are the smallest sets satisfying:

- $S \subseteq S_L$
- for every $(s, i, o, t) \in \rightarrow$, there are transitions in $\rightarrow_L$ such that $s \xrightarrow{i} s_1 \xrightarrow{o} t$, where $s_1 \in S_L$ that does not have any other incoming or outgoing transitions.

Definition 6 Given a set of input traces, a prefix tree acceptor $\mathcal{PTA}$ is a tree-like automaton (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.

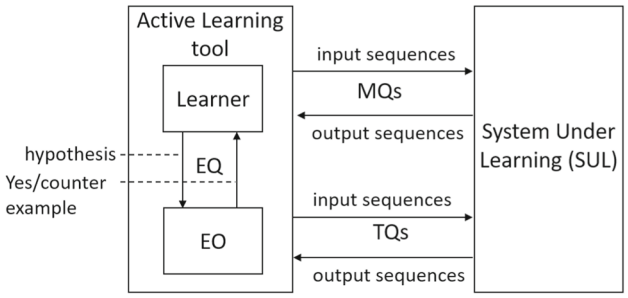


Fig. 5 Active learning framework

4.2 Active learning

Angluin [4] presented active learning in 1987 to learn regular languages. The technique is based on the minimal adequate teacher (MAT) framework. Figure 5 shows the MAT framework for active learning. The MAT framework assumes the availability of a teacher who is minimally able to provide correct answers to the queries posted by the learner. The learner is provided with a finite input alphabet for the SUL.

With this input alphabet, the learner starts learning the SUL by posting membership queries (MQs) to the SUL. Based on the responses to these queries, the learner formulates a hypothesis. To verify the correctness of this hypothesis, the learner posts an equivalence query (EQ) to the equivalence oracle (EO). If the hypothesis is a true representation of the behavior of the SUL, the learning is completed and thus stopped. Otherwise, the EO returns a counterexample based on which the learner further refines the learning. The counterexample shows the difference between the currently learned model and the SUL. The learning cycle continues until the hypothesis is accepted by the EO.

Several learning and equivalence checking algorithms have been proposed over the years. The learning algorithms include the TTT [26], the Rivest–Schapire [47], the Kearns–Vizarani [32], etc. All of these algorithms still follow the basic MAT framework proposed by Angluin. Previous case studies show that TTT scales up to learning larger components. TTT is highly efficient due to the use of redundancy free data structures. It asks fewer membership queries and uses less memory in the process. Therefore, in our work we also choose TTT as the learning algorithm.

To find these counterexamples, usually some conformance testing technique such as the W-method [13], the Wp-method [18] and hybrid-ADS [53] are used to test the hypothesis. In theory, using the conformance testing techniques as equivalence oracle can learn a complete model. However, this requires the number of states of the SUL to be known. Yet

for legacy systems, the (exact) behavior of SUL is unknown. Therefore, in practice an estimation of the number of states is used. As the number of required test queries increases exponentially with the increased difference in number of states between a hypothesis and the actual system behavior [59], the estimates are often kept lower for performance reasons. Due to the number of queries posted to the SUL being finite, and the use of reduced estimates, it is practically impossible to guarantee that the learned model is the true representation of the behavior of the legacy component.

In our work, we have experimented with several existing equivalence checking methods which are explained below.

4.2.1 Wp-method

The Wp-method [18] is a conformance testing technique which generates test cases to characterize the specification. The test trace generated by the Wp-Method is the cross product of three sets of sequences, namely *prefix*, *infix* and *suffix*. Here we informally describe these sets.

The *prefix* P is a prefix closed set which contains all the sequences required to reach every state of the hypothesis from the initial state. This set of sequences ensures we reach all the states of the hypothesis. However, the SUL can have hidden states that do not exist in the specification. The Wp-method introduces a parameter k which is the number of potential extra states in the SUL. The *infix* set is constructed using the value of $k : \{\epsilon \cup \Sigma \cup \cdots \cup \Sigma^{k+1}\}$ where Σ is a set of symbols used in the hypothesis. The Wp-method generates an n-complete test suite consisting of test cases which are concatenations of an element of the prefix set P, the infix set I and the suffix set S. This is given by the equation, $\pi = P \cdot I \cdot S$. The test suite ensures that the test cases exhaustively examine not only the expected states but also k extra states. The infix set ensures we cover the entire SUL in case it has more states than the hypothesis. The *suffix* set S comprises the sequences that identify every single state in the model. The suffix set checks whether the prefix and infix agree on the states that are reached.

The Wp-method reduces the size of the test suite as compared to its predecessor, the W-method [13], by reducing the size of the suffix set. The Wp-method is based on the observation that a subset of the suffix set can be selected, based on the final state in the Mealy machine at the end of the transitions due to the prefix and infix inputs. For example, after a combination of prefix and infix inputs, if the model is in state 1, the suffix set for state 1 is used. So, the entire suffix set is not needed every single time.

4.2.2 Hybrid-ADS

An adaptive distinguishing sequence (ADS), proposed by Lee et al. [36], can be used to reduce the suffix set to a single sequence. An ADS is actually not a sequence; it is a *distinguishing tree* that can be used for *state identification*. State identification refers to the problem of identifying the initial state of a Mealy machine. After the transitions from the prefix and infix sequences, the current state of the running machine becomes the initial state for an ADS. It is then possible to identify that state using an ADS. However, an ADS does not always exist for every Mealy machine.

For this reason, Smeenk et al. [53] proposed Hybrid-ADS (H-ADS), which is an augmentation to the basic ADS method using Harmonic State Identification (HSI) [42]. HSI can identify states in partially specified non-deterministic finite state machines (PNFSMs). A PNFSM is a finite state machine (FSM) with possibly multiple input transitions of the same type for each state, some (or even all) of which may be unknown. Given that FSMs are a subclass of PNFSMs, the same technique can also be applied to all the Mealy machines in our case. In the absence of an ADS, H-ADS supplements HSI with the intermediate results of the ADS tree.

4.2.3 Cache-based oracle

A cache-based oracle is based on the idea that queries that have been presented to the oracle before do not need to be asked again, for instance, the counterexamples that a learner has already processed provided by the tester or the testing queries that have already been asked in the previous testing round.

A cache-based oracle also facilitates sink state optimization. This means that if an illegal trigger is encountered, the whole input sequence after that trigger is not tried. Testing algorithms do not know which inputs are illegal and they will try all combinations up to a certain length after such input. These input queries can be trivially answered using interfacing protocol knowledge (see Sect. 7.2.2).

The active learning tool, LearnLib [45], supports both of the above functionalities.[2] Upon construction, the cache is provided with a delegated oracle. Queries that can be answered from the cache are answered directly, and others are forwarded to the delegated oracle. When the delegated oracle has finished processing these remaining queries, the results are incorporated into the cache.

4.2.4 Log-based oracle

Yang et al. [67] have proposed to use software execution logs to refine the hypothesis proposed by the learner, before posting it to the conformance testing algorithm. The log-based oracle is built on the observation that logs represent real behavior of the system. Hence, counterexamples can be

[2] http://learnlib.github.io/learnlib/maven-site/0.9.1/apidocs/de/learnlib/cache/mealy/MealyCacheOracle.html.

found by identifying traces present in the logs that cannot be generated by the hypothesis model. The log-based oracle constructs a $\mathcal{PTA}$ from the execution logs of the SUL. After constructing the $\mathcal{PTA}$, the log-based oracle computes the difference automaton for $\mathcal{PTA} \backslash Hypothesis$. If the resulting automaton has at least one accepting trace, which shows the language is not empty, the log-based oracle generates the trace and returns it as a counterexample. Otherwise, the hypothesis is forwarded to the conformance testing technique. A log-based oracle provides counterexamples without posting any test queries to the SUL.

In our work, we have combined logs and cache with both the Wp-method and H-ADS. The complete list of equivalence checking methods we experimented with is given below.

1. Wp
2. Cache + Wp
3. Log + Wp
4. Cache + Log + Wp
5. H-ADS
6. Cache + H-ADS
7. Log + H-ADS
8. Cache + Log + H-ADS

4.3 LearnLib and AutomataLib

We use the LearnLib [45] tool to perform active learning. LearnLib is a free and open-source (Apache License 2.0) Java framework for automata learning. LearnLib implements the latest learning techniques and algorithms and is actively being developed and maintained. Automatalib [27] is the standalone finite state machine library[3] that is developed on top of LearnLib. It provides a rich toolbox of data structures and algorithms for finite state machines, facilitating graph theory, automata theory and model checking.

