Refactoring of Legacy Software using Model Learning and Equivalence Checking: an Industrial Experience Report

Mathijs Schuts[1], Jozef Hooman[2,3] and Frits Vaandrager[3]

[1] Philips, Best, The Netherlands
`mathijs.schuts@philips.com`
[2] Embedded Systems Innovation (ESI) by TNO, Eindhoven, The Netherlands
`jozef.hooman@tno.nl`
[3] Department of Software Science, Radboud University, Nijmegen, The Netherlands
`f.vaandrager@cs.ru.nl`

Abstract. Many companies struggle with large amounts of legacy software that is difficult to maintain and to extend. Refactoring legacy code typically requires large efforts and introduces serious risks because often crucial business assets are hidden in legacy components. We investigate the support of formal techniques for the rejuvenation of legacy embedded software, concentrating on control components. Model learning and equivalence checking are used to improve a new implementation of a legacy control component. Model learning is applied to both the old and the new implementation. The resulting models are compared using an equivalence check of a model checker. We report about our experiences with this approach at Philips. By gradually increasing the set of input stimuli, we obtained implementations of a power control service for which the learned behaviour is equivalent.

1 Introduction

The high-tech industry creates complex cyber physical systems. The architectures for these systems evolved over many decades through a constant stream of product innovations. This usually leads to so-called *legacy* components that are hard to maintain and to extend [24,25]. Typically, these components are based on obsolete technologies, frameworks, and tools. Documentation might not be available or outdated and the original developers are often no longer available. In addition, the existing regression test set for validating the component will be very limited in most cases.

Given these characteristics, innovations that require changes of legacy components are risky. Many legacy components implicitly incorporate important business knowledge, hence failures will lead to substantial losses. To avoid a risky greenfield approach, starting from scratch, several techniques are being developed to extract the crucial business information hidden in legacy components in a (semi-)automated way and to use this information to develop a refactored version of the component.

There are several approaches to extract this hidden information. Static analysis methods concentrate on the analysis and transformation of source code. For instance, the commercial Design Maintenance System (DMS)[4] has been used in several industrial projects to re-engineer code. DMS is based on abstract syntax tree (AST) representations of programs [3].

Whereas static analysis techniques focus on the internal structure of components, learning techniques aim at capturing the externally visible behaviour of a component. Process mining extracts business logic based on event logs [23]. In [17], a combination of static analysis and process mining has been applied to a financial management system, identifying tasks, actors, and their roles. Process mining can be seen as a passive way of learning which requires an instrumentation of the code to obtain event logs.

Active learning techniques [4,22] do not require code instrumentation, but need an adapter to interact with a running system. In this approach, a learning algorithm interacts with a software component by sending inputs and observing the resulting output, and uses this information to construct a state machine model. Active learning has, for instance, been successfully applied to learn models of (and to find mistakes in) implementations of protocols such as TCP [12] and TLS [8], to establish correctness of protocol implementations relative to a given reference implementation [2], and to generate models of a telephone switch [18] and a printer controller [21]. Learning-based testing [11] combines active learning and model checking. In this approach, which requires the presence of a formal specification of the system, model checking is used to guide the learning process. In [11] three industrial applications of learning-based testing are described from the web, automotive and finance domains.

In this paper, we report about a novel industrial application of active learning to gain confidence in a refactored legacy component using formal techniques. In the absence of any formal specification of the legacy system, the use of model checking and learning-based testing was not possible. Instead we decided to use a different combination of tools, similar to the approach of [13,2]. The model learning tool LearnLib [15] was used to learn Mealy machine models of the legacy and the refactored implementation. These models were then compared to check if the two implementations are equivalent. Since the manual comparison of large models is not feasible, we used an equivalence checker from the mCRL2 toolset [7] for this task. In brief, our approach can be described as follows (see also Figure 1):

1. Implementation A (the legacy component) is explored by a model learner. The output of the model learner is converted to an input format for the equivalence checker, model MA.
2. Implementation B (the refactored component) is explored by a model learner. The output of the model learner is converted to an input format for the equivalence checker, model MB.
3. The two models are checked by the equivalence checker. The result of the equivalence checker can be:

[4] www.semanticdesigns.com

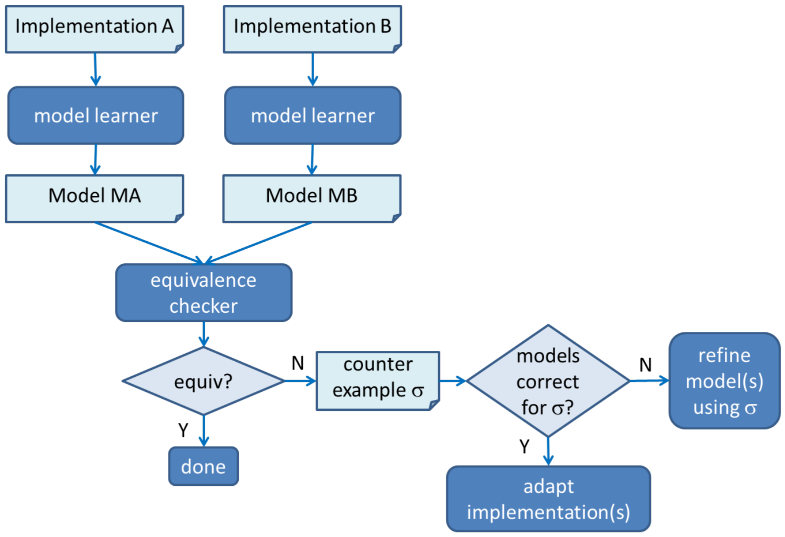


Fig. 1. Approach to compare legacy component and refactored version

- The two models are equivalent. In this case we are done.
- The two models are not equivalent and a counterexample is provided: a sequence of inputs σ for which the outputs produced by the two models are different. In this case we proceed to step 4.

4. Because models A and B have been obtained through a finite number of tests, we can never be sure that they correctly describe implementations A and B, respectively. Therefore, if we find a counterexample σ for the equivalence of models MA and MB, we first check whether implementation A and model MA behave the same for σ, and whether implementation B and model MB behave the same for σ. If there is a discrepancy between a model and the corresponding implementation, this means that the model is incorrect and we ask the model learner to construct a new model based on counterexample σ, that is, we go back to step 1 or 2. Otherwise, counterexample σ exhibits a difference between the two implementations. In this case we need to change at least one of the implementations, depending on which output triggered in response to input σ is considered unsatisfactory behaviour. Note that also the legacy component A might be changed, because the counterexample might indicate an unsatisfactory behaviour of A. After the change, a corrected implementation needs to be learned again, i.e., we go back to step 1 or 2.

Since the learning of an implementation can take a substantial amount of time, we start with a limited subset of input stimuli for the model-learner and increase the number of stimuli once the implementations are equivalent for a smaller number of stimuli. Hence, the approach needs to be executed iteratively.

We report about our experiences with the described approach on a real development project at Philips. The project concerns the introduction of a new hardware component, the Power Control Component (PCC). A PCC is used to start-up and shutdown an interventional radiology system. All computers in the system have a software component, the Power Control Service (PCS) which communicates with the PCC over an internal control network during the execution of start-up and shutdown scenarios. To deal with the new hardware of the PCC, which has a different interface, a new implementation of the PCS is needed. Since different configurations have to be supported, with old and new PCC hardware, the old and new PCS software should have exactly the same externally visible behaviour.

The PCS is described in Sect. 2 to the extend needed for understanding this paper. Section 3 describes the use of model-learning and model-checking to compare the two PCS implementations for the old and the new PCC. The results of testing the two PCS implementations are described in Sect. 4. Section 5 discusses the scalability of our approach. Concluding remarks can be found in Sect. 6.

2 The Industrial Development Project

2.1 Power Control Service

For starting up and shutting down an interventional radiology system multiple components are involved. The Power Control Component (PCC) is a hardware component that gets the mains power input from the hospital. It conditions the mains power, switches the power taps that are connected to system's internal components and acts as the master of the system when executing start-up and shutdown scenarios. All computers in the system are powered by the PCC and are controlled by the PCC via a Power Control Service (PCS) that connects to the PCC via the system's internal control network.

Figure 2 depicts the PCS in its context. The PCS is a software component that is used to start and stop subsystems via their Session Managers (SMs). In addition to the start-up and shutdown scenarios executed by the PCC, the PCS is also involved during service scenarios such as upgrading the subsystem's software.

