Interface protocol inference to aid understanding legacy software components

Kousar Aslam[1], Yaping Luo[1,2], Ramon Schiffelers[1,3], Mark van den Brand[1]
1. Eindhoven University of Technology, The Netherlands 2. ESI (TNO) 3. ASML
{k.aslam,y.luo2,r.r.h.schiffelers,m.g.j.v.d.brand}@tue.nl

ABSTRACT

More and more high tech companies are struggling with the maintenance of legacy software. Legacy software is vital to many organizations, so even if its behavior is not completely understood it cannot be thrown away. To re-factor or re-engineer the legacy software components, the external behavior needs to be preserved after replacement so that the replaced components possess the same behavior in the system environment as the original components. Therefore, it is necessary to first completely understand the behavior of components over the interfaces, i.e., the interface protocols, and preserve this behavior during the software modification activities.

For this purpose, we present an approach to infer the interface protocols of software components, from the behavioral models of those components learned with a blackbox technique, called active automata learning. We then perform a formal comparison between learned models and reference models ensuring the behavioral relations are preserved. This provides a validation for the learned results, thus developing confidence in applying the active learning technique to reverse engineer the legacy software components in the future.

1 INTRODUCTION

Large scale software systems are inherently complex. The software also undergoes 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 in [15]. These factors turn the software into so-called legacy software which becomes harder to understand and costly to maintain. The legacy software usually implements crucial domain logic which cannot be thrown away or easily replaced. However, for different reasons, such as change of technology or performance issues, the legacy components may need to be re-factored or re-engineered. To support this, the implicit domain logic (behavioral models) of these components and interface protocols implemented between the components need to be extracted and learned.

Several techniques exist in literature for reverse engineering of the existing components. These techniques can be widely 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 [11]. Dynamic software analysis techniques analyze the actual execution of the software, either by analyzing execution traces (passive learning [14]), or by interactive interaction with software components (active automata learning or active learning [21]). In this work, we have explored the active learning technique as the technique shows promising results when applied to learning reactive systems [4][17][20].

Fiterău-Broştean et al. [6][7] combine active learning and model checking to learn network protocol implementations. Schuts et al. [20] combine active learning and equivalence checking to gain confidence in the re-factoring of legacy components. Equivalence is checked between active learning results obtained from the legacy and re-factored component using the mCRL2 toolset [10].

These current studies are based on an assumption that active learning can learn the correct and complete model. However, due to the number of queries posted to the System Under Learning (SUL) being finite and limited for practical reasons (see Section 2.1), it is typically impossible to guarantee that the learned model is the true representation of the behavior of the legacy component. Therefore, the correctness and completeness of learned results cannot be ensured. To reduce the uncertainty in the learned models, we propose to formally verify the active learning result obtained from learning the software components.

Moreover, 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 publication on inferring interface protocol for software components based on active learning results. To address this, we propose a methodology to infer interface protocols from the active learning results. We verify the active learning results by comparing with the reference models using the model checking toolset, mCRL2 [10]. After this, interface protocol is inferred from the learned model. A formal validation is then performed again by verifying that the original behavioral relations have been preserved during inference of the interface protocol. The formal verification increases the confidence in the learned and inferred models. This work provides the basis for inferring interface protocols of legacy software components, which do not have reference models.

This research takes place 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 consisting of a huge software codebase (comprising over 50 million lines of code). The software architecture is component based, where components interact with each other over interfaces. ASML uses Model Driven Engineering (MDE) techniques, such as Analytical Software Development (ASD) [3]), to build this software but still a considerable amount of software exists that has been developed with traditional practices, such as, manual coding. It will be very time consuming to manually create models for this huge software codebase. ASML, therefore aims for an automated cost-effective transition from existing software

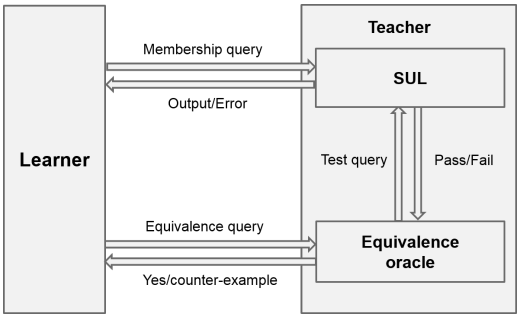


Figure 1: Active Learning framework

to models. In this paper, we propose an approach to support re-factoring or re-engineering of such manually constructed software components with active learning.

