August 27, 2018

Refactoring

A Practical Application of Model Learning and Code Generation

Author:
ing. Gijs van der Meijde

Supervisors:
prof.dr. Frits Vaandrager
dr.ir. Ammar Osaiweran

Second reader
dr.ir. G.J. Tretmans

Radboud Universiteit
Comeniuslaan 4
NL-6525 HP Nijmegen
The Netherlands
`http://www.ru.nl`

Refactoring
A Practical Application of Model Learning and Code Generation

Name	ing. Gijs van der Meijde
Student number	s4645251
Discipline	Master in Computing Science Specialization: Software Science
Host organization	ASML
Start date	12-02-2018
Presentation date	31-08-2018
University coach	prof.dr. Frits. Vaandrager
Host organization coach	dr.ir. Ammar Osaiweran
Second reader	dr.ir. G.J. Tretmans
Date of this document	August 27, 2018

Document created using the KingTex Template (https://www.kingtech.nl/kingtex)

Abstract

Refactoring (or re-engineering) software is not without risks. Changing legacy software poses a risk of introducing bugs into the system. On top of that, it is time-intensive work with high costs. In order to mitigate the struggle of software engineers, we propose a method that uses a combination of model-learning, test- and code generation. Our method complements common software refactoring processes applied in the industry. It aims at reducing the risk of refactoring software and assists in improving software maintainability using proven guidelines.
In this thesis, we show how model-learning can be applied to extract a behavioral model from legacy components and how this model is refined to reflect the desired behavior of the software. This process allows us to gain a better understanding of the legacy code. From the model, a set of tests is generated for the legacy component. These tests are used to form a basis for guaranteeing behavior preservation by enabling engineers to refactor code in a test-driven way. The model is also used to generate an anti-corruption layer that protects the software component from 'illegal' behavior at run-time. Using these techniques we can reduce the risk of refactoring the component by increasing confidence in the preservation of behavior when refactoring.

Keywords: model learning, refactoring, legacy software, model based testing, test generation, anti-corruption layer, industrial application

Executive Summary

In this research, we explored a method to assist engineers in refactoring legacy software. Our method can be used as an addition to the current refactoring methods used by companies. It helps engineers gain a better understanding of the behavior of the legacy software and gain confidence in their refactored code by providing tests and an anti-corruption layer. By looking into the learned model and modifying it, we also force the engineers to think about the desired behavior of the software before refactoring it.

To gain a better understanding of the logic and behavior of legacy modules we used a model learning tool (LearnLib) to learn the behavior of the module in the form of a model. The past few years' model learning algorithms have improved significantly, but they are still not perfect. So to be sure that the model at least describes the behavior that is currently used in the field, we used system traces extracted from a simulator to guide the model learner. This still does not mean our learned model is always correct, but at least the behavior we need is described.

The resulting model might represent the current behavior, but this is not necessarily behavior the engineers desired for the module. If this is the case the model can be manually modified, a lot of tools can be found to assist engineers in doing so. We implemented multiple import and export formats to support as many of these tools as possible. Models can also be marked using the system traces. This way the behavior currently used by the system can be visualized in the model.

When the engineer is satisfied with the model it can be used to generate tests and an anti-corruption layer. Tests prevent engineers from damaging the rest of the system when refactoring the module by showing the refactored code preserves the old (or desired) behavior. While model-based testing is a known concept, not much could be found about the generation of tests that can be used in existing testing frameworks. This is strange since it seems more appealing for most companies. Our tool uses the HADS [1] tool to generate a set of test cases that cover all transitions of the model and is able to translate these into the JUnit and GTest formats.

Anti-corruption layers are meant as a run-time protection for the software that prevents and/or logs 'illegal' action sequences when being called on the legacy or refactored module. Such sequences may occur when external systems (that call the module) are being refactored. We formalize the anti-corruption layer and show 4 reference implementations: The *Observer*, *Armor*, *Enforcer* and *Shield*. For this research, we implemented the first three examples. The *Observer* monitors all operations of the module, and if a behavior violation is detected this will be logged. This implementation is mainly meant for debugging purposes. The *Armor* is a more aggressive form of the Observer. Besides logging the behavior violations the Armor will also block these violations and return an error. This way the Armor actively protects the module from undesired behavior. The *Enforcer* is a lot like the armor, but besides blocking the undesired operation, it will also trigger a reset sequence on the module. When undesired behavior is detected the module returns to its initial state. Finally, we looked at the *Shield* proposed in [2]. While this paper proposed to generate a shield from safety and liveness properties, we state it is also possible to generate it from a model. The Shield corrects illegal behavior rather than blocking it. All implementations have their strengths and limitations which are discussed in this thesis.

Our method is proven by concept using an industrial case at ASML. For confidentiality matters, we explained the method in more detail using an example project.

Preface

I would like to thank my supervisors (Frits Vaandrager and Ammar Osaiweran) for their guidance and help during this master thesis. Frits Vaandrager who signed an NDA with ASML and always gave good feedback so I could get the most out of my thesis. Ammar Osaiweran and the Metrology team who supported me during the project, were always there to answer questions, show me handy ASML tools and provide feedback on my thesis.
I also would like to thank my fellow students at Radboud University. Without you the last two years would not have been possible. I want to thank my girlfriend Astrid for always supporting me and Simone Meeuwsen from the Radboud University for providing a place to work during the weekends and in the evenings.
Finally, I would like to thank ASML for enabling me to conduct this research using their systems as a proof of concept.

Gijs van der Meijde - August 27, 2018

Contents

1 Introduction

An important aspect of software is the ability to evolve with new functionalities based on the demands of its users. This does not only mean adding new features, but also changing existing code. Even when adding new features, changing existing code is often needed. Thus, it is important for software to be maintainable. Unfortunately, this is not always the case. Many organizations face big problems when having to change legacy components in their software. Documentation is usually scarce and the original developers of the legacy modules often no longer work in the organization. Updating legacy modules to comply with more modern software architectures is time intensive and poses the risk of introducing bugs into the system. Since delivering new functionalities is restricted by time-to-market, many organizations leave legacy components intact and add the new features in new layers 'above' the legacy components as a quick solution. This results in high software complexity, lower software maintainability and thus not only postpones but also worsens the problems of low-quality code. Section 1.2 elaborates on the challenges engineers have to deal with when updating legacy code.

In this thesis, we propose a method to reduce the risks of refactoring legacy components using a combination of model learning and code/test generation.

1.1 Software maintainability

An important motivation to refactor code is improving its maintainability. According to IEEE's software engineering terminology (2012), maintainability is *"the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment"*. As stated before, an important aspect of software is the ability to evolve with new trends and the demands of its users. For example; new breakthroughs in mathematical algorithms can improve the space and/or time complexity and execution time of existing programs, keeping a company ahead of its competitors, but to profit from these new developments software must be able to change. Software architectures are usually created for just this purpose, but a lot of legacy code at companies does not yet profit from these architectures, making it harder to understand and maintain. Software maintenance is the most expensive part of software development. According to [3] it is estimated that maintenance after software delivery even takes up to 70% of the development costs, as can be seen in Figure 1. This shows us that also from a financial point of view, it is important to develop software that is easy to maintain and evolve. To help developers achieve this the Software Improvement Group [4] drafted 10 guidelines to measure the general quality of software:

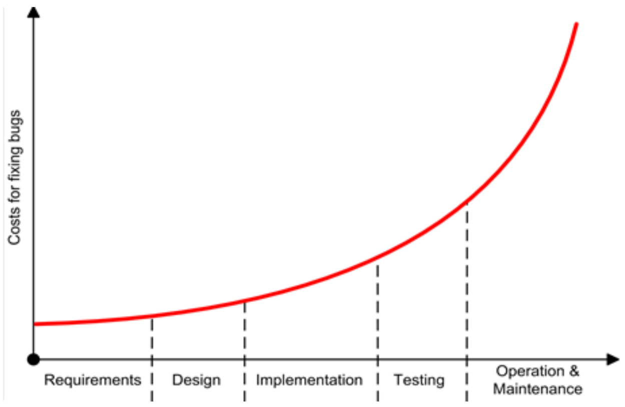


Figure 1: Costs of software development

1. Write short units of code (functions should be <= 15 lines).
2. Write simple units of code.
3. Write code once (no duplicate code).
4. Keep unit interfaces small.
5. Separate concerns in modules.
6. Couple architecture components loosely.
7. Keep architecture components balanced.
8. Keep your codebase small.
9. Automate tests.
10. Write clean code.

Additionally [5] states a function should take no more than 3 parameters. These guidelines are designed to guide developers in creating their software in such a way that their software is more flexible to the changing environment. The guidelines can also be used to measure the improvement of the refactored code over the original code.

1.2 Refactoring legacy software

To gain a better understanding of the challenges that await us when updating legacy systems we must first understand what refactoring is. For this reason, a literature study was conducted. The concept of refactoring means restructuring code (modules, classes, variables, methods) to become more future-proof[1], without actually changing the external behavior of the code. The IEEE provides a more formal description in [6]. They distinguish 3 types of software 'refactoring', namely: *Restructuring, Refactoring* and *Re-engineering*.
Restructuring is defined as *"the transformation from one representation to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics). A restructuring transformation is often one of appearance, such as altering code to improve its structure in the traditional sense of structured design. While restructuring creates new versions that implement or propose change to the subject system, it does not normally involve modifications because of new requirements. However, it may lead to better observations of the subject system that suggest changes that would improve aspects of the system."* [6].
Refactoring hints more at an object-oriented approach and is defined as *"the process of changing a [object-oriented] software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure"* [7].
Re-engineering is a more extreme form of restructuring/refactoring. IEEE [6] describes this as *"The examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. In this context, restructuring is needed to convert legacy code or deteriorated code into a more modular or structured form or even to migrate code to a different programming language or even language paradigm"*.

Regardless of the type of refactoring, the behavior of the system is preserved. Behavior preservation means that, for the same set of input values the resulting set of output values should be identical before and after refactoring. This can even be extended with the execution time of software (for real-time systems), power consumption (for embedded systems) or safety properties (for safety-critical systems). Another, slightly weaker, notion of behavior preservation is *call preservation* [8]. This means that the way a system behaves on function calls is the same before and after refactoring, which implies changes are internal while external interfaces remain unchanged. In this research we use the latter notion, since the models we learn using model learning (Chapter 5) mostly describe the external call behavior of the software. Proving behavior preservation can be done using an extensive set of test cases or, as demonstrated in [9], by model checking.

[1] following the 10 guidelines discussed in Section 1.1.

Our method does not limit itself to one of the refactoring types. As long as the external interfaces of the component that needs to be updated stay intact, our method will be applicable. For convenience, we shall limit ourselves to the more commonly used term 'refactoring' in this thesis.

To gain a better understanding of the global refactoring process, [8] distinguishes the following activities:

– Identify where the software should be refactored.
– Determine which refactoring(s) should be applied to the identified places.
– Guarantee that the applied refactoring preserves behavior.
– Apply the refactoring.
– Assess the effect of the refactoring on quality characteristics of the software (e.g., complexity, understandability, maintainability) or the process (e.g., productivity, cost, effort).
– Maintain the consistency between the refactored program code and other software artifacts (such as documentation, design documents, requirements specifications, tests, etc.).