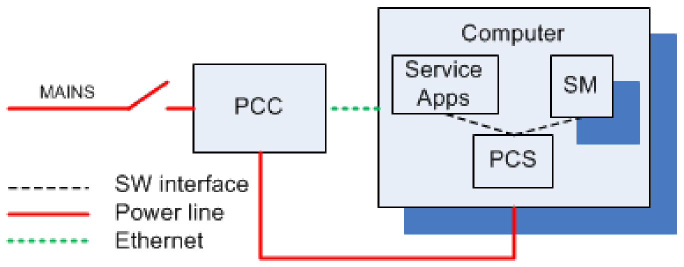


Fig. 2. Context Power Control Service

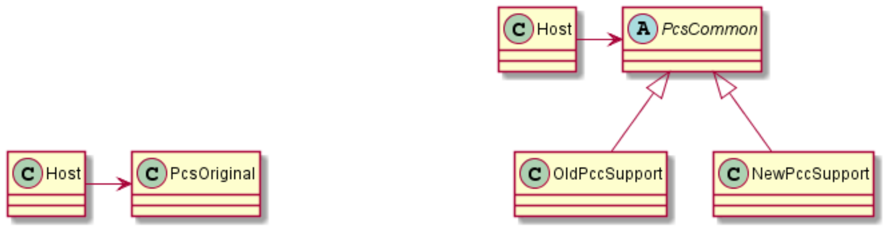
In a typical shutdown scenario, the user presses the off button and the shutdown scenario is initiated by the PCC. The PCC sends an event to all PCSs. The PCS stops the SMs. Once the SMs are stopped, the PCS triggers the Operating System (OS) to shutdown. In the end, the OS will stop the PCS.

Another scenario is to switch from closed profile to open profile when the system is in the operational state. In closed profile only the clinical application can be executed by the user of the system. Open profile is used during development for testing purposes. In this scenario, the service application triggers the PCS to switch to open profile. The PCS will then stop the SMs. When the PCS is ready, the service application reboots the PC. After the reboot, the OS starts up the PCS and the PCS starts a subset of the SMs based on the SM's capabilities. In open profile, the service application can also start the clinical application by providing the PCS with the OpenProfileStartApplication trigger.

2.2 Refactoring

The PCS implementation for the old PCC is event-based. An event is handled differently based on the value of global flags in the source code. Hence, all state behaviour is implicitly coded by these flags, which makes the implementation unmaintainable. The development of a new implementation for supporting the new PCC is an opportunity to create a maintainable implementation. The new implementation makes the state behaviour explicit by a manually crafted state machine.

To be able to support both the old and the new PCC, the PCS software has been refactored such that the common behaviour for both PCCs is extracted. Figure 3(a) depicts the PCS before refactoring. The Host implements the IHost interface that is used by the service application. The implementation of the PCS after refactoring is show in Fig. 3(b).



(a) Before refactoring (b) After refactoring

Fig. 3. Class Diagrams of PCS Design

The PcsCommon class implements the ISessionManager interface to control the SMs. The OldPccSupport class contains the legacy implementation for the old PCC whereas a NewPccSupport class deals with the new PCC. Both classes inherit from the PcsCommon class to achieve the same internal interface for the Host. Depending on the configuration, the Host creates an instance of either the OldPccSupport or the NewPccSupport class.

The PCS as depicted in Fig. 3(b) is written in C++ and consists of a total of 3365 Lines Of Code (LOC): Host has 741 LOC, PcsCommon has 376 LOC, OldPccSupport has 911 LOC, and NewPccSupport has 1337 LOC.

The unit test cases were adapted to include tests for the new implementation. It was known that the unit test set is far from complete. Hence, we investigated the possibility to use model-learning to get more confidence in the equivalence of the old and new implementations.

3 Application of the Learning Approach

To learn models of our implementations, we used the LearnLib tool [19], see `http://learnlib.de/`. For a detailed introduction into LearnLib we refer to [22]. In our application we used the development 1.0-SNAPSHOT of LearnLib and its MealyLearner which is connected to the System Under Learning (SUL) by means of an adapter and a TCP/IP connection.

3.1 Design of the learning environment

Figure 4 depicts the design used for learning the PCS component. Creating an initial version of the adapter took about 8 hours, because the test primitives of the existing unit test environment could be re-used.

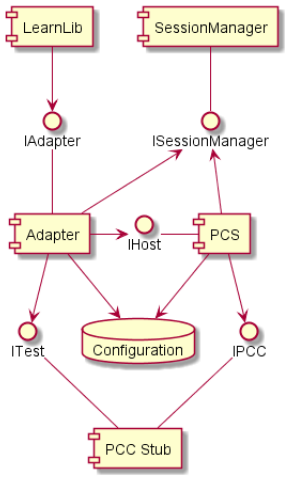


Fig. 4. Design learning environment

With this design, the PCS can be learned for both the old and the new PCC. The adapter automatically changes the configuration of the PCS such that the PCS knows if it needs to instantiate the old or the new implementation. Depending on the old or new PCC, the adapter instantiates a different PCC stub.