4.4 Analytical software design (ASD)

ASD:Suite, a tool by the company Verum[4], is based on Verum's patented Analytical Software Design (ASD) [7] technology. ASD and ASD:Suite are used interchangeably in this paper. ASD is a component-based technology which enables engineers to specify, design, validate and formally verify software components for complex software systems. The basic unit of ASD is a component. ASD components provide and use services from other components. An ASD component is composed of two types of models; namely interface and design models. The external behavior of a component is specified in the interface model. The design model

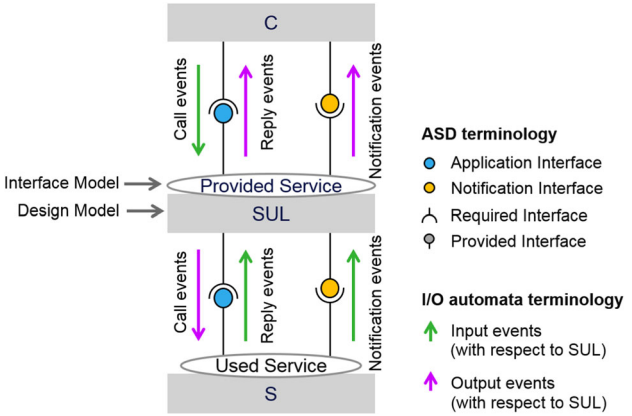


Fig. 6 Communication over ASD interfaces

specifies the internal behavior of the component and how it interacts with other components. All ASD components have both an interface model and a design model. ASD offers formal verification of these models, enabling engineers to verify the system design before starting implementation.

The hardware or software components in a system that are not developed using ASD are said to be foreign components. Foreign components can be some third party component, legacy code or hand written code. They do not necessarily have a design model but need to have an interface model that allows ASD to verify their interaction with the rest of the system.

ASD interfaces With the help of Fig. 6, we explain the terminology related to ASD, w.r.t. the SUL, that will be used later in this paper. The figure shows three components, *C*, *SUL* and *S*. Here, *C* is the client for *SUL* and *S* is the server for *SUL*. The interface model specifies application, notification and modeling interfaces for a component. A component provides services to other components via application interfaces. Over application interfaces, call events are sent and reply events are received. Notification interfaces exist to provide notification events to clients. The application and notification interfaces of the SUL collectively constitute the provided service of the SUL. In Fig. 6, the arrows directed toward and from the SUL are its inputs and outputs, respectively.

Modeling interfaces define modeling events, which represent spontaneous behavior of an interface model, i.e., they are anonymous notifications from servers to be replaced by concrete notifications in an implementation of the component. A modeling event may be *optional*, meaning the event may or may not occur, or it can be *inevitable*, meaning that if no other event occurs, this event will eventually occur.

An ASD component can use services from other ASD or foreign components. Used services define the link between the design model and the used interface model. For foreign components, only interface models are available and these

[3] https://learnlib.de/projects/automatalib/.

[4] http://www.verum.com/.

interface models ensure proper interfacing of foreign components with the rest of the system.

Stable failures refinement For every ASD engineered component, its interface model refines its design model modulo stable failures refinement, denoted by $\sqsubseteq_{SFR}$.[5] Let $\mathcal{L}_i$, where $i \in \{1, 2\}$, be two LTSes. The LTS $\mathcal{L}_1$ refines LTS $\mathcal{L}_2$ in stable failures semantics if and only if *weak traces* $(\mathcal{L}_2) \subseteq$ *weak traces*$(\mathcal{L}_1)$ and *failures*$(\mathcal{L}_2) \subseteq$ *failures*$(\mathcal{L}_1)$. A weak trace is a trace where internal actions, i.e., τ, are ignored. The set of failures contains information about all the actions that are not allowed for each state of that LTS. For a more detailed explanation of SFR, the reader is referred to [34].

4.5 mCRL2

mCRL2 is a formal model verification and validation language, with an associated toolset. Specifications written in mCRL2 can be converted to *LTSes*. Even in the presence of reference models, it is hard to manually compare models when the number of states and actions increases. We therefore use mCRL2 for behavioral reduction of models and checking formal relations between learned and reference models. For this purpose, we convert the Mealy machine to LTS. It is to be mentioned that SFR is called weak trace refinement in mCRL2 toolset.

5 Methodology

In this section, we present our methodology to infer the interface protocol of a software component. The proposed methodology combines equivalence checking and model learning. We use the terms *active learning result* for models obtained by learning the implementation of software components and *derived interface protocols* for interface protocols inferred from active learning results. The overview of the methodology is shown in Fig. 7 which summarizes the learning and validation steps for performing active learning and inferring the interface protocol of software component.

As explained in Sect. 4.2, applying active learning requires to identify the number of states in the SUL. This requires sound knowledge of the SUL, as over-approximation can lead to performance issues and under-approximation will result in learning incomplete behavior of the SUL. We learn MDE-based components, so that we can use the number of states from the behavior of the MDE models (i.e., reference models). In case of non-MDE software components, static code analysis [23] may help provide the estimate for number of states.

The presence of reference models also facilitates formal comparison with the learned results. However, our approach can be easily applied on legacy software components, as we already applied and validated our methodology on components with reference models. ASD components are particularly experimented with, as this research takes place at ASML which uses ASD for developing new control software. This provides us with a chance to validate our approach on industrial components.

While learning ASD software components, ASD design models are used as reference models for comparison with the models learned by the active learning algorithm by interacting with the implementation of the ASD software component. The derived interface protocols are compared with both the ASD design and interface models. This in turn provides the validation for the active learning technique as well. The confidence in the technique and learned results cannot be gained by learning an arbitrary black box component.

Our approach comprises two main steps. Based on the translation schemes presented in [29], a state space can be generated for ASD interface and design models. First, we learn the components with active learning and validate the learning results by comparing them with the reference state space of ASD design models of software components, indicated by 1 in Fig. 7. In the second step, we abstract the active learning results to the level of the desired interface. The inferred interface model is also verified formally to ensure it refines the ASD design model of that component modulo *SFR*, indicated by 2 in Fig. 7. Below we explain both steps of our methodology in detail.

5.1 Step 1: Learning the component's behavior and validating learned results

5.1.1 Learning component's behavior with active learning

Let *A.dm* be the design model of an ASD engineered software component and our *SUL* from now on. The *A.dm* implements the services specified in the interface model *A.im* and also uses services from other components. As stated earlier, *A.im* refines *A.dm* in stable failures semantics. The ASD code generator generates code from *A.dm*. An active learning algorithm interacts with this code to learn the behavior of the component. We perform this learning under the assumptions listed below. These assumptions relate to the conditions needed to be fulfilled by the SUL.

- *A* can be isolated from its environment.
- Outputs from *A* can be observed and *A* does not produce "spontaneous" outputs.

[5] In the conference version of this paper [5], we discussed that for every ASD engineered component, its interface model refines its design model modulo failures divergence refinement, denoted by $\sqsubseteq_{FDR}$. Experiments on a larger set of ASD components showed that this relation does not always hold. This led to further investigation and the correspondence with the company Verum (which developed ASD) revealed that the relation is stable failures refinement.

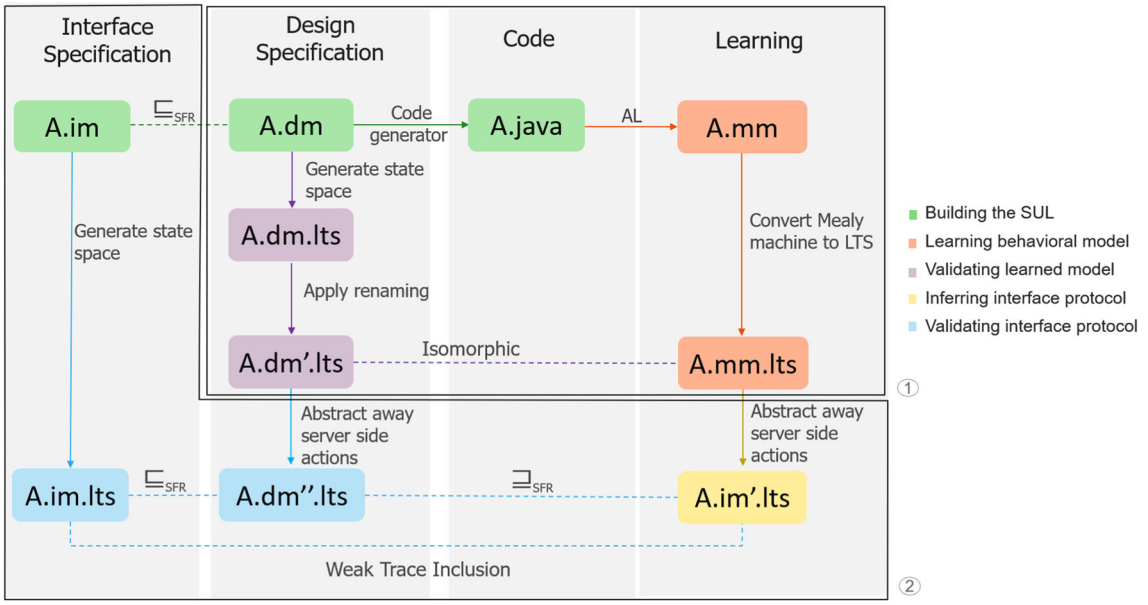


Fig. 7 Methodology: from component to interface protocol. The function arrows represent abstraction and transformation functions. Dotted lines represent formal relations. The extensions .im and .dm represent ASD interface and design model; .lts and .mm refer to labeled transition system and Mealy machine formalisms, respectively

- A can be initialized and reset to its initial state after every query.
- A is input enabled.
- A is deterministic.
- The internal behavior of A does not vary due to timing.
- The values of data parameters do not influence the behavior of A.