Additionally, we propose to determine the state of the to be refactored software using the 10 guidelines previously discussed in Section 1.1, so improvement can be 'measured'. Identifying where software should be refactored mostly happens by detecting so-called *'bad smells'*. These are *"structures in the code that suggest (sometimes scream for) the possibility of refactoring"* [7]. Good examples of bad smells are code duplication or high dependencies (cyclic dependencies) which can be detected using static analysis tools like CodeSonar [10] or BetterCodeHub [11].

The company ASML also has to deal with legacy code that is very risky to change because it is embedded deeply in their systems. For their more recent code, they make a clear separation between data, control, and algorithms. Their legacy code however does not follow a specific pattern. This means that, to be able to improve the maintainability (Section 1.1) of their software, it is important to refactor, restructure or even completely re-engineer this code.

1.3 Models as Mealy Machines

Software is considered *stateful* if it records information about preceding events (like function calls) and adjusts its behavior accordingly. If the recorded information changes the behavior of the software this is called a *state*. Software behavior can be represented using a collection of these states, and can be described using a model. This research uses Mealy Machines [12] to represent models. Let us consider Figure 2 as an example to explain Mealy Machines. Every node in this Mealy Machine (S_0, S_1) represents a state of a system. The transitions represent the inputs with a corresponding output of the system. If we take S_0 as our initial state we can feed the Mealy Machine input a which results in output 1 and brings us in state S_1. If we feed the machine input b in state S_0 this will result in output 0 and leaves us in state S_0.

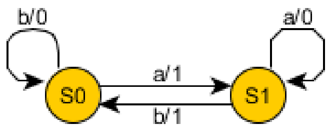


Figure 2: Mealy Machine

This definition assumes that a Mealy Machine is deterministic and input enabled, as is also the case for our models. Formally, a Mealy Machine is described as a 6-tuple: $(S, s_0, \Sigma, \Lambda, T, G)$. These 6-tuples consist of:

- S:	A finite set of states.
- s_0:	An initial (start) state $s_0 \in S$.
- Σ:	A finite set of inputs.
- Λ:	A finite set of outputs.
- T:	A transition function $T : S \times \Sigma \rightarrow S$ mapping a state and an input symbol to the next state.
- G:	An output function $G : S \times \Sigma \rightarrow \Lambda$ mapping a state and an input symbol to an output symbol.

An important rule for Mealy Machines is that for each state in S, for every input defined in Σ a transition is defined. If an input does not make the system change its state, the corresponding transition should loop back to the same state (See Figure 2: $T(S_0, b)$). In this thesis the input alphabet Σ of our models represents the functions we call on the external interface of the system. The output alphabet Λ represents the return values of the functions that are called on the system. Any symbol $e \notin \Sigma$ is not accepted by the system and will result in undefined behavior. This means it depends on the software how these inputs are handled, some programs for example might just ignore these inputs while others might crash completely.

1.4 Research goal

This study was conducted at ASML to find a uniform approach for their refactoring projects. The goal of this research was to explore a method that assists engineers in refactoring legacy code using model learning. In this thesis, we focus on the reasons for refactoring, the risks that come with it and how to reduce these risks. We also present guidelines to assist engineers in writing more maintainable code. If engineers adhere to these guidelines, this would benefit future refactoring.

To reduce the risks in refactoring, a method using a combination of model learning and text/code generation was explored. This method uses the learned model to generate unit tests and an anti-corruption layer that helps engineers gain confidence in the behavior of the refactored code. The generated unit-tests can help prove behavior preservation after refactoring, while the anti-corruption layer is used to enforce certain behavior at run-time.

In addition to providing software engineers with a uniform approach for their refactoring projects, the purpose of exploring this method is showing the current strengths and limitations of the used model-based techniques in similar settings. To show these strengths and limitations, the method was applied to an industrial case at ASML. This thesis discusses these and appoints some possible solutions for the limitations.

1.5 Research questions

The goal of this thesis is to apply model-based techniques and in particular model learning in industrial settings. For this research, we chose to use these techniques to facilitate the refactoring process of complex software components. We explored a practical application of model learning; assisting engineers in refactoring legacy code. The direction this research takes is based on the pre-conducted study on the problems in software refactoring described in the previous sections. The main research questions addressed during this research are:

(RQ1) Can we provide engineers with a method to refactor legacy code based on model learning?
(RQ2) Is model learning mature enough to be applied in an industrial setting?
(RQ3) When model learning is applied for refactoring industrial software, what are the benefits and the limitations?
(RQ4) Based on experience gained in the industry, can we provide engineers with general guidelines for software refactoring?
(RQ5) Can we automatically generate tests from a learned model to assist engineers in refactoring legacy code in a test driven way?
a) If so, what is the most convenient test framework to use?
(RQ6) Can we use models to guarantee behavior preservation when deploying the refactored code?

Some of the subjects mentioned above are also treated by M.T.W. Schuts in his doctoral thesis [13] at Phillips. However, as he describes in his thesis, his research was focused on a specific industrial case. The industrial case used in this thesis yields valuable experience from a different perspective, that can be used to shed a different light on Model Learning in industrial applications.

1.6 Outline of this thesis

This thesis is structured as follows: prior to this research, a literature study was conducted to gain a better understanding of the problems and importance of refactoring software. The findings of this study are discussed in the introduction of this thesis (Chapter 1). Next, we propose a solution by briefly describing a method for reducing the risks of refactoring in Chapter 2. To support our method we applied it to an industrial case study at ASML. Chapter 3 explains our industrial case in more detail and Chapter 4 explains how the case was prepared for model learning.

To help overcome the problems in software refactoring, we use a combination of model learning techniques, explained in Chapter 5, and code/test generation to assist engineers in refactoring legacy code. The learned model can also be used by the engineers to gain a better understanding of the behavior of the system.

In Chapter 6 we show it is possible to generate tests from a learned model to help preserve behavior when refactoring. These tests can be used to refactor legacy code in a test-driven way. Chapter 7 introduces an anti-corruption layer that can also be generated from the learned model. This anti-corruption layer is used to protect the system from 'illegal' behavior at run-time and thus further reduces the risks of refactoring.

To put our method to the test we carried out some light refactoring in Chapter 8. While doing so we encountered some limitations which are discussed in this chapter.

Finally, we end this thesis in Chapter 9 by reflecting on the research and discuss our proposed methods strengths and weaknesses and possible extensions to improve the method.

2 Methodology

In this chapter, we propose a method to refactor software in a safe manner. The different components of this method are explained in more detail in their own chapters. Figure 17 shows an overview of the steps for refactoring the system as explained below.

1.) Model learning is a technique to learn the behavior of a system from a black-box perspective. The piece (slice) of software you want to learn the behavior of, depends strongly on the type of refactoring you want to apply (Section 1.2) and the way the legacy software is structured. If, for example, the components of the software are strongly connected but contain little logic it might prove more effective to learn bigger chunks of, or even the entire, software at once. If the software contains a lot of logic, like in our industrial case, it might prove more effective to isolate a smaller module of the software and learn its behavior first. Thus refactoring the software step by step.
To learn the behavior of a software module using automated learning tools, these tools need to be able to communicate with the software module. This means an adapter to replace the system that normally calls the module by the learning tool needs to be created. To properly learn the behavior of this single software module and not that of its dependencies, the dependencies should be replaced using so-called stubs. The chosen module may also contain cyclic dependencies with other modules. In order to isolate the desired component these cyclic dependencies need to be resolved. By generating a graph of all dependencies and distinguishing all strongly connected components in this graph, it is possible to see how strong or loose the different modules of a system are coupled. This helps in identifying the specific functions that need to be refactored to uncouple a specific component. In Chapter 4, we describe the method we applied for isolating the module in our industrial case.

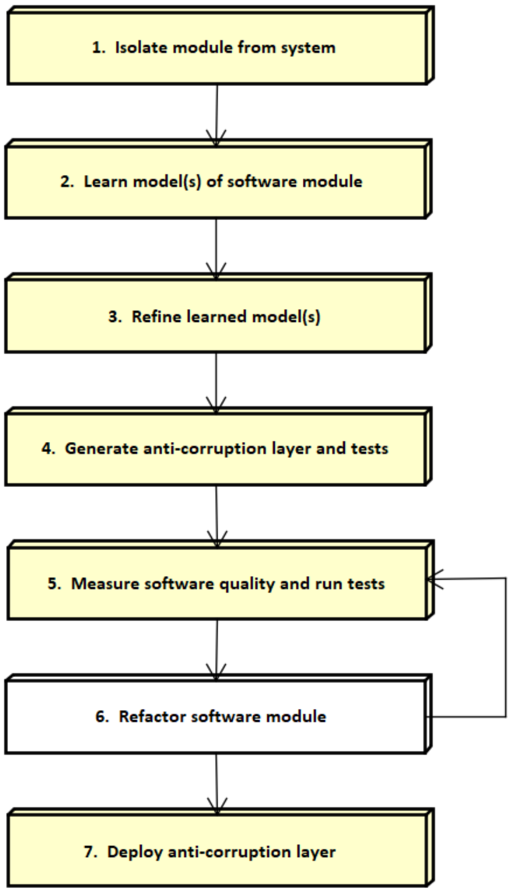

1. Isolate module from system
2. Learn model(s) of software module
3. Refine learned model(s)
4. Generate anti-corruption layer and tests
5. Measure software quality and run tests
6. Refactor software module
7. Deploy anti-corruption layer

Figure 3: Proposed refactoring method

2.) To learn the behavior of the legacy module as precisely as possible, we used a combination of model learning and analysis of system traces from our industrial case (trace analysis). Model learning is a black box method to learn the behavior of a system, but because it is black box it is not guaranteed to learn the complete behavior of the system. If, for example, a system responds identically the first 100 times a button is pressed but responds differently the 101st time, most model learning algorithms will already have stopped 'exploring' that button and not discover this change in behavior. On the other hand, model learning tries to learn all possible combinations of actions on the system. This means it might also learn behavior that should not occur during a normal run of the system. This might be because of defensive programming (the software just ignores illegal moves) or because the original developer never expected someone to use the module in this way.

Trace analysis however might learn a model that does not cover all possible traces because some functions in the module are never (or rarely) used. This means the learned model contains all the currently used behavior but is not robust to future changes. For our method, we propose using the system traces to guide our model learner such that the learned model covers at least these traces. This means while we still cannot guarantee the learned model covers all possible behavior, it at least covers the behavior that is currently used by the system. Chapter 5 describes the method we applied for learning the behavior of our example and industrial case.

3.) While the learned model might reflect the current behavior of the module, this does not necessarily mean it is the desired behavior of this module. Since the model learner tries to learn all possible actions on a system, it might also find flaws in its behavior. If this is the case we propose refining the behavioral model before refactoring the component, in this way the generated tests will guide the engineer in implementing the correct behavior when refactoring the learned software module. To assist the engineers in refining the model, the system traces are used to mark all inconsistencies between the learned and the currently used behavior. After refining the model, the system traces can again be used to increase confidence that the new model does not conflict with the currently used behavior. Section 5.3 discusses this in more detail.