3.2 Learned output

The Mealy machine that is the result of a LearnLib session is represented as a ”dot” file, which can be visualized using Graphviz[5]. A fragment of a model is shown in Table 1.

```
digraph g {
start0 [label=”” shape=”none”];

s0 [shape=”circle” label=”0”];
s1 [shape=”circle” label=”1”];
s2 [shape=”circle” label=”2”];
s3 [shape=”circle” label=”3”];
s4 [shape=”circle” label=”4”];
s5 [shape=”circle” label=”5”];
s6 [shape=”circle” label=”6”];
s7 [shape=”circle” label=”7”];
s8 [shape=”circle” label=”8”];
s0 -> s1 [label=”|PCC(StateSystemOn)| / |PCS(Running);SM1(Running);SM2(Running)|”];
s0 -> s2 [label=”|PCC(StateSystemOff)| / |PCS(Running);SM1(Stopped);SM2(Stopped);Dev(Shutdown)|”];
s1 -> s2 [label=”|PCC(ButtonSystemOff)| / |PCS(Running);SM1(Stopped);SM2(Stopped);Dev(Shutdown)|”];
s1 -> s3 [label=”|Host(goToOpenProfile)| / |PCS(Stopped);SM1(Stopped);SM2(Stopped);Dev(OpenProfile)|”];
...
start0 -> s0;
}
```

Table 1. Fragment of a learned dot-file

3.3 Checking Equivalence

For models with more than five states it is difficult to compare the graphical output of LearnLib for different implementations. Therefore, an equivalence checker is used to perform the comparison. In our case, we used the tool support for mCRL2 (micro Common Representation Language 2) which is a specification language that can be used for specifying system behaviour. The mCRL2 language comes with a rich set of supporting programs for analysing the behaviour of a modelled system [7].

Once the implementation is learned, a small script is used to convert the output from LearnLib to a mCRL2 model. Basically, the learned Mealy machine is represented as an mCRL2 process `Spec(s:States)`. As an example, the two transitions of state $s0$ in the dot-file

[5] www.graphviz.org/

```
s0 -> s1 [label="|PCC(StateSystemOn)| / |PCS(Running);SM1(Running);SM2(Running)|"];
s0 -> s2 [label="|PCC(StateSystemOff)| / |PCS(Running);SM1(Stopped);SM2(Stopped);Dev(Shutdown)|"];
```

are translated into the following process algebra construction:

```
(s==s0) -> (
 (PCC(StateSystemOn) . PCS(Running) . SM1(Running) . SM2(Running) . Spec(s1)) +
 (PCC(StateSystemOff) . PCS(Running) . SM1(Stopped) . SM2(Stopped) . Dev(Shutdown) . Spec(s2))
)
```

A part of the result of translating the model of Table 1 to mCRL2 is shown in Table 2.

```
sort States = struct s0 | s1 | s2 | s3 | s4 | s5 | s6 | s7 | s8;
OsStim = struct StartPcs | StopPcs;
PCCStim = struct StateSystemOn | StateSystemOff | ...;
HostStim = struct stopForInstallation | startAfterInstallation | ...;
ServiceStates = struct Running | Stopped;
DevStates = struct OpenProfile | Shutdown;

act OS:OsStim;
act PCC:PCCStim;
act Host:HostStim;
act PCS:ServiceStates;
act SM1:ServiceStates;
act SM2:ServiceStates;
act Dev:DevStates;

proc Spec(s:States)=
 (s==s0) -> (
  (PCC(StateSystemOn) . PCS(Running) . SM1(Running) . SM2(Running) . Spec(s1)) +
  (PCC(StateSystemOff) . PCS(Running) . SM1(Stopped) . SM2(Stopped) . Dev(Shutdown) . Spec(s2))
 ) +
 (s==s1) -> (
  (PCC(ButtonSystemOff) . PCS(Running) . SM1(Stopped) . SM2(Stopped) . Dev(Shutdown) . Spec(s2)) +
  (Host(goToOpenProfile) . PCS(Stopped) . SM1(Stopped) . SM2(Stopped) . Dev(OpenProfile) . Spec(s3)) +
  (Host(goToClosedProfile) . PCS(Stopped) . SM1(Stopped) . SM2(Stopped) . Spec(s4)) +
  (Host(openProfileStartApplication) . PCS(Running) . SM1(Running) . SM2(Running) . Spec(s1)) +
  (Host(openProfileStopApplication) . PCS(Running) . SM1(Running) . SM2(Running) . Spec(s1)) +
  (OS(StartPcs) . PCS(Running) . SM1(Running) . SM2(Running) . Spec(s1)) +
  (OS(StopPcs) . PCS(Stopped) . SM1(Stopped) . SM2(Stopped) . Spec(s4))
 ) +
 (s==s2) -> (
 ...
 );

init Spec(s0);
```