These assumptions are satisfied for ASD-generated code as ASD has a clear separation between interface and design models. This means that for any ASD design model, the interface it implements and the interface it requires are strictly defined. The outputs are return values of method calls and notifications for which a callback can be registered. As such, all outputs can be observed. ASD-generated code has no spontaneous behavior. Unless a function is called on the API of the component, no code is executed. Resetting is done by simply creating a new instance. Java takes care of destroying the instances through garbage collection. ASD components are not guaranteed to be input enabled. We solve this issue by implementing an interfacing protocol, explained in Sect. 7.2.2. ASD design models must be deterministic. ASD checks this through verification, i.e., it checks that rule cases for the same input in the same state do not have overlapping. All models are un-timed. In ASD, data cannot influence behavior. It can only be passed along to other components (including foreign components) and be used for verification purposes.

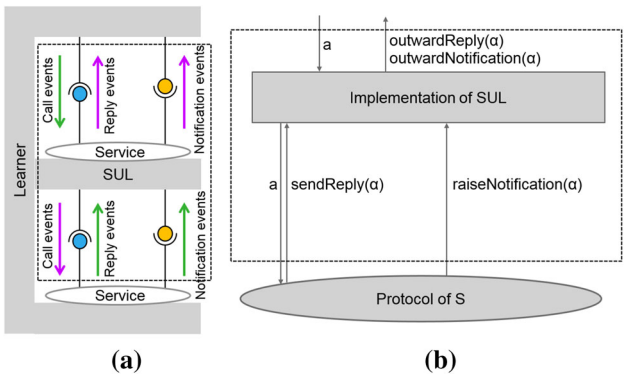
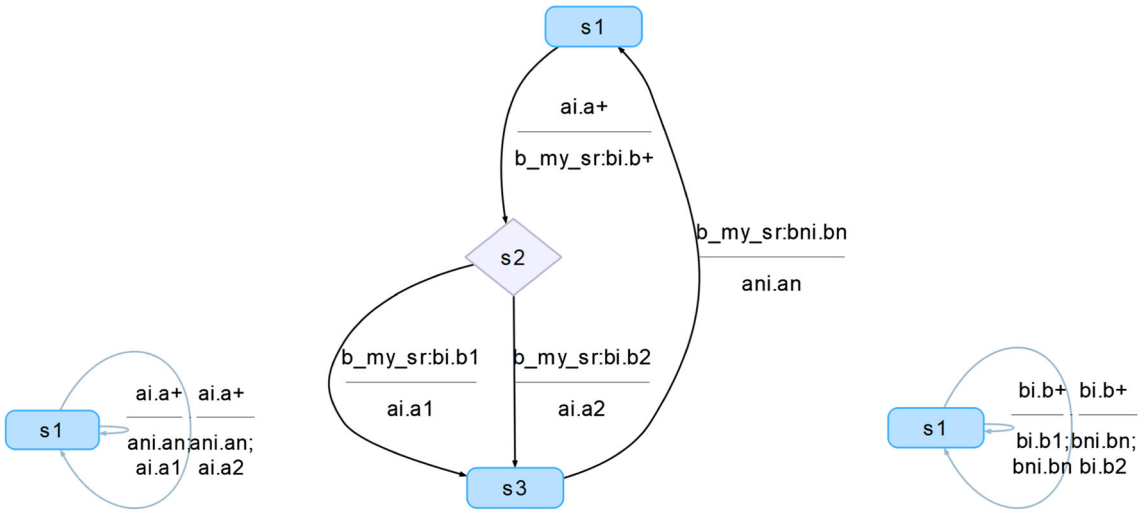


Fig. 8 Mapping between **a** active learning scope and **b** translated ASD models from [29]

Some of these assumptions may not hold in practice. However, for instance, if a SUL cannot be fully isolated, the SUL together with part of its environment could potentially be learned. Besides, there are ways to perform active learning without a means to reset the SUL, for instance, by using homing sequences [47]. For our work, we do not relax any of these assumptions.

5.1.2 Comparison of learned results with reference ASD design model

To validate the learning results, we check the equivalence between the learned results and the ASD design models. As mentioned before, we generate the state space for ASD design



(a) Interface model of A **(b)** Design model of A **(c)** Interface model of S

Fig. 9 ASD models of the SUL and Used service

and interface models based on the translation scheme presented in [29]. The mapping between active learning results and translations obtained from [29] is shown in Fig. 8. Before we compare the translated ASD design models with the active learning results, we apply renaming to synchronize the names of actions between translated and learned models. Without this renaming that is having same names for actions for both learned results and the reference models, we cannot check the equivalence between them. So from now on, as shown in Fig. 8, we use *(outwardNotification, triggerNotification)* for *notification events*, *(outwardReply, sendReply)* for *reply events* and α for *Call events*. To unify the naming between active learning results and reference models, we apply renaming on *A.dm.lts*.

To apply this renaming, we update the transition relation and define $f_{LTS}(\langle S, Act, \rightarrow, \hat{s}\rangle) = \langle S, Act, \{\rho(e) |\ e \in \rightarrow\}, \hat{s}\rangle$, where

$$\rho((s_1, a, s_2)) = \begin{cases} (s_1, \alpha, s_2) & \text{if} a \in \{\text{outwardReply}(\alpha), \\ & \text{sendReply}(\alpha), \\ & \text{outwardNotification}(\alpha), \\ & \text{raiseNotification}(\alpha)\} \\ (s_1, a, s_2) & \text{otherwise.} \end{cases}$$

The renaming is performed using the Automatalib. After renaming, the resulting LTS may contain a relatively large number of τ actions which can be reduced by applying divergence–preserving–branching–bisimulation [21] to increase understandability and readability of the resulting LTS. We choose divergence–preserving–branching–bisimulation for behavior reduction because it preserves the SFR relation.

Now having derived $A.dm'.lts$, we check whether the active learning result $A.mm.lts$ is isomorphic with its reference design model $A.dm'.lts$. Isomorphism is a very strong form of equivalence because if two LTSes are isomorphic, their structure is exactly the same. Isomorphic LTSes are only allowed to differ from each other in the labels of their states. If the isomorphism exists between active learning result and reference design model, it shows the learned model is correct and complete. This comparison is performed using the AutomataLib library.

5.2 Inferring and validating the interface protocol

The second step of our methodology concerns inferring and validating the interface protocols of the software components.

5.2.1 Inferring interface protocol

For an ASD component, the potential interface models for a given design model are those which are stable failures related to that design model, i.e., $\{im |\ im \sqsubseteq_{SFR} dm\}$, where im represents an interface model and dm represents a design model. This suggests that one design model can have several potential interface models. As we have already learned the component behavior, one of the immediate solutions can be to use the learned model itself as the interface protocol by applying the identity function. Every design model is stable failures related to itself so the learned behavioral model can be considered as the interface protocol. From the set of potential interface models, this is the most strict interface model as it allows only what is possible to be done according

to the design model. The most flexible interface model will be a flower model, i.e., a model which allows every possible action.

To abstract away the details of interaction with the services provided by other components, server side actions are abstracted away from *A.mm.lts* and *A.dm'.lts*. A function $f_{IM'}$ is introduced to perform this abstraction. This infers the interface protocol from *A.mm.lts*, the model obtained by active learning. For *A.mm.lts*, we define $\Sigma_{used} \subset \Sigma$ and $\Omega_{used} \subset \Omega$, where Σ_{used} and Ω_{used} represent input and output actions from used services, respectively. We define $Act_{used} = (\Sigma_{used} \cup \Omega_{used})$ containing actions from used services. Then, $f_{IM'}(\langle S, Act, \rightarrow, \hat{s}\rangle) = \langle S, Act, \{\rho(e) \mid e \in \rightarrow\}, \hat{s}\rangle$, where ρ is defined as follows:

$$\rho((s_1, a, s_2)) = \begin{cases} (s_1, \tau, s_2) & \text{if } a \in Act_{used} \\ (s_1, a, s_2) & \text{otherwise} \end{cases}$$

As for f_{LTS}, Automatalib is used for renaming and divergence–preserving–branching–bisimulation can be optionally applied to increase readability of the inferred interface protocol.