4.) As discussed in Section 1.2, one of the most important parts of refactoring is preserving the behavior of the original (legacy) code. Unfortunately, as discussed before, learning the behavior of the model might reveal flaws in the current behavior. Thus, rather than preserving the behavior of the original code, we propose preserving the desired behavior as described in the refined model. To assist engineers in preserving this behavior, a set of (unit)tests can be generated from the model. These tests show the engineers whether their code is correct according to the desired behavior and thus enable refactoring in a test-driven way. In Chapter 6 we present a comparison of different test generation tools and show how we applied test generation to the models of our example and industrial case.
To protect the refactored system from 'illegal' behavior at run-time, we introduce an anti-corruption layer. This layer can be seen as a wrapper for the software module that checks all actions that are called and keeps track of the current state. If an action is triggered that is not allowed in the current state the anti-corruption layer will block (and/or log) this action. Chapter 7 elaborates on this concept.

5.) To be able to measure the improvement gained by refactoring the software module, the quality of the legacy module can be measured using the tools and guidelines discussed in Section 1.1 and Section 1.2. The results of these measurements can then be used to guide the engineers in the refactoring process. When the model used for generating tests is refined, these tests might fail on the current implementation of the legacy code. The results of these tests then show the engineers what to refactor and can be used to confirm the refactored code's behavior is still correct to the model after each refactoring step.

6.) Finally, the engineer can refactor the software module in a step-wise manner. By using the generated tests this can be done in an iterative way. Rather than refactoring the entire module at once we prefer to refactor a small part of the module, running the generated tests to gain confidence in the refactored code, measuring the quality again to compare this to the previous quality metrics. Only if everything is correct the engineer should start refactoring the next part of the module.

7.) To make the system more robust against 'illegal' behavior, we generated an anti-corruption layer from our model and apply it to the learned module. By refactoring the software module using the generated tests in the previous steps, we gained confidence the refactored module behaves as described in the model. The anti-corruption layer however, helps guarantee the module is also used this way by external systems. Thus after refactoring the software, we apply an anti-corruption layer, as explained in Chapter 7, to make the entire system more robust against unexpected behavior.

3 Industrial Case

"Make chips. Faster, smaller, greener."

Founded in the Netherlands in 1984 and listed on the stock exchanges NASDAQ and Euronext since 1995, ASML is currently the largest supplier of photolithography systems for the semiconductor industry. ASML systems engrave circuits on silicon wafers using a laser that can do this with a resolution of 13nm. In comparison: the size of a skin flake is between 500nm and 10.000nm [14]. After engraving the wafers can be processed further to form Integrated Circuits (IC's) or chips. ASML does not manufacture IC's, it only develops the engraving systems. Important customers of ASML are companies like Samsung, Intel, and TSMC.
This chapter introduces the TWINSCAN machines (Figure 4) developed by ASML and the component used as a case study for the refactoring method described in this paper.

Figure 4: TWINSCAN

3.1 The TWINSCAN system

ASML's current line of photo-lithography systems is called the TWINSCAN. These systems use light to print an image on a silicon wafer covered with a photo-sensitive material. The most recent TWINSCAN system uses Extreme UltraViolet (EUV) light to engrave circuits with a 13nm resolution. A laser is fired at microscopic droplets of molten tin, this produces a plasma which emits the EUV light used to engrave the wafers. The reason EUV is used is that it has an extremely short wavelength which is needed for projecting images on such a small scale. The EUV light passes through a screen with the image and displays this on the wafer. After exposure to the image, the unexposed parts of the photosensitive material are etched away, leaving only the projected circuit. After this, a new layer is added to the wafer and the process is repeated

(Figure 5). the different layers do not necessarily have to be processed using the same lithography machine.

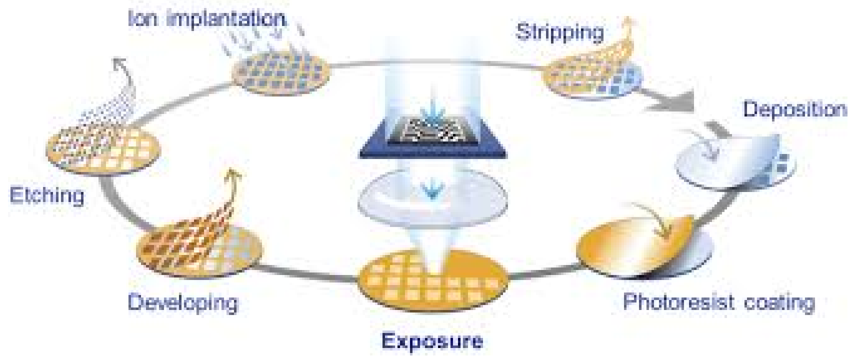


Figure 5: Wafer production [15]

To stay ahead of its competitors, ASML has to develop new technologies to keep decreasing the resolution with which their systems can engrave wafers. This way chips can become smaller, have more capacity and use less power. Until now this goes according to Moore's law [16]. This law predicts circuits to become smaller by a factor of 2 every two years.

3.2 Metrology

"Metrology is the brain of the TWINSCAN that is used to create electronic devices that seem impossible today and that shape our future."

During processing the wafers are being subjected to fast movements (Formula 1 acceleration rates even) and extreme temperatures, any deformation caused by this can cause imperfections in the photolithography process. When working on a nano-metric scale, every unevenness can be fatal. This is why every wafer is constantly measured carefully. From these measurements mathematical models are created and compared to the expected models, this way any deformation can be predicted and countered, optimizing the yields and enabling ASML to stay ahead of its competitors. The department responsible for the software that handles the measuring of the wafers and correcting for mechanical imperfections is Metrology. Their mission statement:

" 'Metrology Measure' measures, models, and corrects wafer deformation at the performance level required to win race against Moore's law - over and over again."

3.3 Case-study

This part is removed for confidentiality and can be found in the original version of this document.

3.4 Chocolate vending machine

In order to prevent spilling confidential information we use an example application to show the different techniques in detail. The example application is a simple vending machine that accepts *5ct* and *10ct* coins, and sells *Mars*, *Twix* and *Snickers* for respectively 10ct, 15ct and 25ct. It is based on a JavaScript web-application used during the Testing Techniques [17] course at Radboud University [18]. The vending machine application has been rewritten to a Java application that uses the *java.net.Socket* to communicate with the learner. The source code for the example application is added in the appendices (Chapter 10) so all our experiments are repeatable. Controlling the application can be done by sending the input commands depicted in Table 2 over a socket connection. The application responds with *OK* if an action is allowed and *NOK* otherwise.

Inputs (Σ)	**Outputs (Λ)**
5ct	*OK*
10ct	*NOK*
mars	
twix	
snickers	

Table 2: Inputs and outputs of the vending machine

4 Isolating the system

For our industrial case, we used an ASML module as our SUL[2]. Since the module is still connected to other modules the resulting model will also reflect the behavior of these modules, potentially giving a wrong impression of our SUL's behavior. Thus, to learn the behavior of only our SUL, it has to be isolated from the rest of the code.

4.1 Removing cyclic dependencies

This part is removed for confidentiality and can be found in the original version of this document.

[2] System Under Learning

4.2 Creating the stubs and adapter

As stated in Section 1.2, the scope of the SUL depends on the software and type of refactoring that is applied. For example, our vending machine application was relatively small and can be learned in one go. But including too much in your scope might result in a very big model, including too little might mean you need to learn multiple modules. In this research we decided to learn the behavior of just the SUL (without its dependencies), thus connections to these dependencies have to be separated.

Since we want to change our SUL as less as possible before learning its behavior, we created stubs to replace the dependencies. A stub is basically a 'dummy' implementation for a class or module. Most of the original dependency functions returned an integer code referring to either an error type or a successful execution. Our stub functions always return 0, which means the functions executed successfully. By doing so we tried to cover the 'happy flow' of our dependencies assuming they will always pass the execution and made sure we only learned the logic contained in our SUL and not that of its dependencies. Some exceptions were made for functions that needed to work for the SUL to function. For example, one of the stubbed functions returns a predefined wafer diameter and a stubbed 'clone' function actually clones a variable. But only a very basic implementation was used.

After removing the cyclic dependencies the structure of the code closely resembled that of Figure 7 (left). Since the original dependencies implemented their corresponding header files, we decided the stubs should do the same. The SUL is not aware of the implementation of these header files and thus it does not need to be modified to use the stubs instead of the original implementations. Because a source (implementation) file can implement multiple header files, we created one stub that implements all the header files as shown in Figure 7 (right).

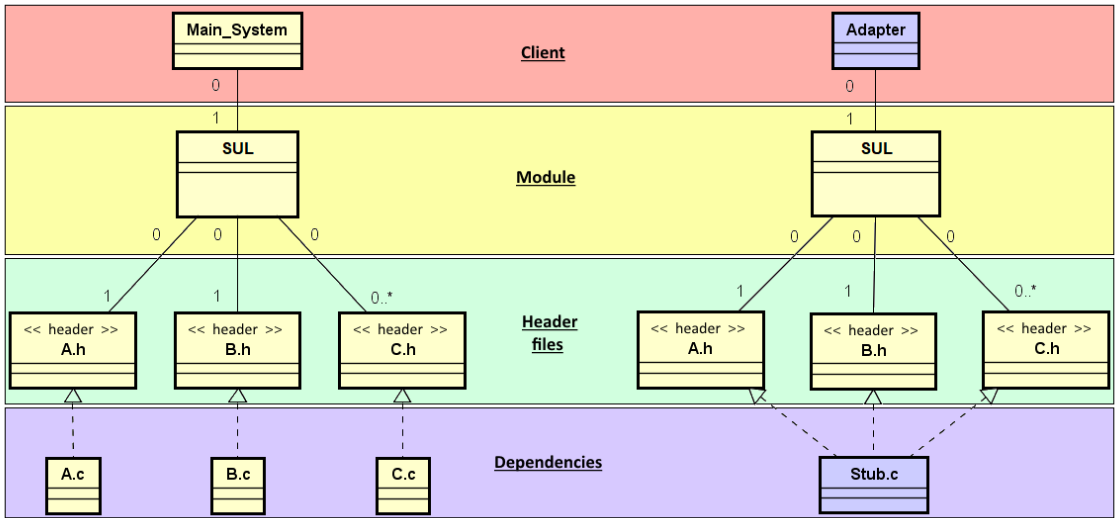


Figure 7: *Left:* dependencies before isolating. *Right:* dependencies after isolating.

To communicate with the SUL we had to replace the system that usually calls the module with an adapter for our model learner as shown in Figure 7. LearnLib only supports string input and output, thus our adapter had to translate these string messages to actual function calls. To be able to influence some of the function parameters, we used multiple string messages to denote these function calls with different parameters. For example, function *foo* contains a boolean parameter. This can be resolved by using two different messages: *foo_true* and *foo_false*. To communicate with the learner, our adapter uses a TCP/IP socket connection. The adapter receives commands listed above via the socket connection and responds by calling the corresponding functions of the SUL and returning their outputs. To keep our model small, we abstracted the outputs to *OK* and *NOK*, where *OK* denotes a successful execution, and any error type is being translated to *NOK*.

4.3 Resulting system

After isolating the SUL, we are able to compile the SUL and its dependencies as a separate library. By replacing the dependency modules by our stubs and adding the adapter, we compiled our SUL as a standalone executable. This way, we are not dependent on the ASML systems. Using this executable, we learned the models by connecting the SUL to LearnLib via a TCP/IP connection as shown in Figure 8. Chapter 5 continues by describing how we learned models from the SUL.