2 BACKGROUND

2.1 Active Learning

Angluin [2] presented active learning in 1987 to learn regular languages. The technique is based on the minimal adequate teacher (MAT) framework. Fig. 1 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 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 to the teacher. If the hypothesis is a true representation of the behavior of the SUL, the learning is completed and thus stopped. Otherwise the teacher 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. To find these counterexamples, the teacher generates test queries using some conformance testing method, such as the W-method [5], the Wp-method [9] or the Hopcroft [1] method, to test the hypothesis. The learning cycle continues until the hypothesis is accepted by the teacher.

In theory, using the W-method as equivalence oracle can learn a complete model. This requires the number of states of the SUL to be known. But for legacy systems, the (exact) behavior of SUL is unknown. Therefore, in practice an estimation of the number of states is used. As the W-method requires exponentially more queries when the difference in number of states between a hypothesis and the actual system behavior increases, the estimates are often reduced 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.

We choose *Mealy machines* to represent the results of active learning because Mealy machines are good representations for reactive systems. A Mealy machine is a tuple $\mathcal{M} = \langle S, \Sigma, \Omega, \rightarrow, \hat{s} \rangle$, where

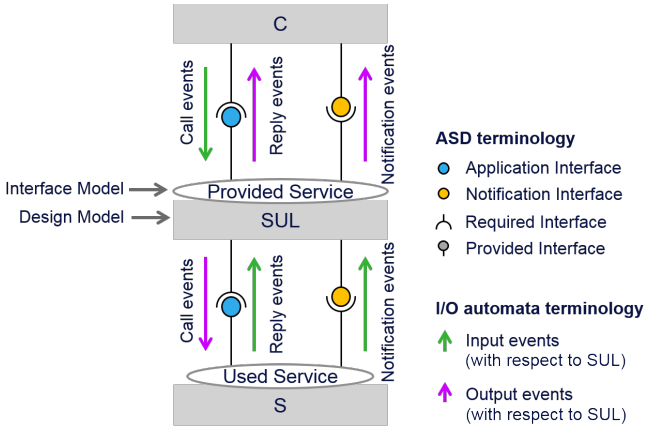


Figure 2: Communication over ASD interfaces

- 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.

2.2 Analytical Software Development (ASD)

ASD:Suite, a tool by the company Verum[1] is based on Verum's patented Analytical Software Development (ASD) [3] 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.

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 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 interfaces: With the help of Fig. 2 we explain the terminology related to ASD, w.r.t. the SUL, that will be used later in this paper. The figure consists of three components, *C*, *SUL* and *S*. The interface model specifies application, notification and modelling 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. Here, *C* is the client for *SUL* and *S* is the server for *SUL*. Notification interfaces exist to provide notification events to clients. Modelling interfaces defines modelling events, which represent spontaneous behavior of an interface model. In this paper modelling interfaces are out of scope.

2.3 mCRL2

mCRL2 is a formal model verification and validation language, with an associated toolset. Specifications written in mCRL2 can

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

be converted to *labelled transition systems*. A labelled 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.

Even in presence of reference models, it is hard to manually compare the models when the number of states and actions increases. We therefore use mCRL2 for behavioral reduction of models and formal comparison between learned and reference models. For this purpose we define the transformation from Mealy machine to LTS.

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.

For every ASD engineered component, its interface model refines its design model modulo failure divergence refinement, denoted by $\sqsubseteq_{FDR}$. Let $\mathcal{L}_i$, where $i \in \{1, 2\}$, be two LTSs. The LTS $\mathcal{L}_1$ refines on LTS $\mathcal{L}_2$ in failure divergence semantics if and only if *divergences*($\mathcal{L}_2$) $\subseteq$ *divergence*($\mathcal{L}_1$) and *failures*($\mathcal{L}_2$) $\subseteq$ *failures*($\mathcal{L}_1$). An action that can happen but cannot be directly observed is called an internal action, denoted by τ. A state from which infinite number of τ actions can be done is called a divergent state and the trace leading to that state is called a divergent trace. The set of divergences for an LTS contains all divergent traces for that LTS. 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 FDR, the reader is referred to [25].