5.2.2 Validation of interface protocol

To confirm the inferred interface protocol $A.im'.lts$ is a valid interface protocol, we need to check how it formally relates to $A.dm''.lts$. Before applying this comparison check, we apply $f_{IM'}$ to $A.dm'.lts$ as well to hide actions from used services. Since we hide the same actions on both sides and the LTSes $A.dm'.lts$ and $A.mm.lts$ were isomorphic, surely $A.im'.lts \sqsubseteq_{SFR} A.dm''.lts$, SFR is reflexive in nature [34]. This validates $A.im'.lts$ as interface protocol for $A.dm''.lts$. We also verify that the original interface model, $(A.im.lts)$, refines the inferred interface protocol, $(A.im'.lts)$, modulo weak trace inclusion. We choose weak trace inclusion to be checked because the theory of active learning is based on traces, and the weak trace inclusion also takes care of the τ actions.

To guarantee that the translation from ASD models to mCRL2 models is correct and does not lose any behavior, we check that $A.im.lts$ is a stable failures refinement of $A.dm''.lts$, as expected.

In this way, we have an approach for the formal validation of the results learned from active learning. In the following section, we discuss a small example to illustrate our methodology.

6 Example

To demonstrate our approach, we designed an example ASD component *A* with one application and one notification interface. Figure 9 shows the interface and design models of component *A* and the interface model of its server component *S*. The *ai.a* and *bi.b* are the call events in application interfaces of *A* and *S*, respectively, *ani.an* is the notification event sent by *A* to *S*, and *bni.bn* is the notification event sent by *S* to *A*. The interface models of *A* and *S* are flower models, allowing every triggered behavior.

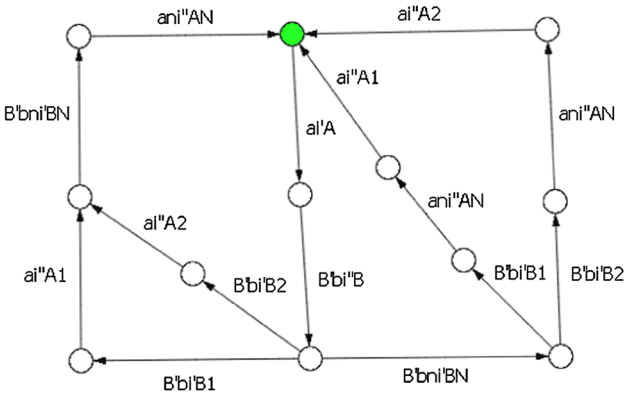


Fig. 10 Active learning result of component A, denoted by A.mm.lts in Fig. 7

We learn the behavior of the component *A* by applying active learning, using TTT as the learning algorithm and the Wp-method as the equivalence checking algorithm. Figure 10 shows the active learning result. The green circle in the state diagrams shows the initial state. To validate the active learning result, we check isomorphism between the learned result and state space generated from its ASD design model. Then, server side actions are being hidden using $f_{IM'}$ to infer the interface protocol. Divergence–preserving–branching–bisimulation is also applied to reduce the number of τ actions. The transformations are applied as specified in the methodology to obtain $A.mm.lts$ and $A.im'.lts$ as shown in Figs. 10 and 11, respectively.

The inferred interface protocol $(A.im'.lts)$, shown in Fig. 11, refines the learned model, $(A.dm'.lts)$, modulo stable failures refinement. The original interface model, $(A.im.lts)$, refines the inferred interface protocol, $(A.im'.lts)$, modulo weak trace inclusion.

7 Our framework for applying active learning and inferring interface protocols

In this section, we illustrate our Java framework which automates the proposed methodology for inferring interface protocols of software components using active learning in an automated manner. The framework consists of a state-of-the-art learning algorithm (TTT), several equivalence checking methods, the middleware to connect SUL to the learning and equivalence checking algorithm and the abstraction of the active learning result to obtain interface protocols from

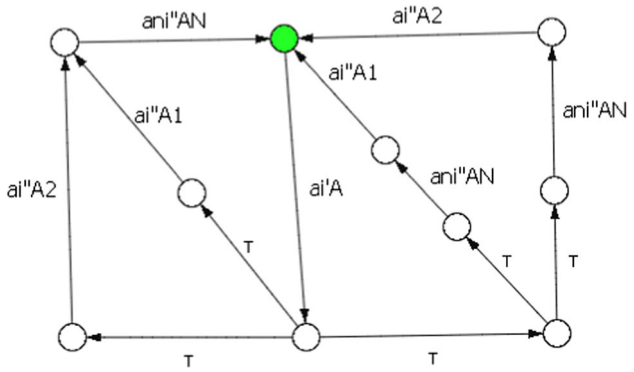


Fig. 11 Inferred interface model of component A, denoted by A.im'.lts in Fig. 7. *T* represents τ actions

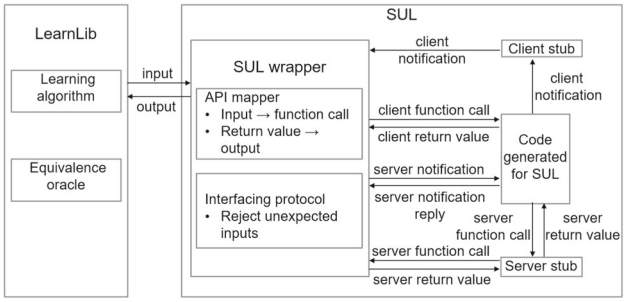


Fig. 12 Learning setup to connect LearnLib and MDE SUL

a learned component. For validation of the framework, it is augmented with the formal checking w.r.t. the reference models (if available). The framework can be used to apply active learning on any software, that is, MDE-based or traditionally developed software. Below, we explain the framework with reference to ASD components.

Overview of framework The toolchain used to develop our framework is as follows (Fig. 7): from A.im to A.java, ASD is used to automatically generate the code from the design model. LearnLib is used to learn the implementation of A.java with active learning and its output is A.mm. The toolchains for learning the behavior and generating the reference behavior are implemented separately, independent of each other, to reduce the chances for similar mistakes in both chains. The independent toolchains provide confidence that the comparison check between learned and reference models is a proper validation of the approach. Isomorphism is checked with AutomataLib. Renaming is done with Automatalib, and the SFR check is performed using the mCRL2 toolset. The state spaces from the interface and design models are generated, which provide the reference behavior of the software component. Next we describe the details of our framework.

7.1 Obtaining the SUL

To obtain the SUL, we generate Java code for the software component with the ASD toolset. We also collect information about the API of the SUL, such as the names of client and server function calls, the number of inputs and outputs, the number of states in the Mealy machine representation of the reference model. This information is later used during the analysis of the obtained results.

7.2 Learning the behavior

Figure 12 shows the learning setup for connecting the active learning tool (LearnLib) to the SUL. The setup consists of two parts: the SUL and the LearnLib. The details of the learning setup are discussed below.

7.2.1 Client and server stubs

The SUL is isolated from its environment. To learn all the interactions of the SUL with its environment, the clients and servers of the SUL are stubbed, similar to how this is done in the field of software testing. LearnLib can interact directly with the SUL or through the client and server stubs, depending on what is simpler to achieve. Client stubs store the client callbacks received from the SUL, so that the interfacing protocol of the SUL wrapper may retrieve them when needed. Server stubs are needed so that the SUL can create instances of its server components. Then, the SUL can invoke methods for server calls on those server components.

7.2.2 SUL wrapper

Typically, a SUL is not directly coupled to the LearnLib, but instead a SUL wrapper is used. The SUL wrapper is used as a bridge between the SUL and the LearnLib. In our case, the SUL wrapper includes an *API mapper* and an *interfacing protocol*.

Usually, inputs and outputs in the LearnLib are represented as strings. The API mapper maps input names to client function calls on the actual SUL and maps client return values from the SUL back to learner outputs. On the server side, the role of inputs and outputs is swapped with respect to the client side. The API mapper also creates and destroys instances of the SUL. A new instance of the SUL is used for each input sequence, to start from the same initial state.

To reject illegal inputs related to the particular state of the SUL, we inject an interfacing protocol between the LearnLib and the SUL. For instance, when the component is idle, there is no active server call. Thus, a server reply will be an illegal input. The protocol will reject such illegal inputs from the learner with an 'error' output. The interfacing protocol is designed in a maximally permissive way, that is, it only

rejects inputs as much as necessary. This way, all the relevant behavior of the SUL can be learned.