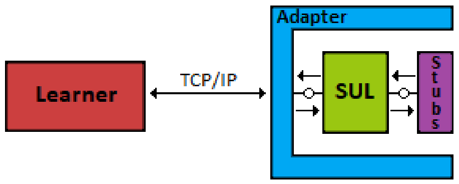


Figure 8: Connecting the learner to our SUL

5 Learning the model

After isolating the SUL as described in Chapter 4, we can begin learning its behavior. Therefore, different model learning techniques can be applied to the component. This chapter explains the model learning techniques used to construct behavioral models of legacy components. First, we will give a brief introduction to model learning using a simple vending machine as case-study in Section 5.1. Next, Section 5.3 continues by explaining the learning methods used on the SUL in our industrial case. Finally, Section 5.2 explains why it is sometimes needed to refine the learned models to describe the desired behavior of the refactored code.

5.1 Chocolate vending machine

In [19] it is stated that *"Model Learning aims to construct black-box state diagram models of software and hardware systems by providing inputs and observing outputs"*. But what does this mean? Looking at a system from a black-box perspective means looking at its external behavior without checking its internal structure. The most intuitive ways to learn a system like this is by trying every possible sequence of inputs and see what happens. Since this is impractical due to the infinite number of possible input sequences, there are algorithms that can help us 'try sequences' in a more efficient way.

One of these model learning algorithms is the L* algorithm [20]. This algorithm uses a matrix to learn a model from the input and output of a system. Every system has at least one state (the initial state). The L* matrix consists of a row for each examined prefix and a column for each suffix (input). The initial state is denoted using an empty (ϵ) prefix and a suffix for each input, the values in the matrix represent the given outputs. In Table 3 we see that in its initial state the vending machine returns *OK* on inputs *5ct* and *10ct* whereas the inputs *mars*, *twix* and *snickers* return *NOK*.

Prefix	**Suffix**				
	5ct	10ct	mars	twix	snickers
ϵ	OK	OK	NOK	NOK	NOK

Table 3: L*: Initial state

When the L* algorithm discovers a new state, which can be seen as a unique combination of output values, it uses the prefix for this state in combination with all suffixes as a new set of prefixes to be explored. So after the initial state is discovered, the L* algorithm will try each input for the initial state (ϵ) to discover where they lead. Table 4 shows this results in five new rows.

Prefix	**Suffix**				
	5ct	10ct	mars	twix	snickers
ϵ	OK	OK	NOK	NOK	NOK
5ct	OK	OK	NOK	NOK	NOK
10ct	OK	OK	OK	NOK	NOK
mars	OK	OK	NOK	NOK	NOK
twix	OK	OK	NOK	NOK	NOK
snickers	OK	OK	NOK	NOK	NOK

Table 4: L*: First iteration

The highlighted rows in the tables are considered as a new state because they result in new unique sets of outputs. Four of the five new rows in Table 4 consist of a set of outputs that equals the set of the initial state. These rows will not be considered as a new state, instead, these sequences lead to the initial state. The prefix *10ct* (row 3) results in a new set of prefixes that did not occur in the matrix before. This means a new state is detected and L* will now continue exploring using a set of sequences consisting of the newly found state (*10ct*) and all possible suffixes (inputs). The result of this continued exploration is added in Table 5.

Prefix	**Suffix**				
	5ct	10ct	mars	twix	snickers
ϵ	OK	OK	NOK	NOK	NOK
10ct	OK	OK	OK	NOK	NOK
5ct	OK	OK	NOK	NOK	NOK
mars	OK	OK	NOK	NOK	NOK
twix	OK	OK	NOK	NOK	NOK
snickers	OK	OK	NOK	NOK	NOK
10ct 5ct	OK	OK	OK	OK	NOK
10ct 10ct	OK	OK	OK	OK	NOK
10ct mars	OK	OK	NOK	NOK	NOK
10ct twix	OK	OK	OK	NOK	NOK
10ct snickers	OK	OK	OK	NOK	NOK

Table 5: L*: Second iteration

This time it appears that prefixes *10ct 5ct* (row 7) and *10ct 10ct* (row 8) result in a new state. The L* algorithm will now continue exploring from the newly found states. It will continue this way until no new unique sets of outputs are discovered. If no new states are detected the algorithm will consider the model as complete. If the system actually contains more states the model learner can be guided using counterexamples. These counterexamples are sequences of actions that result in a new state. In the L* Algorithm these counterexamples are added to the list of suffixes and will form new columns. The L* Algorithm will then consider these new columns and check if new states can be found.

While L* is probably the easiest model learning algorithm to understand, it is not the most efficient one. There are a number of research papers available on model learning. For this research, we looked at a paper about refactoring software using model learning and equivalence checking at Philips [21], and a thesis about refactoring using model learning and Domain Specific Languages [9] that builds on the previous paper. In [21] the L* algorithm was chosen for model learning, however, the author notes this algorithm might not be the most efficient way and performed tests with the TTT learning algorithm [22] showing it is much faster.

Both papers use LearnLib [23] to learn a model of legacy software. LearnLib is a black-box model learning tool that learns a model without looking at the internal structure of the SUL. It is released as a Java library that contains multiple learning algorithms and can be connected to a SUL by writing your own adapter as shown in Chapter 4. After LearnLib finishes learning, it generates models in graphviz .DOT format.

To enable communication between the LearnLib tool and the SUL, we use TCP/IP Sockets since they provide a generic way to communicate, which makes them easier to reuse on other SUL's. To increase the confidence in the model learner we first tested it on a simple example application of which we knew the model beforehand: the chocolate vending machine example explained in Section 3.4.

5.1.1 First model

Our first experiments with the TTT algorithm show that it is very fast indeed, which is in line with the observations of [21]. But without prior knowledge of the system it cannot learn a complete model. The initial model in Figure 9 shows a stateless behavior which was far from complete.

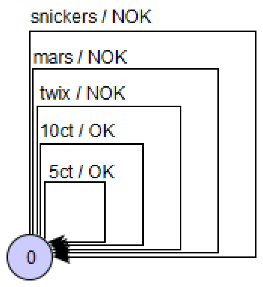


Figure 9: Initial TTT model - Vending machine

While the behavior in the model is indeed correct, it is immediately clear the model learner did not learn more than one action per sequence. If it, for example, would have tried the *mars* action after the *10ct* action it would have found a new state. In our example application, the amount of money is also limited using the function below. This makes every state that contains a self-loop where money is added and OK is returned suspicious.

```
private static boolean addMoney(int x){
  if(money <= 25){
     money += x;
     return true;
  }
  return false;
}
```

Using the input sequence $\{10ct\ mars\}$ for example, we can disprove the learned model. This sequence should result in *OK* as response, but according to Figure 9 the response would be *NOK*. If such a sequence exists, we call it an counterexample. These counterexamples can be used to guide the learner in learning more complete models. The initially learned model shows the TTT algorithm alone was not enough for our learner to get a complete image of the system's behavior. Therefore, we had to give the learner more information, in the form of counter-examples, to guide it in exploring more behavior. Fortunately, LearnLib contains several testing algorithms to automatically find these counterexamples. In Section 5.1.2 we show how the initial model was improved.

After our first try with the TTT algorithm failed to gain a complete model, we tried to learn a new model using the L* algorithm. Except for the learning algorithm, we kept the setup of our learner identical to our initial attempt. Figure 10 shows the first model learned with the L* algorithm. This experiment shows that, without any additional information, the L* algorithm learns more complete models.

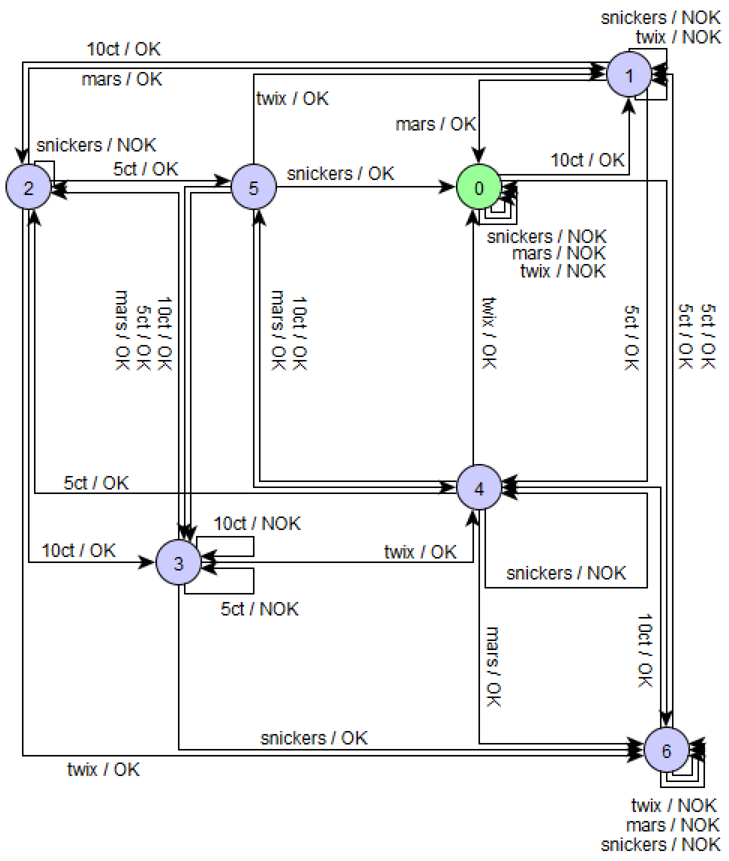


Figure 10: Initial L* model - Vending machine

5.1.2 Refining the model

If a model is not correct with respect to the SUL, counterexamples can be presented to the learner to improve the learned model. The learner will check if these counterexamples indeed conflict with the learned behavior and are correct according to the SUL. If a presented sequence is indeed a counterexample, it will be considered a new state and the learner will continue learning the behavior from this state. While engineers can present counterexamples manually, LearnLib also features multiple algorithms to automatically find counterexamples. One of these algorithms is the W-method [24], which is explained in Section 6.1.17.

To improve the learned model, the experiments in Section 5.1.1 were extended with the W-method to find counterexamples. Both the L* algorithm and the TTT algorithm now learned the same model shown in Figure 11.

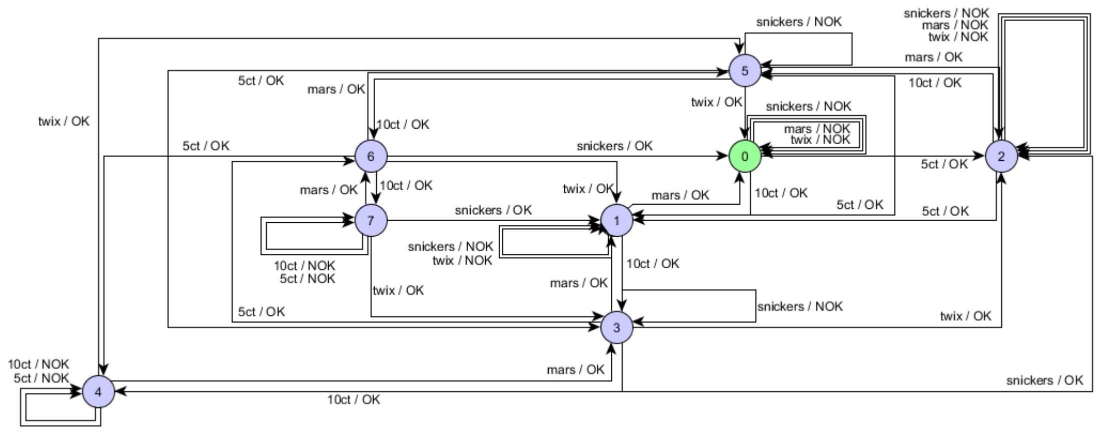


Figure 11: Improved model - Vending machine

Normally we would try and improve the model by experimenting with different learning techniques and finding more counterexamples, but since we created the vending machine ourselves we are confident this model reflects its actual behavior. So trying different learning techniques should have no positive effect on the learned model.