We check weak trace equivalence and weak trace inclusion [10] for verifying the learned result and inferred interface protocol, respectively. We choose these relations because the theory of active learning technique is based on traces. For some cases we may observe stronger relations but at least weak trace equivalence and inclusion must hold between corresponding models.

3 METHODOLOGY

In this section we present our methodology to infer the interface protocol of a software component. In this paper, we use the terms *learned models* for models obtained by learning implementation of software components, and *inferred models* for interface protocols inferred from learned models. The overview of the methodology is shown in Fig. 3 which summarizes the learning and validation steps for learning the software component behavior and inferring its interface protocol. Often the formal verification and learning approaches are considered disjoint. We propose to combine these in our methodology. We choose to learn ASD components as this research takes place at ASML where ASD is used for developing new software. This provides us with a chance to validate our approach on industrial components. For ASD designed components, the reference models are also available for formal comparison with the learned results. However, our approach can be easily applied on legacy software components, as we already applied and tuned our methodology on components with reference models providing us the confidence to apply this methodology on components without reference models.

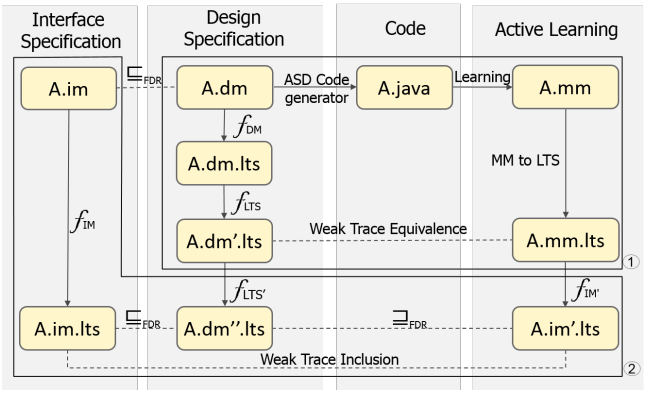


Figure 3: Methodology: From component to interface protocol. The function symbols represent abstraction and transformation functions. Dotted lines represent formal relations. The extensions .im and .dm represent ASD interface and design model, .lts and .mm refer to Labelled Transition System and Mealy machine formalisms, respectively.

We learn ASD software components and use ASD design models as reference models for formal comparison with the learned result (the model learned by active learning algorithm by interacting with the implementation of the ASD software component), and ASD interface models for formal comparison with inferred interface protocols for that particular component. 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 blackbox component.

Our approach comprises two main steps. First, we learn the components with active learning and validate the learning results by formally comparing them with the reference ASD design models of software components, indicated by 1 in Fig. 3. 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 *FDR*, indicated by 2 in Fig. 3. Below we explain each step in detail.

3.1 Learning the components behavior and validating learned results

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 already explained in Section 2, *A.im* refines *A.dm* in failure divergence 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.

- A is input enabled.
- A can be isolated from its environment.
- Outputs from A can be observed.

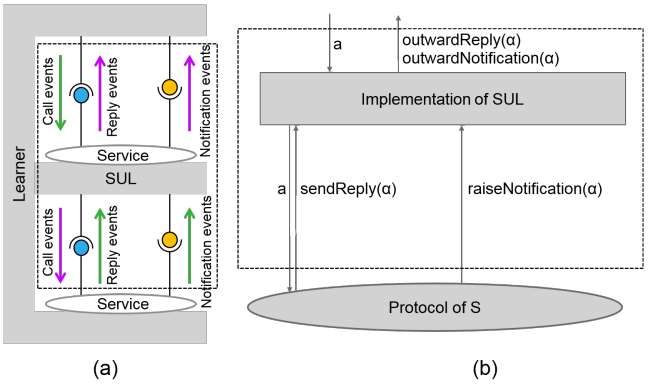


Figure 4: Mapping between (a): active learning scope and (b): translated ASD models from [13].

- A can be reset to initial state after every query.
- A is deterministic.

These assumptions relate to the conditions needed to be fulfilled by the SUL. Non-determinism is out of the scope for this work. The learning algorithm interacts with the component through a mapper. The mapper translates the input symbols from the input alphabet to concrete function calls, sends these to SUL, receives return values as responses and translates these to output symbols from the output alphabet. The output of learning is a Mealy machine.