Furthermore, we use LearnLib to learn Mealy machines. Learning Mealy machines require alternating inputs and outputs, to avoid needing timeouts to decide whether or not there will be an output. In case of multiple consecutive outputs, we join them into a single output, for instance, client notifications collected in the client stub and subsequently joined with the next client reply or server call. In case there are no outputs, we inject an artificial output. Server notification replies are an example of artificial outputs.

7.2.3 Learning and testing algorithms

The framework can be easily used and integrated with different learning and equivalence checking algorithms. The equivalence oracles based on caching, logs and passive learning results are already integrated into the framework.

7.3 Deriving interface protocol

After learning the behavior of the software component, the actions from used services of the component are being hidden using AutomataLib. This provides the interface protocol of the software component. If a component interacts with several other components over different interfaces, choice can be made about which interface needs to be inferred.

7.4 Comparison with reference models

AutomataLib is used to check isomorphism between active learning results and reference ASD design models. The comparison result indicates whether the correct and complete behavior of the SUL is learned, and provides validation for our framework. SFR is checked between the derived interface protocol and the ASD reference design model using *ltscompare* from the mCRL2 toolset.

8 Evaluation of proposed approach

Using the framework described in the previous section, we perform an evaluation of our methodology on industrial MDE-based software components. We conducted this work at ASML, the world market leader in lithography systems. ASML develops complex machines called TWINSCAN, which produce integrated circuits (ICs). A TWINSCAN is a cyber-physical system containing software based on a codebase of over 50 million lines of code. The software architecture is component based, where components interact with each other over interfaces. ASML uses MDE techniques, such as ASD [7] to develop new components in a model-driven way. We present our study according to the guidelines of Reuneson et al. [49].

8.1 Goal

The study aims to evaluate our methodology by inferring interface protocols for industrial software components.

8.2 Study object

We apply active learning on industrial MDE-based control software components from a sub-system of ASML's TWINSCAN machine [67]. Using ASD technology, the components were initially designed in 2012 . The behavior of each component is represented by a control flow model, modeled as a Mealy machine. The generated code consists of more than 700 KLOC. Table 1 shows the range of inputs and states for the components under study. The number of inputs and the number of states in the behavioral model are key characteristics that determine the SUL complexity with respect to active learning [59]. The chosen components are nicely distributed over the number of inputs and number of states. Figure 14 shows the spread of our chosen components over these features.

8.3 Experimental setup

Using the learning setup shown in Fig. 12, we perform active learning. The learning algorithm selected is the TTT, as implemented in the LearnLib. TTT is highly efficient due to redundancy free data structures. Testing is known to hinder the performance of active learning process significantly [67]. For testing the correctness of the hypothesis, we experimented with several equivalence checking methods, explained in Sect. 4.2 to learn the maximum number of components. We evaluated the impact of several testing methods on the performance of the active learning process as well.

We chose two conformance testing techniques, the Wp-method and the H-ADS. The Wp-method is a traditional conformance testing method implemented in the LearnLib, and H-ADS has recently shown promising results when applied for learning a large industrial software component. H-ADS reduces the test suite and is therefore expected to outperform the Wp-method. We combine both these conformance testing techniques with the logs- and cache-based oracles. Figure 13 depicts the setup we used for these experiments.

Timeout of experiments Active learning can learn the complete model of a software component if (i) provided with the correct number of expected states in the active learning result, and (ii) continued for a long enough period of time, which increases exponentially with the increase in the number of states. As this is not practically possible, we set the timeout

Table 1 Features summary of 202 components

	Inputs				States
	Client func. calls	Server ret. values	Server notifs.	Total	
Min	1	0	0	1	1
Max	77	176	32	201	8446

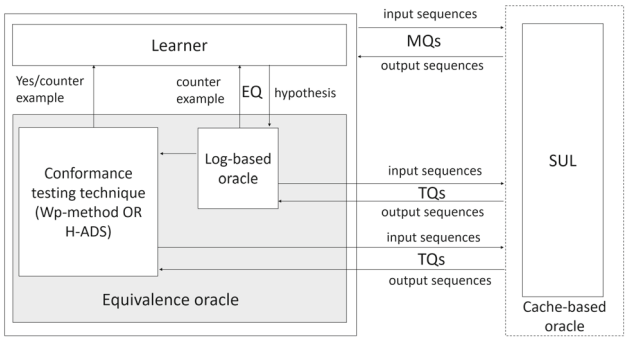


Fig. 13 Experimental setup for applying different equivalence checking methods. Besides the Wp-method, we experiment with H-ADS as well. An instance of Cache-based oracle is also integrated into the framework

for learning a single component to a duration of 1 h. In one experiment, we extended the timeout to 8 h, 2 days and 8 days (Sect. 8.7). This extension did not benefit the active learning process significantly, so we settled for a timeout of 1 h. If a timeout occurs, the last hypothesis produced during learning is used for comparison with the reference model. Since in case no timeout occurs, the final testing round will not find a counterexample, this may be a lengthy testing round. After obtaining and comparing the active learning results with the reference models, i.e., checking for isomorphism between learned and reference models, we remove the server side actions to infer the interface protocols. We later form our conclusions based on the components learned with in 1 h.

8.4 Hardware setup

All the experiments are executed on high-performance cluster with Skylake Gold 6126 CPU, 2.6 Ghz, 190 gigabytes memory and RHEL 7.2 operating system.

8.5 Metrics

We record the *number of components learned* and the *total learning time* to measure the performance of the active learning process. For all the learned components, we are able to derive interface protocols.

The *total learning time* is the sum of the time spent in the learning and the testing phases of active learning. The time spent in then deriving the actual interface protocols by hiding server side actions is negligible and therefore not taken into account.

8.6 Stability of measurements

Since we ran our experiments on a cluster and one of our metrics is the time spent in the active learning process, there can be noise in our measurements due to factors like shared memory, caching, garbage collection and CPU throttling. Therefore, we repeat each experiment with different testing algorithms 30 times to ensure minimization of the effect of noise. For a given testing algorithm, for each model we calculate the coefficient of variation (standard deviation/mean) for the time spent in active learning over these 30 runs. If the value of coefficient of variation was less than 0.3, the measurements are acceptable and we average the 30 values [22]. If it was larger, we further analyzed the data and saw that in such cases there are 1–3 outliers contributing to the greater values of coefficient of variation. We therefore take out the outliers and average the rest of the values. In this way, we ensure stability of our measurements.

8.7 Results

8.7.1 Number of components learned

As both the theoretical (worst-case) bounds for learning and testing algorithms are influenced more by the number of states than by the number of inputs [16,26], we grouped the components according to the number of states for the analysis of our results.

The number of components learned by using different equivalence checking methods is shown in Table 2. The number of states of our chosen software components ranges from 1 to 8446. The first column of the table shows the range of number of states. The second column shows the number of components belonging to a particular range of number of states. There are more components with number of states from 1 to 100, so we have subdivided those components in smaller-sized groups with intervals of 25. From 101 to 200, the interval is 50, and then till 800, the interval is 100. We do not have any component in the range between 401 and 500 states. Above 800, we have only 9 components. The third column shows the number of components that are not learned by any equivalence oracle. The fourth column and onwards of the table show the number of components learned by each

Table 2 Number of components learned by each equivalence oracle grouped according to the number of states

No. of states	Comps.	Comps. not learned by any oracle	WP	Cache + Wp	Log + Wp	Cache + Log + Wp	H-ADS	Cache + H-ADS	Log + HADS	Cache + Log + H-ADS
1–25	99	0	88	97	94	99	90	94	94	98
26–50	34	4	9	20	22	27	13	21	23	27
51–75	10	3	2	5	2	6	2	4	4	5
76–100	16	5	1	8	0	9	1	7	3	11
101–150	7	6	0	0	0	1	0	0	0	0
151–200	7	6	0	1	0	1	1	1	1	1
201–300	1	1	0	0	0	0	0	0	0	0
301–400	3	3	0	0	0	0	0	0	0	0
401–500	–	–	–	–	–	–	–	–	–	–
501–600	4	3	0	1	0	1	0	1	0	1
601–700	9	5	0	0	0	4	0	0	0	0
701–800	3	1	0	2	0	2	0	2	0	2
801-onwards	9	9	0	0	0	0	0	0	0	0
Total	202	46	100	134	118	150	107	130	125	145

Cache + *Log* + *Wp* learned maximum number of components and *Wp* learned the least. Out of 202 components, 46 components were not learned by any equivalence oracle and 156 components were learned by at least one equivalence oracle