To be able to learn models with different Model Learning algorithms and use counterexamples in a fast and easy way, we developed a prototype tool using LearnLib as a basis (Figure 12). By using this tool an engineer can select one of the model learning algorithms embedded in LearnLib and configure all possible inputs and outputs. It is also possible to feed the tool counterexamples to guide LearnLib in learning more complete models. As can be seen in Figure 12 the prototype tool automates a big part of the method described in Chapter 2. It featured the model learner (LearnLib) explained in this chapter and is able to export and import models. From these models, it can generate tests and an anti-corruption layer, which will be explained further on in this thesis.

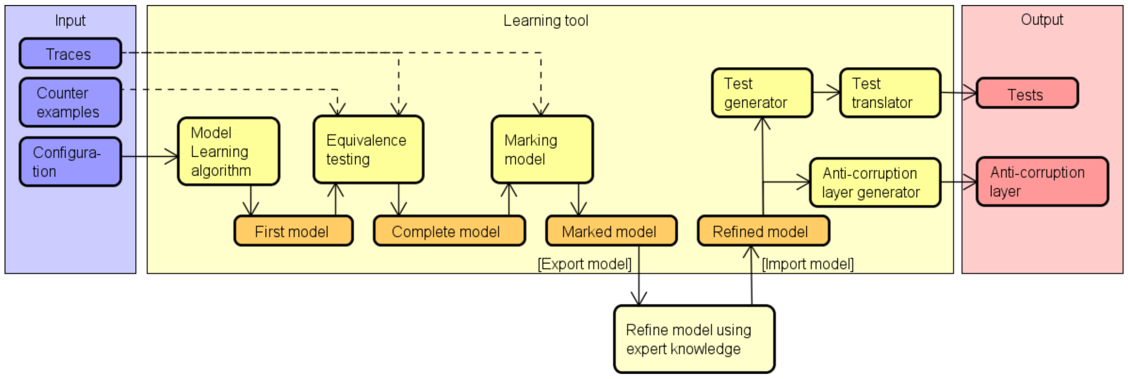


Figure 12: Prototype learning tool

5.2 Industrial case

For our industrial case at ASML, we learned the legacy module explained in Chapter 3. In preparation of learning the SUL, it was isolated from the rest of the code as described in Chapter 4. In this section, we discuss the model learning process for the ASML SUL. For confidentiality, we maintain an abstract perspective.

5.2.1 The first model

After isolating the SUL and running the first model learning algorithms we learned the model shown below. Since the SUL responded nearly instantly and our previous experiment showed the L* algorithm learned a more complete model without any additional information, we decided to use this algorithm for our industrial case.

This part is removed for confidentiality and can be found in the original version of this document.

5.2.2 Improving the model

This part is removed for confidentiality and can be found in the original version of this document.

Guiding the learner with system traces

As explained in Section 5.1, model learning algorithms can be guided using counterexamples. These counterexamples are sequences that are possible in the SUL, but not in the learned model. If such a sequence is discovered the model learner can use this counterexample to discover a new state and continue learning from this state.

Since legacy systems usually are fully operational, it is possible to trace function calls as the system is being used. These traces give us a good picture of the currently used behavior of the system. Since systems often do not utilize all possible behavior, the traces will be a subset of the actual behavior of the system. By translating the extracted traces to counterexamples, LearnLib can disprove the correctness of a learned model and modify the model accordingly. This way, the previously learned model might be improved.

One of the options of the simulator ASML developed for their TWINSCAN hardware, is to log system traces in an external file. These traces contain all functions that are called in the order they are called. The simulator also features predefined test cases to check how the system behaves under certain circumstances. Running these tests on the current software yields system traces that describe the system's behavior as it is at run-time.

The tool we created is able to use the extracted trace files as counterexamples for LearnLib. Figure 12 shows the global activity flow of our tool. By using a configuration file, the inputs and outputs of the SUL can be configured and one of the model learning algorithms and Equivalence testing methods (used for automatically finding counterexamples) embedded in LearnLib can be chosen. All the traces we used were already contained in the model learned in Section 5.2.2, so the model stayed unchanged. While we can not guarantee the learned model is correct to the SUL, we can be certain it at least contains all traces used by the current systems. The learned model did however contain undesired behavior. In the next section, we explain how we refined the learned model and discuss the results.

5.2.3 Refined model

After learning the final model, we presented the model to domain experts from ASML. To highlight the current behavior of the system, we used the system traces, explained in Section 5.2.2, to mark all used transitions in the model by symbolical execution ('walking' through the model using input sequences as steps). The marked transitions form the subset of the behavior that is currently used at run-time.

Transitions that are not marked, and thus not used in the current implementation, are not necessarily wrong, but they are suspicious. Vice versa transitions that are marked are not necessarily correct to the desired implementation, but we advise extra caution when changing this behavior. If the behavior that is used at run-time is changed, this might cause unexpected errors in the system.

This part is removed for confidentiality and can be found in the original version of this document.

The new model is used to generate tests (Chapter 6) and an anti-corruption layer (Chapter 7).

We work with a white-box SUL, and the model learning techniques used in this research are created for black-box learning. This means we learn the software by calling its interfaces rather than inspecting its internal behavior. It might prove worthwhile to look into tools for static code analysis to learn models from the SUL, like CPAchecker [25]. Unfortunately, due to the limitation of time, we did not do this during our research.

5.3 Modifying the model

Before the refactoring process of the code takes place, we want to use the learned model to generate tests and other artifacts, like the anti-corruption layer described in Chapter 7, to guard the correctness of the refactoring process later on. In Section 1.2 it is stated that refactoring should preserve the behavior of the legacy code. However, as we discovered in Section 5.2.3 hidden flaws (bugs) in the current behavior might be revealed during the process of refactoring code.

One of the advantages of using model learning is that when learning models of the legacy code these models show us the exact behavior of the current implementation. Thus any flaws, if they exist, will appear in the learned model. When such flaws are discovered refactoring does not limit itself to redistributing code without changing its behavior, but also means solving possible issues. This means the behavior of the refactored code is not necessarily the same as that of the legacy code.

When undesired behavior is detected in the learned model, generating tests and other artifacts from this model does not make sense because this behavior should first be corrected in the legacy code. However, we think correcting these flaws in the legacy code to learn a new model is a waste of time since this code is going to be changed (or completely replaced as in our industrial case) when refactoring anyway. A more logical move is modifying the learned model to reflect the desired behavior. This way we can generate tests and an anti-corruption layer for the behavior that is actually intended and implement that behavior when refactoring the legacy code.

This part is removed for confidentiality and can be found in the original version of this document.

Because the learned model might include faulty behavior that should be corrected, our prototype tool is able to export the learned model to the various output formats named in Appendix B. This way, engineers can use a tool of their own choice to modify the model to represent the desired behavior. After a model is modified it can be imported to our tool again and then tests and an anti-corruption layer reflecting the desired behavior rather than the learned behavior can be generated. These tests can then guide refactoring the code and be used as a basis to guarantee correctness. The process to reach the final refined model is visualized in Figure 12.

External systems that call the legacy code might actually ‘exploit’ the undesired behavior. Thus changing this behavior can cause unexpected problems in these external systems. This is why, when changing the behavior of (legacy) code, it is important to know the way it is currently used by other systems. For this, the generated anti-corruption layer can be used to confirm the new behavior will not conflict with the current usage of the legacy code. This process will be further discussed in Chapter 7.

6 Generating tests

The goal of this research is to assist engineers in refactoring legacy code. To gain confidence in the refactored code, the behavior of the module before refactoring can be compared to the behavior after refactoring. There are different possibilities when using a model for testing. One way to be considered is model-based testing [26] [27]. Model-based testing uses the model to check if a transition from state A to state B in the model also results in a transition from state A to state B in the System Under Test (SUT).

Another way to gain confidence, is to compare the learned model with a new one learned after refactoring the code and use model-checking techniques to see if this new model is IOCO [28] or even bi-similar [29] to the current model. IOCO, or Input Output COnformance, states that for every input in model A, model B must result in the same output. Bisimulation means that 2 systems are equal from a black box perspective, thus for each input in model A, model B must result in the same output and vice versa. Both methods assume the learned models are correct according to the SUT.

From an engineering and industrial perspective it might be more appealing to generate a set of test cases that can be used in existing test suites (like Gtest, JUnit or xUnit). Also, these generated tests can be reused multiple times for regression when code is being refactored and changed. Often coverage reports are generated from these test suites to indicate the quality of the code to engineers and management. Hence, we decided to generate test cases rather than using model-based testing.

Multiple ways of test generation are considered, in Section 6.1 we discuss some tools we looked into to generate tests sequences. These generated sequences still need to be translated to tests that can be included in, for example, the GTest tool suite ASML uses for testing their code. The tests reduce the risk to the system as a whole by proving the refactored component's behavior does comply with that of the model.

6.1 A review on tools for test case generation

There are a number of tools for testing using models. Since we do not wish to reinvent the wheel, we decided to use an existing tool to generate our test sequences. For the generation of test sequences we looked at external tools, specifically, the 15 tools described in [30], JUMBL (J Usage Model Building Library) [31], GraphWalker [32] and Hybrid Adaptive Distinguishing Sequences (HADS) [1].

In this section we will discuss the pros and cons of the tools we looked at, a brief summary can be found in Table 6. We checked the tools for the following criteria:

- **Publicly available:** We need a tool that is actually available to us.
- **Model compatible:** We need a tool that is compatible with our models and does not need additional information to generate tests.
- **Offline testing:** Offline testing means testing happens in 2 phases. First the tests are generated, then the tests are executed. Online testing generates tests on-the-go. We need a tool that can generate tests without executing them.
- **Export functionality:** Since we want to use the test sequences in our own tool the test tool needs to be able to export its test sequences.
- **Coverage based:** For our test purpose it is important that all transitions are covered. Thus the generated test sequences need to be coverage based.
- **Multi platform:** This criterion is not as important as the other ones, but since we work with both Linux and Windows systems a multi-platform tool is preferred.

Unfortunately, not all of these tools where suitable for this research because of the criteria mentioned above. That does not mean these tools are not potentially interesting for future work.

We also came up with the idea of a simulator (interpreter) for the model. Since a lot of tools do not save or export their tests but instead execute them immediately on the SUT, a future extension of this research could be creating a simulator that reflects the behavior of the model and log's all input's as a way of retrieving the test sequences from these tools. Unfortunately, we did not have time to implement and test this.