Table 2. Fragment of mCRL2 model

Given two (deterministic) Mealy machines, the labelled transition systems for the associated mCRL2 processes are also deterministic. Since the labelled transition systems also do not contain any τ-transitions, trace equivalence and bisimulation equivalence coincide, and there is no difference between weak and strong equivalences [10]. Thus, two Mealy machines are equivalent iff the associated mCRL2 processes are (strong) trace equivalent, and the mCRL2 processes are (strong) trace equivalent iff they are (strong) bisimulation equivalent.

3.4 Investigating Counterexamples

When the equivalence check indicates that the two models are not equivalent, the mCRL2 tool provides a counterexample. To investigate counterexamples, we created a program that reads a produced counterexample and executes this on the implementations. In the design depicted in Fig. 4, the LearnLib component has been replaced by the counterexample program. As before, switching between the two implementations can be done by instructing the adapter. In this way, the standard logging facilities of program execution are exploited to study the counterexample.

4 Results of Learning the Implementations of the PCS

In this section we describe the results of applying the approach of Sect. 1 to the implementations of the PCS component.

4.1 Iteration 1

The first iteration was used to realize the learning environment as is described in Sect. 3.1. An adapter was created to interface between the PCS and LearnLib. Because the communication between the PCS and the adapter is asynchronous, the adapter has to wait some time before the state of the PCS can be examined. In this iteration we performed a few try runs to tweak the wait time needed before taking a sample. In addition, the first iteration was used to get an impression on how long learning the PCS takes with different numbers of stimuli. The necessary waiting time of 10 second after a stimulus for learning the PCS is quite long, and this greatly influenced the time needed for learning models.

4.2 Iteration 2

After a first analysis of the time needed for model learning in iteration 1, we decided to start learning with 9 stimuli. These 9 stimuli were all related to basic start-up/shutdown and service scenarios. We learned the PCS implementation for the old PCC and the PCS implementation for the new PCC. The results are presented in Table 3. The table has a column for the number of stimuli, for the number of states and transitions found, and for the time it took for LearnLib to learn the implementations.

	Stimuli	States	Transitions	Time (in seconds)
PCS implementation for old PCC	9	8	43	32531
PCS implementation for new PCC	9	3	8	1231

Table 3. Results learning PCS with 9 stimuli

Note that learning a model for the old implementation took 9 hours. (This excludes the time used to test the correctness of the final model.) As described in Sect. 3.3, the learned models were converted to mCRL2 processes. Next, the mCRL2 tools found a counterexample starting with:
PCC(StateSystemOn), PCS(Running), SM1(Running), SM2(Running), ...

We investigated this counterexample and found an issue in the PCS implementation for the new PCC. The new implementation did not make a distinction between the SystemOff event, and the ServiceStop and ServiceShutdown events.

Note that before performing the learning experiment the new and old implementations were checked using the existing regression test cases. This issue was not found by the existing unit test cases.

4.3 Iteration 3

In the third iteration, the PCS implementation for the new PCC was re-learned after solving the fix. Table 4 describes the results.

	Stimuli	States	Transitions	Time (in seconds)
PCS implementation for old PCC	9	8	43	32531
PCS implementation for new PCC	9	7	36	8187

Table 4. Results learning PCS with 9 stimuli

An equivalence check with the mCRL2 tools resulted in a new counterexample of 23 commands:
PCC(StateSystemOn), PCS(Running), SM1(Running), SM2(Running),
Host(goToOpenProfile), PCS(Stopped), SM1(Stopped), SM2(Stopped),
Dev(OpenProfile), OS(StartPcs), PCS(Running), SM1(Stopped),
SM2(Running), Dev(OpenProfile), Host(openProfileStopApplication),
PCS(Running), SM1(Stopped), SM2(Running), Dev(OpenProfile),
PCC(ButtonSystemOff), PCS(Running), SM1(Stopped), SM2(Running).