To validate the learning results, we formally compare the learned results with the ASD design models using the mCRL2 toolset. Based on the translation schemes presented in [13], ASD interface and design models can be translated into mCRL2 models, indicated by the functions f_{IM} and f_{DM}, respectively in Fig. 3. The mapping between active learning results and translations obtained from [13] is shown in Fig. 4. Before we can 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. So from now on 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 the function f_{LTS} on $A.dm.lts$.

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

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

This renaming is done within the mCRL2 toolset. The resulting LTS may contain a relatively large number of τ actions which can be reduced by applying weak trace reduction to increase understandability and readability of the resulting LTS. Now having derived $A.dm'.lts$, we check whether weak trace equivalence holds between the active learning result $A.mm.lts$ and its reference design model $A.dm'.lts$. If the relation holds, it shows the learned model is correct. This comparison is done using *ltscompare* tool from mCRL2.

3.2 Inferring interface protocol

The second step of our methodology concerns inferring the interface protocols of the software components. For an ASD component, the potential interface models for a given design model are those which are failure divergence related to that design model, i.e., $\{im |\ im \sqsubseteq_{FDR} dm\}$, 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 failure divergence 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 is the flower model, i.e. the model which allows every possible action.

To abstract away the details of the interaction with the services provided by other components, we introduce a function $f_{IM'}$ to infer 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:

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

In the same way as for f_{LTS}, mCRL2 is used for renaming. Weak trace reduction can be optionally applied after renaming for reducing τ's to increase readability of the inferred interface protocol. To confirm the inferred interface protocol $A.im'.lts$ is a valid interface protocol, we need to formally check how it relates to $A.dm'.lts$. Before applying this comparison check, we apply $f_{LTS'} = f_{IM'}$ to $A.dm'.lts$ to hide actions from used services. The expected result is $A.im'.lts \sqsubseteq_{FDR} A.dm''.lts$, which validates $A.im'.lts$ as interface protocol for $A.dm''.lts$.

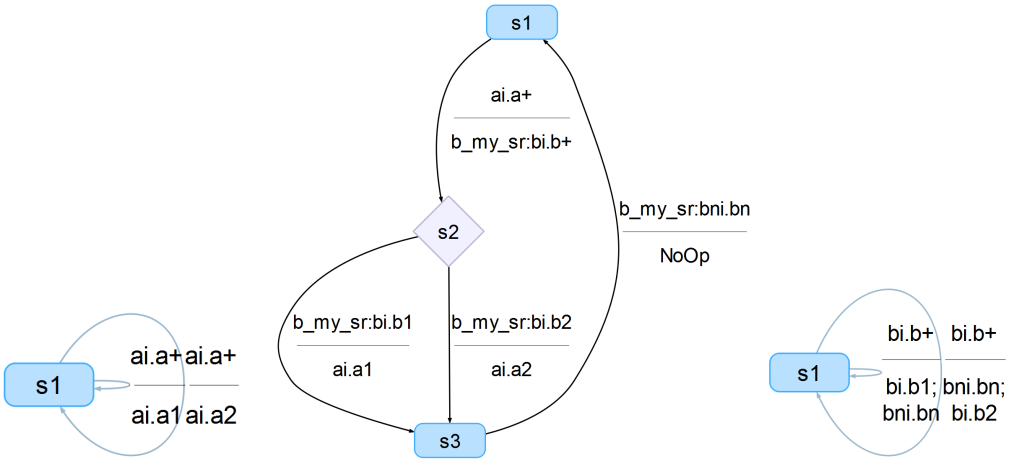
To guarantee that the translation from ASD models to mCRL2 models is correct and does not lose any behavior, we check that $A.im'.lts$ failure divergence refines $A.dm''.lts$, as expected.

The tool chain used to implement our methodology is as follows: Referring Fig. 2, from A.im to A.java ASD is used to automatically generate the code from the design model. LearnLib is used to learn implementation of A.java with active learning algorithm and output is A.mm. All the formal relations are checked with mCRL2 toolset.

In this way, we presented an approach for the formal validation of the results learned from active learning. We have also developed a framework in Java to automate this methodology, which will be used to perform further case studies on a range of industrial components. In the following section, we discuss a small example as a proof of concept for our methodology.