Tool	publicly available	Compatible with model	Offline testing	Export functionality	Coverage based	Multi-platform
Lutess	Yes	No	No	No	Yes	-
Lurette	Yes	No	No	No	Yes	-
GATeL	Yes	No	Yes	-	-	-
AutoFocus	Yes	No	Yes	Yes	Yes	-
Conformance kit	No	-	-	-	-	-
PHACT	No	-	-	-	-	-
TVEDA	Yes	Yes	Yes	-	Yes	-
Cooper	Yes	Yes	-	-	Yes	-
AsmL	Yes	Yes	Yes	Yes	Yes	No
TGV	Yes	-	Yes	Yes	Yes	-
TorX	Yes	Yes	No	No	Yes	Yes
STG	No	-	Yes	Yes	-	-
AGEDIS	No	No	Yes	Yes	Yes	-
TestComposer	Yes	Yes	Yes	Yes	Yes	Yes
AutoLink	Yes	Yes	Yes	Yes	Yes	-
Jumbl	Yes	Yes	Yes	Yes	Yes	Yes
GraphWalker	Yes	Yes	Yes	Yes	Yes	Yes
HADS	Yes	Yes	Yes	Yes	Yes	Yes

Table 6: Comparing test tools

In the subsections below, we briefly describe the different tools we looked at. For our proof of concept, we use the HADS tool to generate test sequences, since this seemed the most promising way. The HADS tool met all our criteria and the original developer was available if needed. This does not mean the other tools cannot be used as well, provided they are available and compatible with our models.

The generated test sequences need to be translated to a different format to be accepted by a test suite. We designed our tool in such a way we can extend it with more output formats and sequence generation methods in the future. For this research we limited ourselves to JUnit [33] and GTest (Section 6.2).

6.1.1 Lutess

Lutess [34] is a tool that uses Binary Decision Diagrams (BDD's) to generate its tests. One limitation is that it can only validate systems which have boolean inputs and outputs. Lutess expects a complete environment description in the Lustre language and only supports Online testing. For testing, Lutess randomly selects a new transition for every step, these steps can be guided using a condition/property or behavior pattern. Lutess will calculate the probabilities for the different transitions towards this condition and generate the sequences with the highest probabilities. Unfortunately, Lutess does not provide a way to generate tests based on coverage criteria, which is one of the cornerstones of our research.

6.1.2 Lurette

Lurette [35] is very similar to Lutess, but works with numerical inputs and outputs in contrast to Lutess boolean values. Besides Lustre, the Lurette tool also accepts the Lucky and Lutin language as input. Unfortunately for us, this does not change the underlying system and Lurette also does not support offline testing.

6.1.3 GATeL

GATeL [36] uses constraints for its test generation. These constraints consist of invariants and a test purpose. Test purposes can be expressed using path predicates. The tools translate the models into the CPL language and interpret (symbolically execute) this to generate test sequences. GATeL starts generating tests from the last state in a test sequence and tries to find a sequence towards this state such that it satisfies the invariant(s) and test purpose. This tool requires a complete specification of the SUT (in Lustre code), an environment description and a test objective.

6.1.4 AutoFocus

AutoFocus [37], like GATeL, uses constraints for its test generation and uses CPL to symbolically execute the model and generate test cases which it can save for later use. AutoFocus uses functional, structural and stochastic test specifications to guide the generation of test sequences. The advantage of AutoFocus is that it can generate test-cases conform given coverage criteria to the model.

6.1.5 Conformance Kit and PHACT

PHACT is a tool that builds on Conformance Kit. Unfortunately, both tools are not publicly available and thus we did not look into them any further.

6.1.6 TVEDA

TVEDA is an older tool (1995). It can generate TTCN-2 formatted test sequences. TVEDA's test approach is achieving a complete test coverage. It also uses symbolic execution to generate it's test cases. First, it tries to reach all states using a breath-first-search that is limited to 30.000 states. Then it (re)uses the previously discovered sequences to get as close as possible to all non-discovered transitions and executes another breath-first-search to reach them. This results in a set of test cases that uses all transitions. This tool might be interesting for future work.

6.1.7 Cooper

Cooper is meant for educational purposes and not for practical work. Cooper uses the implementation relation conf, this states that: for two processes B_1 and B_2, B_1 conforms to B_2 if and only if when B_2 deadlocks, B_1 should also deadlock. This means that B_1 may contain traces that are not possible in B_2. For our method, this would mean we allow either the model or the implementation to be more strict.

6.1.8 Asml

The AsmL Test Tool [38] generates a Finite State Machine from its model. Using the Chinese Postman algorithm to cover all branches in the FSM, the tool tries to generate test cases covering all transitions. Asml is written for the .NET framework and embedded in the .NET development environment. This makes the tool interesting for future work, but connecting it to our tool would have taken too much time for our current research.

6.1.9 TGV

TGV uses the IOCO implementation relation for its test generation. It is available as part of the Caesar Aldebaran Development Package so it should be relatively easy to integrate it into our tool. TGV uses IOLTS (Input Output Linear Transition Systems) equipped with an *accept* sink state and a *refuse* sink state to generate tests. If there is a state with an incomplete set of transitions TGV will complete it with self-loop transitions. TGV features a documented API to connect new ‘specification languages’ to. It has been tested in multiple industrial environments, making it interesting for our research.

6.1.10 TorX

Like TGV, TorX also uses the IOCO implementation relation for its test generation. It is a free tool that can be used for both test generation and execution. TorX features a “batch mode” and a “on-the-fly mode”. The batch mode can be used to generate tests without executing them. Unfortunately, this mode is only described in the TorX architecture and not actually implemented. In the on-the-fly mode, generating tests and executing them happen simultaneously. TorX can work with any model that can be expressed as an LTS, and interfaces between components are well documented so it should be possible to integrate it into our tool in the future. As an added bonus TorX is also able to work with non-deterministic systems.

6.1.11 Symbolic Test Generator (STG)

STG (Symbolic Test Generator) is inspired by TGV and TorX. It is able to generate executable test cases based on the IOCO principle but unfortunately, the STG tool is not publicly available.

6.1.12 AGEDIS

AGEDIS matches most of the list of criteria we are interested in. It is written in Java, Can generate tests for both state and transition coverage, it supports Java, C, and C++ for its test execution. But it is not publically available and cannot use a model alone to generate tests. It needs more information conform the UML standard, and thus it is not convenient for our research.

6.1.13 TestComposer

TestComposer is inspired by TVEDA and TGV and combines their strengths. It features both online and offline testing and uses a model with inputs and outputs to generate its tests. Making it suitable for our research. TestComposer can generate a set of tests given a coverage percentage and output it in multiple formats. Tests are automatically split into 3 parts: preamble, body, and postamble.

6.1.14 AutoLink

AutoLink [39], like TestComposer, is based on TVEDA and TGV and can automatically generate tests based on state space exploration. It uses an on-the-fly method based on that of TGV to generate TTCN formatted test suites from an SDL specification and can work with non-determinism.

6.1.15 JUMBL

JUMBL (J Usage Model Building Library) [31] supports construction and analysis of models, generation of test sequences, automated execution of tests and analysis of the test results. It uses Markov Chains [40] to determine the most useful test set based on the likeliness a path in the model is taken and generates it, but can also work with standard transition systems.

Unfortunately, these systems need to be finite, meaning they have a clear start and end state, while our learned models often are not. For test generation, JUMBL uses the Chinese Postman Algorithm [41] to construct test sequences for the model based on the probability of usage.

6.1.16 GraphWalker

GraphWalker [32] is a Model-Based testing tool that implements multiple algorithms to generate tests. These algorithm's keep generating tests until a given stop condition is reached. In our case edge coverage (transition coverage) is an interesting stop condition. GraphWalker features a Java library that can be embedded in other tools. It uses Extended Finite State Machines (EFSM) as input to generate tests and execute them. GraphWalker features an 'online testing' and an 'offline testing' mode. The online testing mode is an on-the-fly test generator that generates tests and executes them immediately. The offline testing mode stores the test-cases for later use. This means we can use GraphWalker to generate test sequences, store them and translate them into GTest tests using our own tool. Unfortunately, the documentation on GraphWalker is a bit lacking, so we decided to not take the risk of adopting the tool in our work.

6.1.17 Hybrid adaptive distinguishing sequences

Hybrid adaptive distinguishing sequences (HADS) [26] builds on the W-method introduced by Chow [24] and Vasilevskii [42]. The improved method reduces the number of test cases needed to confirm an implementation is correct to a given model.

The test sequences consist of 3 parts. First, we get to each state by taking the shortest prefix. Then all sequences of length k are taken from each state. Finally, for each of these sequences, we verify that the state we reach is the correct state by checking it's distinguishing sequence(s). This results in $PI^{\leq k}W$ test-sequences. Where P is the number of states, $I^{\leq k}$ the number of sequences of length k and W the amount of distinguishing sequences of each resulting state. For $k = 1$ this set of test-cases is such that if these tests succeed either the SUT is correct to the model, or the SUT consists of more states than the model. To check additional states we can increase k, but this increases the number of test cases exponentially.

Since the source code of this tool is available on GitLab [1], we can compile it for both Linux and Windows. The tool is compatible with the models generated by LearnLib and most importantly it is possible to use the tool in combination with a Java application and export the generated test sequences.

6.2 Google Test and Bullseye

To test its software, ASML used the Google Test (GTest) framework [43]. GTest is a unit-testing framework for C/C++ projects based on xUnit [44]. Unit-testing focuses on testing the functions of the code rather than testing the functionalities of the code (integration testing). One of the most common ways of testing is by checking for equality. Tests can be defined as the example test below:

```
TEST (SquareRootTest, PositiveNos) {
    EXPECT_EQ (18.0, square-root(324.0));
    EXPECT_NE (45.4, square-root(645.16));
    EXPECT_EQ (50.3321, square-root(2533.310224));
}
```

Where SquareRootTest is the hierarchy of the test, PositiveNos the name of the test and the EXPECT_EQ and EXPECT_NE functions are the test steps. EXPECT_EQ checks if the values given in its parameters are equal, and EXPECT_NE checks if they are not equal.

Developers can compose a set of tests like the one above and use GTest to run them and view the results. GTest features a mechanism for mocking classes called Google Mock (GMock).

This feature can be used to replace a class with a dummy instance that mocks its behavior. We experimented with this for stubbing our dependencies but found it easier to create the stubs mentioned in Chapter 4. Another feature of the GTest framework is monitoring function calls that happen in the deeper layers of the code. This way it is possible to check how many times a given function is called by the SUT when a test is executed. Unfortunately, this functionality needs information that is not contained in our models, thus we cannot generate useful tests for them.

Additionally, ASML used the Bullseye Code Coverage Analyzer [45] to extract test coverage information at run-time. The Bullseye tool monitors the usage of the code and generates a report about the function and decision coverage. Running Bullseye while running the generated tests gives us the coverage metrics of these tests.

6.3 Chocolate vending machine

For our chocolate vending machine we generated JUnit tests [33] since the application is written in Java and GTest only works for C/C++. The JUnit test format is relatively easy so it seemed like a good first step. For the model described in Section 5.1.2 the tool generated 38 test sequences. Part of these test sequences is included in Appendix C.