When we executed this counterexample on the PCS implementation for the old PCC, we found the following statement in the logging of the PCS: "Off button not handled because of PCS state (Stopping)". A quick search in the source code revealed that the stopSessionManagers method prints this statement when the Stopping flag is active. This is clearly wrong, because this flag is set by the previous stimulus, i.e., the openProfileStopApplication stimulus. The PCS implementation for the old PCC was adapted to reset the Stopping flag after handling the openProfileStopApplication stimulus.

4.4 Iteration 4

In the fourth iteration, the PCS implementation for the old PCC was re-learned after solving the fix. Table 5 describes the results after re-learning. Note that, after correcting the error, learning the model for the old implementation only takes slightly more than one hour. When checking the equivalence, the mCRL2 tool reports that the two implementation are (strong) trace equivalent for these 9 stimuli.

	Stimuli	States	Transitions	Time (in seconds)
PCS implementation for old PCC	9	7	36	4141
PCS implementation for new PCC	9	7	36	8187

Table 5. Results learning PCS with 9 stimuli

4.5 Iteration 5

As a next step we re-learned the implementations for the complete set of 12 stimuli; the results are shown in Table 6. Note that learning the new implementation takes approximately 3.5 hours. The mCRL2 tools report that the two obtained models with 12 stimuli are trace equivalence and bisimulation equivalent.

	Stimuli	States	Transitions	Time (in seconds)
PCS implementation for old PCC	12	9	65	10059
PCS implementation for new PCC	12	9	65	12615

Table 6. Results learning PCS with 12 stimuli

5 Scalability of the Learning Approach

Using model learning we found issues in both a legacy software component and in a refactored implementation. After fixing these issues, model learning helped to increase confidence that the old and the new implementations behave the same. Although this is a genuine industrial case study, the learned Mealy machine models are very small. Nevertheless, learning these tiny models already took up to 9 hours. For applying these techniques in industry there is an obvious need to make model learning more efficient in terms of the time needed to explore a system under learning. Clearly, our approach has been highly effective for the PCC case study. But will it scale?

Below we present an overview of some recent results that make us optimistic that indeed our approach can be scaled to a large class of more complex legacy systems.

5.1 Faster implementations

The main reason why model learning takes so long for the PCC case study is the long waiting time in between input events. As a result, running a single test sequence (a.k.a. membership query) took on average about 10 seconds. One of the authors was involved in another industrial case study in which a model for a printer controller was learned with 3410 states and 77 stimuli [21]. Even though more than 60 million test sequences were needed to learn it, the task could be completed within 9 hours because on average running a single test sequence took only 0.0005 seconds. For most software components the waiting times can be much smaller than for the PCS component studied in this paper. In addition, if the waiting times are too long then sometimes it may be possible to modify the components (just for the purpose of the model learning) and reduce the response times. For our PCC case study such an approach is difficult. The PCS controls the Session Managers (SMs), which are Windows services. After an input event we want to observe the resulting state change of the SMs, but due to the unreliable timing of the OS we need to wait quite long. In order to reduce waiting times we would need to speed up Windows.

5.2 Faster learning and testing algorithms

There has been much progress recently in developing new algorithms for automata learning. In particular, the new TTT learning algorithm that has been introduced by Isberner [16] is much faster than the variant of Angluin's L^* algorithm [4] that we used in our experiments. Since the models for the PCS components are so simple, the L^* algorithm does not need any intermediate hypothesis: the first model that L^* learns is always correct (that is, extensive testing did not reveal any counterexample). The TTT algorithm typically generates many more intermediate hypotheses than L^*. This means that it becomes more important which testing algorithm is being used. But also in the area of conformance testing there has been much progress recently [9,21]. Figure 5 displays the results of some experiments that we did using an implementation of the TTT algorithm that has become available very recently in LearnLib, in combination with a range of testing algorithms from [9,21]. As one can see, irrespective of the test method that is used, the TTT algorithm reduces the total number of input events needed to learn the final PCS model with a factor of about 3.