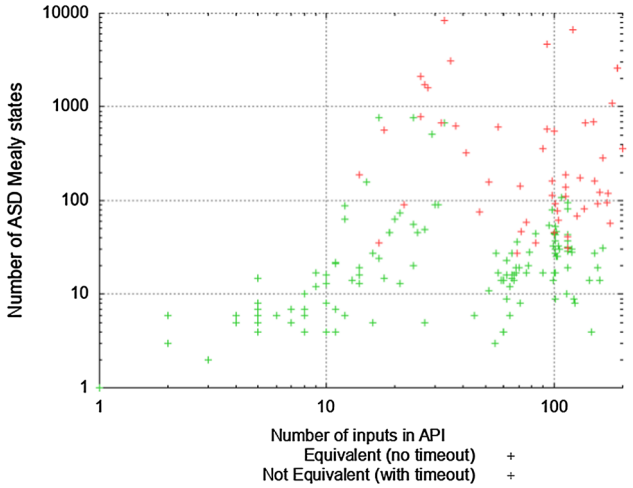


Fig. 14 Inputs vs states with learning results for *Cache* + *Log* + *Wp*

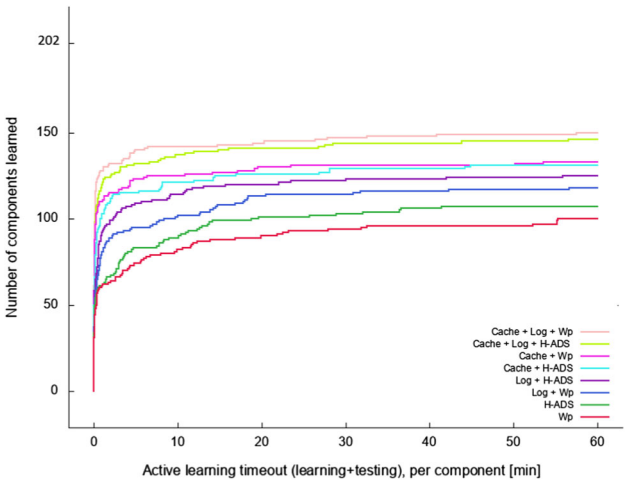


Fig. 15 Comparison of different equivalence oracles. Here we only include components for which learning is finished with in 1 h

equivalence oracle from a particular group. The last row of the table shows the total number of components learned by each oracle.

Overall, the smallest number of components is learned by the Wp-method. This is expected because the Wp-method generates longer test sequences and therefore takes more time to find counterexamples. H-ADS performed better than the Wp-method. We see a considerable increase in the number of learned components when either the Wp-method or H-ADS is combined with caching, and even more when logs are used. The logs represent the real behavior of the software. Using logs, counterexamples are found without sending queries to the SUL. This makes the whole process quite faster, resulting in more components being successfully learned within the time limit of 1 h.

A closer look at the table shows that for components with smaller number of states (0–25), all the algorithms learned the vast majority if not all of the components. As the number of states grows, the difference in performance of the oracles becomes more pronounced. Especially when the number of states rises above 100, except for one component that is also learned by H-ADS, the components are only learned when we aid conformance testing techniques with cache and logs. Out of 202 components, there exist 46 components that are not learned by any testing method and 156 components are learned by at least one of them.

The maximum number of components is learned by *Cache* + *Log* + *Wp*. There still exist six components that are not learned by this best performing oracle, but that are learned by at least one other oracle. The relation of the fully and partially learned components by *Cache* + *Log* + *Wp* to the size of the components is shown in Fig. 14. Regarding the number of inputs in the API, components with smaller as well as components with larger numbers of inputs have been correctly and completely learned. Regarding the number of states, as the number of states increases, less components can be learned completely. The largest component w.r.t. the number of states learned by *Cache* + *Log* + *Wp* has 24 inputs and 767 states; the largest component w.r.t. the number of inputs learned has 164 inputs and 31 states. For models with a large number of states and a large number of inputs, the number of states has more impact than the number of inputs, as discussed before.

Another important finding from Table 2 is that the number of states is not the only criterion to measure the complexity of a component with respect to active learning. For instance, there are four components with 201–400 states. None of these components is learned by any equivalence oracle. However, components with even more states are still learned. This shows that there might be some difficult design patterns that make a particular component harder to be learned, apart from the number of states. To improve the active learning performance, such design patterns can be looked into particularly, as done previously [67].

8.7.2 Time spent in active learning

To analyze how much cost different equivalence checking methods incur to the active learning process, we plot the time spent in active learning by each equivalence oracle against the number of components learned, in Fig. 15. On the x-axis, the total time spent on active learning is shown, while the y-axis presents the number of components learned within that time, by a certain equivalence checking method. Each line in the plot represents one equivalence oracle, as per the legend.

The higher a line appears in the graph at some particular time instant, the greater the number of components learned by the equivalence oracle represented by that line and thus better the performance of the equivalence oracle.

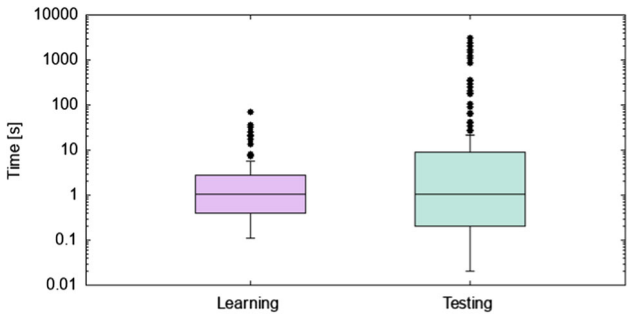


Fig. 16 Learning time versus testing time in active learning, for the 150 fully learned components by $Cache + Log + Wp$

Similarly, the more to the left a line appears in the graph for some particular number of learned components, the faster the equivalence oracle represented by that line learned that number of components. The graph shows that using better equivalence checking methods, more components can be learned completely and in less time. However, the rate at which the number of learned components increases decreases over time, as can be seen by the flattened ends of the lines in the graph. This shows that *only* granting more time to the active learning process does not help; instead better techniques and strategies are required to find counterexamples faster to finish learning.

At the end of the 1 h maximum time per component, $Cache + Log + Wp$ learns the highest number of components within the given time bound. *Wp* performed worst among all equivalence oracles. The above results show that caching should always be turned on while performing active learning. If available, the real-time logs of software execution should also be used. Applying these two techniques greatly reduces the number of test queries, thus significantly improving the performance of the active learning process.

We analyze the time spent in the learning and testing phases of active learning for $Cache + Log + Wp$. Figure 16 shows boxplots for the duration of the learning and testing algorithms of the execution of active learning phase. Only models that are correctly and completely learned within 1 h, without timeout, are included for this analysis. The plot shows that testing takes significantly more time than learning. Using this equivalence oracle, learning and testing together take approximately 52 min. By maximum, learning takes a little above one minute. As the total learning and testing time increases, testing becomes more and more dominant. Thus, the performance of active learning process is highly dominated by the testing time.

Effect of increasing learning timeout One may argue that increasing the timeout can increase the number of fully learned models. To investigate this, we increased the timeout of the experiment using Wp-method for components that were only partially learned with the Wp-method from 1 to 8 h. This increased the percentage of fully learned models from about 53% to about 57%. Increasing the timeout further to 2 and 8 days does not bring any benefit, as no additional models are fully learned. This shows that only giving more time to the active learning process is not the solution; instead, efficient testing methods may help improve performance of the active learning process significantly.

8.7.3 Statistical significance test

Given a set of equivalence checking methods, we want to analyze which testing method performed significantly better than others. For this purpose, we perform a statistical significance test on the time spent by different testing methods in the active learning process. For this statistical analysis, we include the components learned by *all* testing methods with in 1 h. Out of 202, 100 components are learned by all equivalence checking methods.

To perform statistical comparison, we employ the $\widetilde{\mathbf{T}}$ procedure proposed by Konietschke et al. [33] and successfully applied in empirical software engineering studies [28,61,68]. We prefer $\widetilde{\mathbf{T}}$ to more traditional two-phase approaches such as ANOVA followed by pairwise *t*-tests or Kruskal–Wallis test followed by pairwise Mann–Whitney–Wilcoxon tests, since $\widetilde{\mathbf{T}}$ avoids possible inconsistencies between the overall testing (e.g., using ANOVA or Kruskal–Wallis) and the subsequent pairwise tests [19]. Specifically, we apply $\widetilde{\mathbf{T}}$ to detect relations between all pairs of testing methods employing the so-called Tukey-type contrasts [58] in combination with the traditional family-wise error rate threshold of 5%.