4 EXAMPLE

To demonstrate our approach, we designed an example ASD component A with one application interface. Fig. 5 shows the interface and design models of component A and the interface model of its server



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

Figure 5: ASD models of the SUL and Used service.

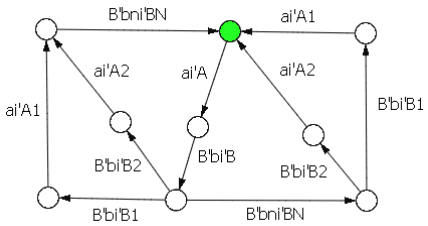


Figure 6: Active learning result of component A, denoted by A.mm.lts in Fig. 3.

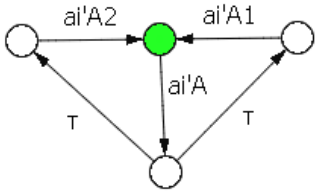


Figure 7: Inferred interface model of component A, denoted by A.im'.lts in Fig. 3.

component S [2]. The *ai.a* and *bi.b* are the call events in application interfaces of A and S respectively, and *bni.bn* is the notification event sent by S to A. We learn the behavior of the component A by applying active learning, using TTT [12] as the learning algorithm and Wp method [9] with depth 4 as the equivalence checking algorithm. To validate the active learning result (Fig. 6), we check the formal relation between the learned result and its ASD design model. We apply $f_{IM'}$ to infer an interface protocol. The transformations are applied as specified in the methodology to obtain $A.mm.lts$ and $A.im'.lts$ as shown in Fig. 6 and Fig. 7, respectively.

The design model (A.dm'.lts) and active learning result (A.mm.lts) are compared and found to be weak trace equivalent, as expected in Fig. 3. The inferred interface protocol ($A.im'.lts$), shown in Fig. 7, refines the learned model, ($A.dm'.lts$), modulo failure divergence refinement. The original interface model, ($A.im.lts$) refines the inferred interface protocol, ($A.im'.lts$), modulo weak trace inclusion.

5 RELATED WORK

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

[2]For more information on ASD concepts and notations, we refer to http://community.verum.com/Files/Starting_ASD_Slides/StartingASDCourse.pdf.

Static Analysis. An overview of static analysis techniques and tools for automated API property inference is given in [19]. An annotation assistant tool to infer pre-conditions and post-conditions of methods of java programs is presented in [8]. Identifying the pre-conditions of functions to infer both data- and control-flow is discussed in [18].

Passive learning. Data mining techniques are combined with passive learning algorithms in [24] to learn data-flow behavior of software components. A passive learning algorithm named GK-tail is presented in [16]. Authors show in [16] 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 in [23].

Active learning. The active learning technique has been used for understanding existing software and analyzing current implementations for detecting flaws. Active learning is used to learn a control software component from Oce printers in [22]. To handle the learning of a bigger system, the input alphabet set is being split to subalphabets and testing is divided to subphases for each subalphabet. A smartcard reader for internet banking has been reverse engineered using active learning [4]. One security flaw was also detected during the learning of this behavior. In [20], the authors combines 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 counter-examples 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.

All of the work stated above deals with learning existing software and the learned results are not formally verified. We propose that formally verifying the learning techniques (active learning in our case), brings greater confidence in the methodology when applied to legacy components. Therefore we formally compared the learned results with the reference models and guaranteed that during learning we do not lose any behavioral information and the formal relations were preserved.

To support the re-factoring and re-engineering of existing software, 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. In this work, we infer interface protocol from active learning results and then formally compare the inferred results with reference models using mCRL2 toolset.

6 CONCLUSIONS & FUTURE WORK

In this paper we have proposed a methodology to gain confidence in the inferred interface protocols through formal comparison between learned models and reference models. 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. The inferred interface protocol, after formal verification, provides a starting point for re-factoring or re-engineering of the software components. To the best of our knowledge, this is the first work to use active learning technique for inferring interface protocols of software components.

In the future, we plan to handle more complex cases including notifications. Currently we are working on automating the process to validate the methodology against a larger set of industrial components. Based on the validation results, we plan to combine existing techniques, such as passive learning and active learning, to infer interface protocols for legacy software. We expect scalability to be a challenge when dealing with industrial components therefore the cost of inferring interfaces needs to be estimated. Data guarded behavior is unavoidable when dealing with real industrial components and therefore will be a concern for future work.