5.3 Using parallelization and checkpointing

Learning and testing can be easily parallelized by running multiple instances of the system under learning (in our case the PCS implementation) at the same time. Henrix [14] reports on experiments in which doubling the number of parallel instances nearly doubles the execution speed (on average with a factor 1.83). Another technique that may speed-up learning is to save and restore software states of the system under learning (checkpointing). The benefit is that if the learner wants to explore different outgoing transitions from a saved state q it only

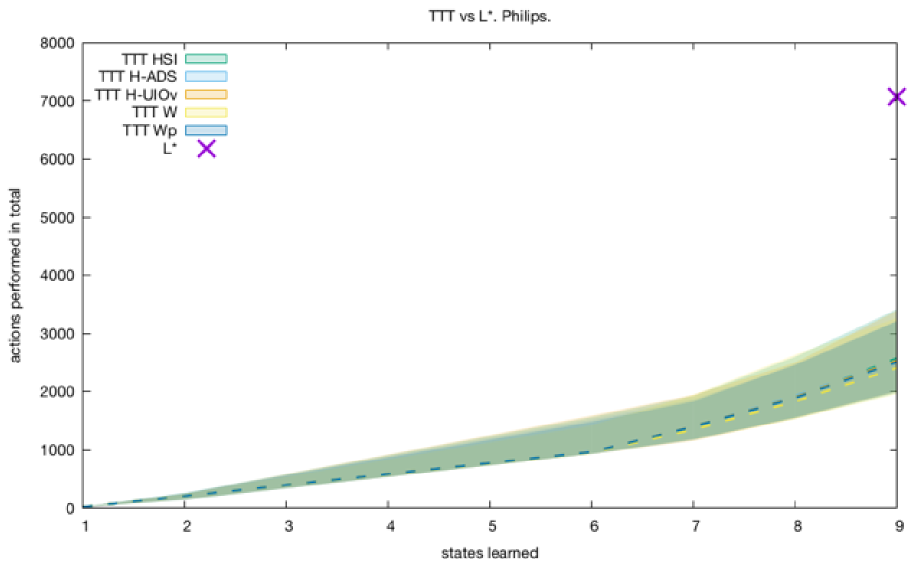


Fig. 5. Experiments with TTT algorithm for final PCS implementation for new PCC. The used test methods (W, Wp, hybrid adaptive distinguishing sequences, hybrid UIOv) were all randomised. For each test method 100 runs were performed. In each case 95% of the runs were in the shaded area. The dotted lines give the median run for a given test method.

needs to restore q, which usually is much faster than resetting the system and bringing it back to q by an appropriate sequence of inputs. Henrix [14] reports on experiments in which checkpointing with DMTCP [5] speeds up the learning process with a factor of about 1.7.

5.4 Using abstraction and restriction

The number of test/membership queries of most learning algorithms grows linearly with the number of inputs. However, these algorithms usually assume an oracle that provides counterexamples for incorrect hypothesis models. Such an oracle is typically implemented using a conformance testing algorithm. In practice, conformance testing often becomes a bottleneck when the number of inputs gets larger. Thus we seek methods that help us to reduce the number of inputs.

To get confidence that two implementations with a large number of stimuli exhibit the same behaviour, a simple but practical approach is to apply model learning for multiple smaller subsets of stimuli. This will significantly reduce the learning complexity, also because the set of reachable states will typically be smaller for a restricted number of stimuli. Models learned for a subset of the inputs may then be used to generate counterexamples while learning models for larger subsets on inputs. Smeenk [20] reports on some successful experiments in which this heuristic was used.

A different approach, which has been applied successfully in many case studies, is to apply abstraction techniques that replace multiple concrete inputs by a single abstract input. One may, for instance, forget certain parameters of an input event, or only record the sign of an integer parameter. We refer to [1,6] for recent overviews of these techniques.

6 Concluding Remarks

We presented an approach to get confidence that a refactored software component has equivalent external control behaviour as its non-refactored legacy software implementation. From both the refactored implementation and its legacy implementation, a model is obtained by using model learning. Both learned models are then compared using an equivalence checker. The implementations are learned and checked iteratively with increasing sets of stimuli to handle scalability. By using this approach we found issues in both the refactored and the legacy implementation in an early stage of the development, before the component was integrated. In this way, we avoided costly rework in a later phase of the development. As future work, we intend to apply our approach to other software components that will be refactored, including a substantially larger component.

Acknowledgements We are most grateful to Joshua Moerman for helping with the experiments with the TTT algorithm. We also thank Petra van den Bos for careful proofreading of an earlier version. This research was supported by STW project 11763 (ITALIA) and the Dutch national program COMMIT.