The example below shows there are still some limitations to the way we generate unit-tests from the model. The model lacks some critical knowledge of what is needed to test the actual SUT because the learner only communicated with an adapter. Supplying data parameters to the functions, for example, is done by the adapter since LearnLib does not support this yet. Therefore the model has no knowledge of the parameters needed for calling the actual functions. Also, the model had no knowledge of the reset function or sequence of the actual SUT. This is fixed by using the reset command used by the learner, but this is not necessarily a valid function call since, as in our industrial case, this might be a function created in the adapter that is not present in the actual application.

```
public void test6(){
    reset();
    assertEquals(5ct(),"OK");
    assertEquals(twix(),"NOK");
    assertEquals(mars(),"NOK");
    assertEquals(5ct(),"OK");
    assertEquals(mars(),"OK");
}
```

The generated test files can be used as a template for the final tests. Most of the information needed to execute the tests might not be available in the test tool but has already been used for constructing the adapter. Thus the reset function and function parameters, for example, can be copied from there. For our experiments simple find and replace operations sufficed to gain an operational test suite.

The generated test cover all transitions, but this does not mean they also cover all functions of the SUT. Internal functions are typically left out of consideration since they were not called by the model learner. In our case, the messages sent to the adapter did not match the functions called in the SUT at all, thus we had to modify the SUT to support these unit tests by subtracting the logic in the main function that received and handled the messages to separate functions we could call from the generated tests.

From this experiment, we can conclude that, while there are still limitations, it is possible to generate test cases from a model to test the behavior of a system using an existing test suite like JUnit. This way model learning can be an addition to a company's current development environment. The generated test cases can show an implementation is correct to the model and thus, providing the model is complete and correct to the implementation, show behavior preservation when refactoring the software.

6.4 Industrial case

Since ASML uses the GTest tool suite (Section 6.2) for testing its more recent software modules, we implemented a way to translate test sequences to the GTest format. We chose to generate a test-set similar to the JUnit tests shown in Chapter 12 but leave room for engineers to add additional functionalities by hand if they feel like it adds value to the test.

This part is removed for confidentiality and can be found in the original version of this document.

Using the proposed model, 60 test cases are generated. Like in our example application, we face limitation concerning function parameters. To be able to execute these tests, we added the parameters used by our adapter since choosing a different set of data parameters might result in a different behavior, causing the tests to fail unexpectedly.

Using the Bullseye Code Coverage Analyzer [45] we can generate a coverage report for the generated tests.

This part is removed for confidentiality and can be found in the original version of this document.

The assumption our model covers the entire behavior of the code is also based on the assumption the model learning algorithm learns the entire behavior. But like we mentioned in Chapter 2: if, for example, a system responds identically the first 100 times a button is pressed but responds differently the 101st time, most model learning algorithms will already have stopped 'exploring' that button and not discover this change in behavior. If this behavior is intended, this can be added by using it as a counterexample for the learned. This behavior will then be covered in the model and a test sequence will be generated for it. If this behavior is not intended but does occur, it will most likely not be covered in the model and thus not be discovered by the generated tests.

7 Anti-corruption layer

The generated tests described in Chapter 6 increase the engineer's confidence in the refactored module's behavior and are meant to protect the rest of the system while refactoring the module. But refactoring often does not limit itself to one module. If multiple parts of the system are being refactored, this might accidentally lead to 'illegal' action sequences being called, resulting in crashing modules. To protect the SUL from such illegal actions at run-time we introduce an anti-corruption layer (ACL).

For our ACL we looked at the Shield Synthesis [2] approach (Section 7.2.4) and the way ASML currently regulates the connection between different sub-systems using so-called *armors*. First, we give a more formal definition for the ACL. Next, Section 7.2 shows four possible types of the ACL. In Section 7.3 we show a proof of concept using our chocolate vending machine example, and finally in Section 7.4 we discuss our industrial case at ASML.

7.1 What is an anti-corruption layer (ACL)?

An ACL is a wrapper for a software module that monitors and/or enforces certain behavior depending on its implementation, protecting the module at run-time. When an external client triggers a sequence that is not allowed, the ACL will jump in and act on it.

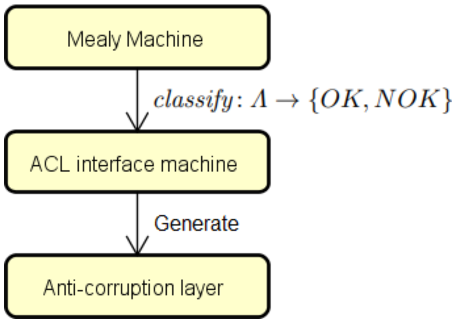


Figure 17: Generating an ACL from a Mealy Machine

To generate an ACL from a behavioral model (Mealy Machine) $M = (S, s_0, \Sigma, \Lambda, T, G)$, output set Λ is split into two subsets: $\Lambda_{ok} \subseteq \Lambda$ denotes the set of all accepted outputs, and $\Lambda_{nok} \subseteq \Lambda$ denotes the set of all not-accepted outputs. This last category contains any output that is the result of erroneous or undesired behavior. To determine if an output ψ is accepted or not we define the function *classify* that maps any output $\psi \in \Lambda_{ok}$ to OK, and any output $\psi \in \Lambda_{nok}$ to NOK.

Using this function we can convert a Mealy Machine to a specific kind of Mealy Machine called an *ACL interface machine* $M' = (S, s_0, \Sigma, \{OK, NOK\}, T, classify \circ G)$ that describes what actions are and are not accepted. To protect the system from 'illegal' behavior the ACL should take action if, and only if, a sequence is triggered results in an output $\psi \notin \Lambda_{ok}$. Note that this definition not only protects the software module from any not-accepted output $\psi \in \Lambda_{nok}$, but also from unexpected inputs $\sigma \notin \Sigma$.

By example: If we consider the behavior in Figure 18 where *S0* is the initial state. The ACL acts for any sequence φ if and only if φ does not result in output OK. Meaning input sequences *abaa* and *abba* are not accepted and thus the ACL will be triggered, but also input c is not recognized and will trigger an action from the ACL.

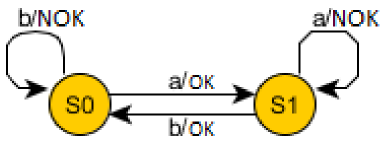


Figure 18: ACL interface machine for regular expression $(ab)^*$

7.2 Types of anti-corruption layers

For our research we implemented three types of anti-corruption layers, additionally we looked at Shield Synthesis [2] for a possible fourth implementation. These types all use our abstract definition as a starting point, but have their own implementation for the action taken by the anti-corruption layer and can be used in different stages of the refactoring process.

7.2.1 ACL: Observer

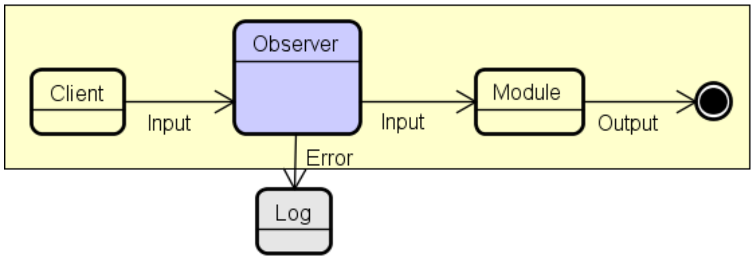


Figure 19: Abstract activity diagram: Observer

The first type is called the *Observer* (Figure 19). Via this type of ACL, it is possible to identify clients that do not adhere to the behavior described in the ACL interface machine. The Observer forms a layer between the client and the module and passes all calls from the client to the module. The Observer has an internal representation of the ACL interface machine and keeps track of the current state of the module using the observed inputs. As shown in Figure 20, when an input is passed that is not accepted by the ACL interface machine, the Observer logs this input to an external file. When an engineer is not 100% sure a model describes the correct behavior, it might be preferable to first implement this weaker form of ACL to detect potential violations without affecting the system. If the generated file shows behavior violations occur, the engineer can analyze whether these violations are caused by faults in the model or in the implementation.

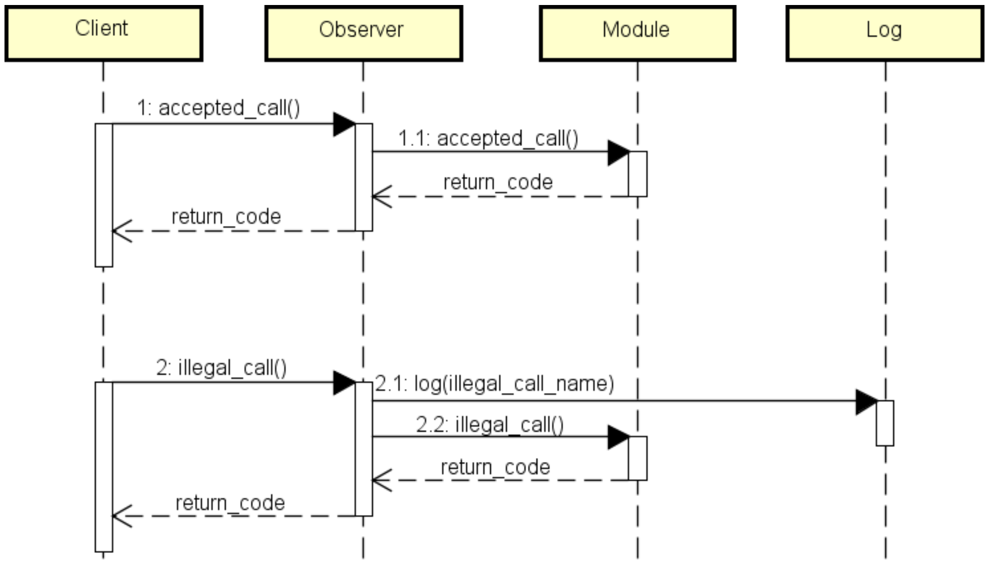


Figure 20: Abstract sequence diagram: Observer

7.2.2 ACL: Armor

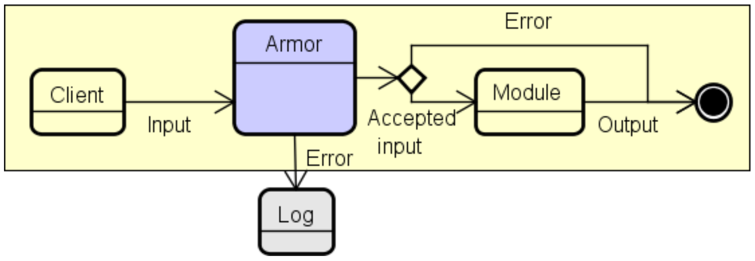


Figure 21: Abstract activity diagram: Armor

The second type is called an *Armor* (Figure 21). Like the previously explained Observer, the Armor uses an internal representation of the ACL interface machine to keep track of the current state of the module. By actively blocking not-accepted inputs, as can be seen in Figure 22, the Armor actively protects the module. Enforcing the behavior described in the ACL interface machine can prevent systems from crashing or, when considering embedded systems, causing physical damage. The Armor also logs not-accepted inputs since in practice it turned out to be useful to log behavior violations to allow the tracking of potential errors caused by refactoring.

An important effect of the Armor to consider is that, when a client calls a not-accepted input for its side effects instead of for the 'failing' part of the operation, this side effect will also be blocked. If blocking these side effects is the desired effect of the Armor, such clients should be modified.