The results of the $\widetilde{\mathbf{T}}$ procedure are summarized in Table 3. As we have eight equivalence checking methods in total, there are $\frac{8*7}{2} = 28$ pairs to be compared. For the sake of brevity, we only list the pairs that showed significant statistical difference. The second column shows the pairs of the testing methods to be compared. For a particular pair, the null hypothesis states that *there is no statistically significant difference between the performance of equivalence oracles of that pair*. For each pair, we analyze the 95% confidence interval to test whether the corresponding null hypothesis can be rejected. The third and fourth columns show the lower and upper boundary for a confidence interval, respectively. If the lower boundary for a pair (A, B) is greater than zero, then we claim that A performed significantly better than B. Similarly, if the upper boundary for a pair (A, B) is less than zero, then we claim that A performed significantly worse than B. Out of 28 pairs, the null hypothesis can be rejected for twelve pairs (p value < 0.05). For all pairs listed in the table, the lower boundary is greater than zero. This shows that, on the set of components learned by all testing methods, the first testing method in the pair performed better than the second. We see that the first testing methods for all the pairs include *Cache* and few of them involve *Log* as well. This is in agreement

Table 3 Results of the $\widetilde{\mathbf{T}}$ procedure. Each row shows the pair of equivalence oracles to be compared, the lower and upper bounds for the confidence interval and the p value for that pair

No.	Pair	Lower	Upper	p-value
1	Cache + H-ADS, Cache + Log + Wp	0.243	0.473	5.54e−03
2	Cache + H-ADS, Cache + Wp	0.228	0.454	1.24e−03
3	Cache + H-ADS, Wp	0.552	0.781	9.60e−04
4	Cache + Log + H-ADS, Wp	0.577	0.803	1.67e−05
5	Cache +Log+ Wp, H-ADS	0.593	0.812	4.06e−05
6	Cache + Log + Wp, Log + H-ADS	0.611	0.822	2.13e−07
7	Cache + Log + Wp, Log + Wp	0.589	0.809	4.49e−06
8	Cache + Log + Wp, Wp	0.655	0.856	1.01e−09
9	Cache + Wp, H-ADS	0.615	0.826	1.46e−07
10	Cache + Wp, Log + H-ADS	0.617	0.825	1.60e−07
11	Cache + Wp, Log + Wp	0.609	0.821	2.53e−07
12	Cache + Wp, Wp	0.675	0.867	3.26e−11

with our previous inference that testing methods perform significantly better when aided with logs and cache.

9 Conclusions and future work

In this paper, we have presented a two-step methodology to infer interface protocols of software components using active learning. First, the active learning result obtained from learning the implementation provides an insight into the software, thus facilitating adaptations that need to be made to the component or documentation that needs to be updated. We then abstract the learned model to infer an interface protocol. We gain confidence in the derived interface protocols through equivalence checking and validating formal relations between learned models and reference models. The inferred interface protocols provide a starting point for several engineering and maintenance activities of the software components that we have discussed in Sect. 2 of the paper. To the best of our knowledge, this is the first work to use active learning technique for inferring interface protocols of software components.

We also presented an automated framework for applying active learning to software components and then inferring their interface protocols. Using the framework, we performed an evaluation for our methodology on a set of MDE-based industrial software components. We performed experiments using several equivalence checking methods to learn maximum number of software components. The results show that faster equivalence checking techniques contribute significantly to increase efficiency of the active learning process. We observed that few components, despite having less number of states, cannot be learned by any equivalence checking method. Such components are planned to be studied in detail to discover the design patterns that make learning so difficult. In total, we were able to derive interface protocols for 156 out of 202 components. We want to use these interface protocols for engineering and maintenance tasks, few of which are already discussed in the paper.

In the future, we also plan to infer interface protocols for industrial legacy components using the proposed methodology and framework. We see several research challenges in that direction. Data-guarded behavior is unavoidable when dealing with the legacy software and therefore will be a concern for future work. Also, having the reference models for our chosen software components, we could find the number of states from the reference models to feed the conformance testing techniques. If the number of states is unknown, as will be the case for legacy software, we may give a low or high estimate resulting in an incomplete result or influencing the scalability, respectively. Static code analysis [23] may help in providing this estimate for non-MDE software components.

Acknowledgements This research was partially supported by Eindhoven University of Technology and ASML Netherlands B.V., carried out as part of the IMPULS II project, and 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.' The authors would also like to express deep gratitude to Dennis Hendriks and Leonard Lensink for providing support on the design of experiments and implementation, Alexander Serebrenik for advice on results analysis and Alessandro di Bucchianico for discussions on statistical tests.

References

1. Aarts, F.: Tomte: bridging the gap between active learning and real-world systems. Doctoral Dissertation, Radboud University (2014)
2. Aarts, F., De Ruiter, J., Poll, E.: Formal models of bank cards for free. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 461–468. IEEE (2013)
3. Al Duhaiby, O., Mooij, A., van Wezep, H., Groote, J.F.: Pitfalls in applying model learning to industrial legacy software. In: International Symposium on Leveraging Applications of Formal Methods, pp. 121–138. Springer (2018)
4. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
5. Aslam, K., Luo, Y., Schiffelets, R., van den Brand, M.: Interface protocol inference to aid understanding legacy software components. In: MODELS Workshops, pp. 6–11 (2018)
6. Atkinson, C., Bunse, C., Gross, H.G., Peper, C.: Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends, vol. 3778. Springer, Berlin (2005)
7. Broadfoot, G.H., Broadfoot, P.J.: Academia and industry meet: some experiences of formal methods in practice. In: Software Engineering Conference, 2003. Tenth Asia-Pacific, pp. 49–58. IEEE (2003)
8. Buse, R.P., Weimer, W.R.: Automatic documentation inference for exceptions. In: Proceedings of the International Symposium on Software Testing and Analysis, pp. 273–282. Citeseer (2008)
9. Cai, X., Lyu, M.R., Wong, K.F., Ko, R.: Component-based software engineering: technologies, development frameworks, and quality assurance schemes. In: Proceedings of Seventh Asia-Pacific Software Engineering Conference. APSEC 2000, pp. 372–379. IEEE (2000)
10. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Active learning for extended finite state machines. Formal Asp. Comput. **28**(2), 233–263 (2016)
11. Chalupar, G., Peherstorfer, S., Poll, E., De Ruiter, J.: Automated reverse engineering using lego®. In: 8th USENIX Workshop on Offensive Technologies, WOOT, vol. 14, pp. 1–10 (2014)
12. Cho, C.Y., Shin, E.C.R., Song, D., et al.: Inference and analysis of formal models of botnet command and control protocols. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 426–439. ACM (2010)
13. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. **4**(3), 178–187 (1978)
14. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans. Softw. Eng. **30**(12), 859–872 (2004)
15. Diaz, M., Juanole, G., Courtiat, J.P.: Observer-a concept for formal on-line validation of distributed systems. IEEE Trans. Softw. Eng. **20**(12), 900–913 (1994)
16. Dorofeeva, R., El-Fakih, K., Maag, S., Cavalli, A.R., Yevtushenko, N.: FSM-based conformance testing methods: a survey annotated with experimental evaluation. Inf. Softw. Technol. **52**(12), 1286–1297 (2010)
17. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: International Symposium of Formal Methods Europe, pp. 500–517. Springer (2001)
18. Fujiwara, S., Bochmann, Gv, Khendek, F., Amalou, M., Ghedamsi, A.: Test selection based on finite state models. IEEE Trans. Softw. Eng. **17**(6), 591–603 (1991)
19. Gabriel, K.R.: Simultaneous test procedures—some theory of multiple comparisons. Ann. Math. Stat. **40**, 224–250 (1969)
20. González, C.A., Cabot, J.: Formal verification of static software models in MDE: a systematic review. Inf. Softw. Technol. **56**(8), 821–838 (2014)
21. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press, Cambridge (2014)
22. Gross, D., Harris, C.M.: Fundamentals of Queueing Theory. Wiley, New York (2008)
23. Hamilton, V.: The use of static analysis tools to support reverse engineering. In: IEE Colloquium on Reverse Engineering for Software Based Systems, pp. 6/1–6/4 (1994)
24. Howar, F., Giannakopoulou, D., Rakamarić, Z.: Hybrid learning: interface generation through static, dynamic, and symbolic analysis. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis, pp. 268–279. ACM (2013)
25. Hutchinson, J., Rouncefield, M., Whittle, J.: Model-driven engineering practices in industry. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 633–642. ACM (2011)
26. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: a redundancy-free approach to active automata learning. In: RV, pp. 307–322 (2014)
27. Isberner, M., Howar, F., Steffen, B.: The open-source Learnlib. In: International Conference on Computer Aided Verification, pp. 487–495. Springer (2015)
28. Jongeling, R., Sarkar, P., Datta, S., Serebrenik, A.: On negative results when using sentiment analysis tools for software engineering research. Empir. Softw. Eng. **22**(5), 2543–2584 (2017)
29. Jonk, R.J.W.: The semantic of alias defined in MCRL2. Master's Thesis, Eindhoven University of Technology, The Netherlands (2016)
30. Kalbarczyk, Z., Iyer, R.K., Wang, L.: Application fault tolerance with armor middleware. IEEE Internet Comput. **9**(2), 28–37 (2005)
31. Kalbarczyk, Z.T., Iyer, R.K., Bagchi, S., Whisnant, K.: Chameleon: a software infrastructure for adaptive fault tolerance. IEEE Trans. Parallel Distrib. Syst. **10**(6), 560–579 (1999)
32. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
33. Konietschke, F., Hothorn, L.A., Brunner, E.: Rank-based multiple test procedures and simultaneous confidence intervals. Electron. J. Stat. **6**, 738–759 (2012)
34. Laveaux, M., Groote, J.F., Willemse, T.A.: Correct and efficient antichain algorithms for refinement checking. In: International Conference on Formal Techniques for Distributed Objects, Components, and Systems, pp. 185–203. Springer (2019)
35. Le Goues, C., Forrest, S., Weimer, W.: Current challenges in automatic software repair. Softw. Qual. J. **21**(3), 421–443 (2013)
36. Lee, D., Yannakakis, M.: Testing finite-state machines: state identification and verification. IEEE Trans. Comput. **43**(3), 306–320 (1994)
37. Leemans, M., van der Aalst, W.M., van den Brand, M.G.: The statechart workbench: enabling scalable software event log analysis using process mining. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 502–506. IEEE (2018)
38. Lehman, M.M.: Laws of software evolution revisited. In: European Workshop on Software Process Technology, pp. 108–124. Springer (1996)
39. Loose, R.: Component-wise supervisory controller synthesis using existing plant models in a client/server structure. Master's Thesis, Eindhoven University of Technology, The Netherlands (2017)
40. Lorenzoli, D., Mariani, L., Pezz, M.: Automatic generation of software behavioral models. In: 2008 ACM/IEEE 30th International Conference on Software Engineering, pp. 501–510 (2008)
41. Mohagheghi, P., Gilani, W., Stefanescu, A., Fernandez, M.A.: An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. Empir. Softw. Eng. **18**(1), 89–116 (2013)
42. Petrenko, A., Yevtushenko, N., Lebedev, A., Das, A.: Nondeterministic state machines in protocol conformance testing. In:

Proceedings of the Sixth International Workshop on Protocol Test Systems VI, pp. 363–378 (1993)
43. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans. Softw. Eng. **28**(11), 1056–1076 (2002)
44. Plattner, B., Nievergelt, J.: Special feature: monitoring program execution: a survey. Computer **11**, 76–93 (1981)
45. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: Learnlib: a framework for extrapolating behavioral models. Int. J. Softw. Tools Technol. Transf. (STTT) **11**(5), 393–407 (2009)
46. Ramanathan, M.K., Grama, A., Jagannathan, S.: Static specification inference using predicate mining. In: ACM SIGPLAN Notices, vol. 42, pp. 123–134. ACM (2007)
47. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Inf. Comput. **103**(2), 299–347 (1993)
48. Robillard, M.P., Bodden, E., Kawrykow, D., Mezini, M., Ratchford, T.: Automated API property inference techniques. IEEE Trans. Softw. Eng. **39**(5), 613–637 (2013)
49. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. **14**(2), 131 (2009)
50. Schmidt, D.C.: Model-driven engineering. Comput. IEEE Comput. Soc. **39**(2), 25 (2006)
51. Schuts, M., Hooman, J., Vaandrager, F.: Refactoring of legacy software using model learning and equivalence checking: an industrial experience report. In: International Conference on Integrated Formal Methods, pp. 311–325. Springer (2016)
52. Shatnawi, A., Seriai, A.D., Sahraoui, H., Alshara, Z.: Reverse engineering reusable software components from object-oriented APIS. J. Syst. Softw. **131**, 442–460 (2017)
53. Smeenk, W., Moerman, J., Vaandrager, F., Jansen, D.N.: Applying automata learning to embedded control software. In: International Conference on Formal Engineering Methods 2015, pp. 67–83. Springer (2015)
54. Sommerville, I., Cliff, D., Calinescu, R., Keen, J., Kelly, T., Kwiatkowska, M., Mcdermid, J., Paige, R.: Large-scale complex IT systems. Commun. ACM **55**(7), 71–77 (2012)
55. Steinbauer, G., Wotawa, F., et al.: Detecting and locating faults in the control software of autonomous mobile robots. In: IJCAI, vol. 5, pp. 1742–1743. Citeseer (2005)
56. Stramaglia, S.: Data integrity for compaq non-stop Himalaya servers (1999)
57. Tillmann, N., Chen, F., Schulte, W.: Discovering likely method specifications. In: International Conference on Formal Engineering Methods, pp. 717–736. Springer (2006)
58. Tukey, J.W.: Quick and dirty methods in statistics. Part II. Simple analyses for standard designs. American Society for Quality Control, pp. 189–197 (1951)
59. Vaandrager, F.: Model learning. Commun. ACM **60**(2), 86–95 (2017)
60. van Beek, D.A., Fokkink, W., Hendriks, D., Hofkamp, A., Markovski, J., Van De Mortel-Fronczak, J., Reniers, M.A.: CIF 3: model-based engineering of supervisory controllers. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 575–580. Springer (2014)
61. Vasilescu, B., Serebrenik, A., Goeminne, M., Mens, T.: On the variation and specialisation of workload—a case study of the Gnome ecosystem community. Empir. Softw. Eng. **19**(4), 955–1008 (2014)
62. Vierhauser, M., Rabiser, R., Grünbacher, P.: Requirements monitoring frameworks: a systematic review. Inf. Softw. Technol. **80**, 89–109 (2016)
63. Walkinshaw, N., Bogdanov, K.: Inferring finite-state models with temporal constraints. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 248–257. IEEE Computer Society (2008)
64. Walkinshaw, N., Taylor, R., Derrick, J.: Inferring extended finite state machine models from software executions. Empir. Softw. Eng. **21**(3), 811–853 (2016)
65. Wonham, W.M.: Supervisory Control of Discrete-Event Systems. Springer, Berlin (2015)
66. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: Proceedings of the 28th international conference on Software Engineering, pp. 282–291. ACM (2006)
67. Yang, N., Aslam, K., Schiffelers, R., Lensink, L., Hendriks, D., Cleophas, L., Serebrenik, A.: Improving model inference in industry by combining active and passive learning. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 253–263. IEEE (2019)
68. Yu, Y., Wang, H., Yin, G., Wang, T.: Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? Inf. Softw. Technol. **74**, 204–218 (2016)
69. Zulkernine, M., Seviora, R.: Towards automatic monitoring of component-based software systems. J. Syst. Softw. **74**(1), 15–24 (2005)

Kousar Aslam is currently finalizing her Ph.D. study at the Software Engineering and Technology group of the Eindhoven University of Technology. The research was conducted in close collaboration with the software research group of ASML, the world market leader in lithography machines. Kousar obtained her master's degree in computer science from Sabanci University, Turkey. Her research interests include software reverse engineering, model driven engineering and software evolution.

Loek Cleophas is an assistant professor in Software Engineering Technology at Eindhoven University of Technology, where he obtained his doctorate in computer science and engineering, and a research fellow at Stellenbosch University. His work has covered taxonomies and toolkits of (tree and text) pattern matching algorithms; model-driven virtualization of high-tech systems; and recently analyzing collections of models, with a focus on extracting variability and commonality information from them. He has experience in industry in the Netherlands and the USA, and at universities in South Africa, Sweden and Germany, on research funded by various national and international projects and industrial partners.

Ramon Schiffelers is leading the Software Research Group at ASML, world's leading provider of lithography systems for the semiconductor industry, and is assistant professor at the Department of Mathematics and Computer Science at the Eindhoven University of Technology. He is positioned at the interface between scientific knowledge from academia and its application in the industry. Besides innovative products, this resulted in long-term collaborative research and innovation between ASML, academia and several knowledge institutes.

Mark van den Brand is a full professor of Software Engineering and Technology in the Department of Mathematics and Computer Science, and a visiting professor at Royal Holloway, University of London. His current research activities are on model driven engineering, domain specific languages, meta-modeling, model management, digital twins and automotive software engineering. His research is industry inspired; he works with most of the high-tech companies in the Eindhoven (The Netherlands) region. He has been an invited lecturer and keynote speaker at various conferences, workshops and doctoral schools. He was and is member of PCs on workshops and conferences related to software engineering language engineering, rewriting, reverse engineering, and software maintenance. He initiated the special issues of Science of Computer Programming devoted to academic software development (Experimental Software and Toolkits) and since 2007 has been guest editor of six of these. He is on the editorial board of the journals Science of Computer Programming, Open Computer Science and Computer Languages (COLA). He is Editor-in-Chief of the Journal on Automotive Software Engineering. He is associate Editor-in-Chief of the Software Section of the Science of Computer Programming. He is deputy Editor-in-Chief of platinum open access journal JOT.