References

1. F. Aarts, B. Jonsson, J. Uijen, and F.W. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, 46(1):1–41, 2015.
2. F. Aarts, H. Kuppens, G.J. Tretmans, F.W. Vaandrager, and S. Verwer. Improving active Mealy machine learning for protocol conformance testing. *Machine Learning*, 96(1–2):189–224, 2014.
3. R. L. Akers, I. D. Baxter, M. Mehlich, B. J. Ellis, and K. R. Luecke. Case study: Re-engineering C++ component models via automatic program transformation. *Information and Software Technology*, 49(3):275 – 291, 2007.
4. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2), 1987.
5. J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *IEEE Parallel and Distributed Processing Symposium*, 2009.
6. S. Cassel. *Learning Component Behavior from Tests: Theory and Algorithms for Automata with Data.* PhD thesis, University of Uppsala, 2015.

7. S. Cranen, J.F. Groote, J.J.A. Keiren, F.P.M. Stappers, E.P. de Vink, W. Wesselink, and T.A.C. Willemse. An Overview of the mCRL2 Toolset and Its Recent Advances. In N. Piterman and S. A. Smolka, editors, *TACAS 2013*, LNCS, vol. 7795, pages 199–213. Springer, Heidelberg, 2013.
8. J. de Ruiter and E. Poll. Protocol state fuzzing of tls implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206. USENIX Association, 2015.
9. R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, and N. Yevtushenko. FSM-based conformance testing methods: A survey annotated with experimental evaluation. *Information & Software Technology*, 52(12):1286–1297, 2010.
10. J. Engelfriet. Determinacy - (observation equivalence = trace equivalence). *Theoretical Computer Science*, 36:21–25, 1985.
11. L. Feng, S. Lundmark, K. Meinke, F. Niu, M.A. Sindhu, and P.Y.H. Wong. Case studies in learning-based testing. In H. Yenigün, C. Yilmaz, and A. Ulrich, editors, *ICTSS 2013*, LNCS, vol. 8254, pages 164–179. Springer, Heidelberg, 2013.
12. P. Fiterău-Broştean, R. Janssen, and F.W. Vaandrager. Learning fragments of the TCP network protocol. In F. Lang and F. Flammini, editors, *FMICS 2014*, LNCS, vol. 8718, pages 78–93. Springer, Heidelberg, 2014.
13. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. *Logic Journal of IGPL*, 14(5):729–744, 2006.
14. M. Henrix. *Performance improvement in automata learning.* Master thesis, Radboud University, Nijmegen, 2015.
15. F. Howar, M. Isberner, M. Merten, and B. Steffen. Learnlib tutorial: From finite automata to register interface programs. In T. Margaria, editor, *ISoLA 2012*, LNCS, vol. 7609, pages 587–590. Springer, Heidelberg, 2012.
16. M. Isberner. *Foundations of Active Automata Learning: An Algorithmic Perspective.* PhD thesis, Technical University of Dortmund, 2015.
17. A.C. Kalsing, G.S. do Nascimento, C. Iochpe, and L.H. Thom. An incremental process mining approach to extract knowledge from legacy systems. In *Enterprise Distributed Object Computing Conference (EDOC)*, pages 79–88, 2010.
18. T. Margaria, O. Niese, H. Raffelt, and B. Steffen. Efficient test-based model generation for legacy reactive systems. In *9th IEEE Int. High-Level Design Validation and Test Workshop*, pages 95–100, 2004.
19. H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009.
20. W. Smeenk. *Applying Automata Learning to Complex Industrial Software.* Master thesis, Radboud University, Nijmegen, September 2012.
21. W. Smeenk, J. Moerman, F. W. Vaandrager, and D. N. Jansen. Applying automata learning to embedded control software. In M. Butler, S. Conchon, and F. Zaïdi, editors, *ICFEM 2015*, LNCS, vol. 9407, pages 67–83. Springer, Heidelberg, 2015.
22. B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In Marco Bernardo and Valrie Issarny, editors, *SFM 2011*, LNCS, vol. 6659, pages 256–296. Springer, Heidelberg, 2011.
23. W. van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes.* Springer-Verlag Berlin Heidelberg, 2011.
24. C. Wagner. *Model-Driven Software Migration: A Methodology.* Springer Vieweg, 2014.
25. I. Warren. *The Renaissance of Legacy Systems - Method Support for Software-System Evolution.* Springer London, 1999.