ACKNOWLEDGMENTS

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

The authors would also like to express deep gratitude to Dennis Hendriks, Leonard Lensink, Loek Cleophas, Thomas Neele and Jan Friso Groote, for their valuable feedback regarding the work in general and this paper in particular.

REFERENCES

[1] Marco Almeida, Nelma Moreira, and Rogério Reis. 2009. Testing the equivalence of regular languages. *arXiv preprint arXiv:0907.5058* (2009).
[2] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and computation* 75, 2 (1987), 87–106.
[3] Guy H Broadfoot and Philippa J Broadfoot. 2003. Academia and industry meet: Some experiences of formal methods in practice. In *Software Engineering Conference, 2003. Tenth Asia-Pacific.* IEEE, 49–58.
[4] Georg Chalupar, Stefan Peherstorfer, Erik Poll, and Joeri De Ruiter. 2014. Automated Reverse Engineering using Lego®. *WOOT* 14 (2014), 1–10.
[5] Tsun S. Chow. 1978. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering* 3 (1978), 178–187.
[6] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. Combining model learning and model checking to analyze TCP implementations. In *International Conference on Computer Aided Verification.* Springer, 454–471.
[7] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. 2017. Model learning and model checking of SSH implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software.* ACM, 142–151.
[8] Cormac Flanagan and K Rustan M Leino. 2001. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe.* Springer, 500–517.
[9] Susumu Fujiwara, G v Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. 1991. Test selection based on finite state models. *IEEE Transactions on software engineering* 17, 6 (1991), 591–603.
[10] Jan Friso Groote and Mohammad Reza Mousavi. 2014. *Modeling and analysis of communicating systems.* MIT press.
[11] V. Hamilton. 1994. The use of static analysis tools to support reverse engineering. In *IEE Colloquium on Reverse Engineering for Software Based Systems.* 6/1–6/4.
[12] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning.. In *RV.* 307–322.
[13] Ruben J. W. Jonk. 2016. *The semantic of ALIAS defined in mCRL2.* Master's thesis.
[14] Maikel Leemans, Wil MP van der Aalst, and Mark GJ van den Brand. 2018. 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).* IEEE, 502–506.
[15] Manny M Lehman. 1996. Laws of software evolution revisited. In *European Workshop on Software Process Technology.* Springer, 108–124.
[16] D. Lorenzoli, L. Mariani, and M. PezzÃĺ. 2008. Automatic generation of software behavioral models. In *2008 ACM/IEEE 30th International Conference on Software Engineering.* 501–510. https://doi.org/10.1145/1368088.1368157
[17] Tiziana Margaria, Oliver Niese, Harald Raffelt, and Bernhard Steffen. 2004. Efficient test-based model generation for legacy reactive systems. In *High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International.* IEEE, 95–100.
[18] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static specification inference using predicate mining. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 123–134.
[19] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API property inference techniques. *IEEE Transactions on Software Engineering* 39, 5 (2013), 613–637.
[20] Mathijs Schuts, Jozef Hooman, and Frits Vaandrager. 2016. Refactoring of legacy software using model learning and equivalence checking: an industrial experience report. In *International Conference on Integrated Formal Methods.* Springer, 311–325.
[21] Anas Shatnawi, Abdelhak-Djamel Seriai, Houari Sahraoui, and Zakarea Alshara. 2017. Reverse engineering reusable software components from object-oriented APIs. *Journal of Systems and Software* 131 (2017), 442–460.
[22] Wouter Smeenk, Joshua Moerman, Frits Vaandrager, and David N Jansen. 2015. Applying automata learning to embedded control software. In *International Conference on Formal Engineering Methods.* Springer, 67–83.
[23] Neil Walkinshaw and Kirill Bogdanov. 2008. Inferring finite-state models with temporal constraints. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering.* IEEE Computer Society, 248–257.
[24] Neil Walkinshaw, Ramsay Taylor, and John Derrick. 2016. Inferring extended finite state machine models from software executions. *Empirical Software Engineering* 21, 3 (2016), 811–853.
[25] Ting Wang, Songzheng Song, Jun Sun, Yang Liu, Jin Song Dong, Xinyu Wang, and Shanping Li. 2012. More anti-chain based refinement checking. In *International Conference on Formal Engineering Methods.* Springer, 364–380.