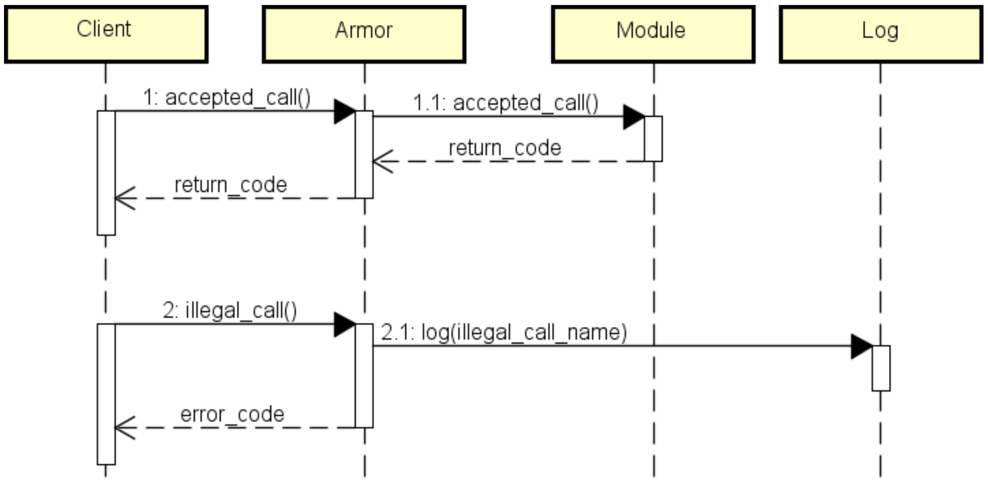


Figure 22: Abstract sequence diagram: Armor

7.2.3 ACL: Enforcer

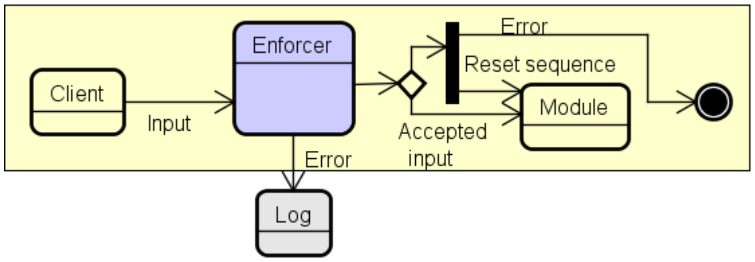


Figure 23: Abstract activity diagram: Enforcer

A more sophisticated way of handling behavior violations might be the *Enforcer* (Figure 23). When this type encounters a behavior violation, it will not only log and block it, but also reset the module. Like the Armor, the Enforcer blocks not-accepted inputs and returns an error code, but additionally, it triggers a reset sequence on the module to bring both the module and the Enforcer back to their initial state.

The reset sequence can be the shortest sequence in the ACL interface machine, assuming such a sequence exists for every state with not-accepted inputs. These sequences can be obtained by applying breadth-first search between the current state and the initial state, which can either be calculated at run-time or when generating the Enforcer. In the latter case, these sequences will be hard-coded in the implementation.

If there is a clear and safe reset sequence for the code, as used during the model learning process, it is also possible to use this as the reset sequence triggered by the Enforcer.

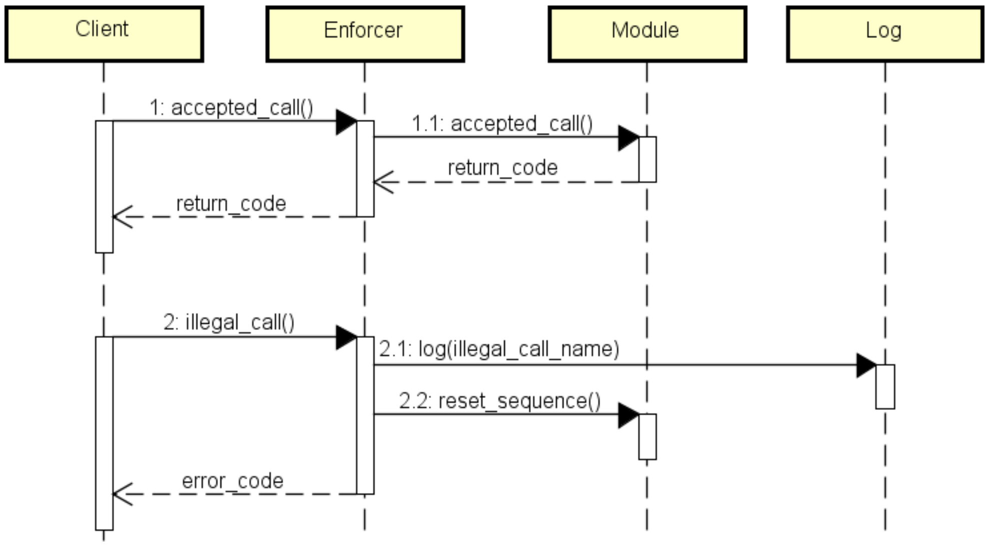


Figure 24: Abstract sequence diagram: Enforcer

7.2.4 Shield Synthesis

Another possible type of the anti-corruption layers was proposed in the Shield Synthesis approach [2]. In [2] a pattern similar to the anti-corruption layer, called a *Shield*, is 'generated' from safety-properties (*"No bad things happen"*) and liveness-properties (*"A good thing will eventually happen"*). However, the way the Shield is described would also make it possible to generate it using our method.

The main difference between the Shield and the *Observer*, *Armor* and *Enforcer* mentioned before is that Shields are positioned 'after' the module, whereas the *Observer*, *Armor* and *Enforcer* are positioned 'before' the module (see Figure 25). The Shield is described to modify the output of a module if this output is not accepted. However, experts in ASML preferred reporting and preventing an error to the client that caused it, like our Armor does, rather than performing a corrective action like the Shield does. By moving the perspective of the Shield from the module to the client, the client's output, and thus the input of the module, can be corrected.

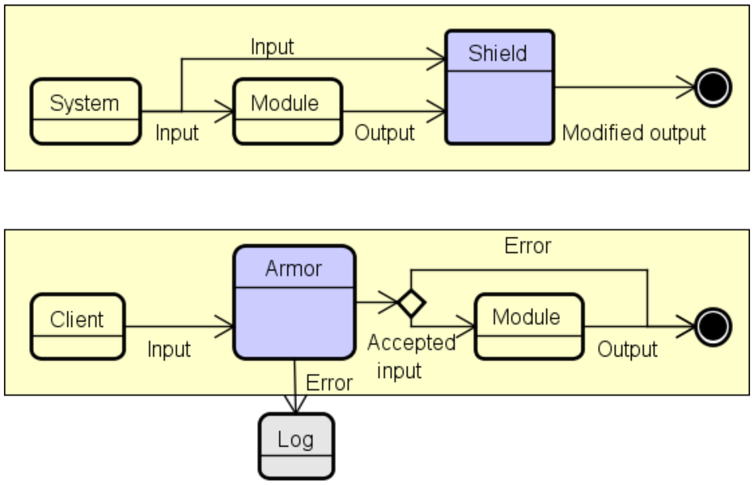


Figure 25: Shield VS Armor

When experimenting with the Shield implementation, we encountered an important limitation. In [2] the author explains how the Shield works by using a traffic light as an example. As shown in Figure 25 the Shield modifies illegal output's to represent a pseudo-random legal value. In the traffic light example, it should not be possible to have multiple traffic lights showing green. If the traffic-light-controller gives this output nonetheless, the Shield will modify it to have at most one green light.
In this example, it makes perfect sense, but if we go back to our chocolate vending machine example and apply a Shield here the following situation might occur:

Suppose that a Snicker which costs 25 cents is requested, but only 20 cents is inserted. Normally the machine should refuse this request, but if for some reason it wants to give the Snicker anyway, the Shield will be triggered and it changes the Snicker output into an output that is actually allowed in this state: for example, a Twix or a Mars. The problem with this is that a Twix or a Mars was not requested by the user and if the system would have produced a warning, the user might have inserted the remaining 5 cents or canceled the purchase.

More dangerous even is that in Embedded Systems like our industrial case modifying values can give the software a wrong picture of the 'real world'. For example, when measuring a wafer in the TWINSCAN machines a sensor might be faulty, but the software would not know that since it only receives correct looking values because the Shield has modified them.

From these hypothetical situations, we can conclude the Shield is useful in specific situations but can cause problems in others. This all of-course strongly depends on how the Shield is applied and one might argue the same goes for blocking undesired behavior or returning to the initial state as occurs in our Armor and Enforcer approach. Thus, when deploying an anti-corruption layer, one should beware of the possible effects this might have on the system.

When experimenting with generating a Shield, we tried to implement the corrective action in such a way it finds the nearest state where the requested input is accepted and then handles the requested input. In Figure 26 this is called the *legalizing sequence*. For each not-accepted input this legalizing sequence is added as a prefix. Assuming such a sequence exists, finding the legalizing sequence can be done using breadth-first search.

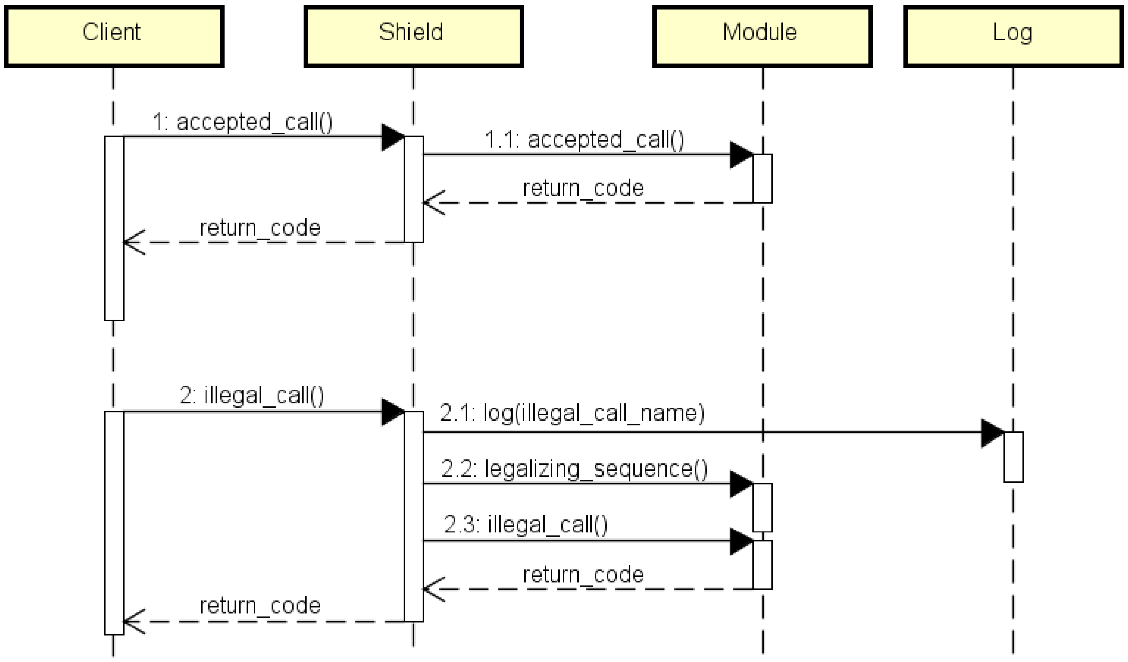


Figure 26: Abstract sequence diagram: Shield

The experiment revealed a problem with this implementation. Sometimes a total reset sequence appears to be the shortest path. If this is the case, a lot of previously gathered data might be lost. A possible solution to this problem would be adding weights to the transitions or even the states. Unfortunately, the current model does not contain enough information to implement this, but it could be interesting for future work.

From our experiment with the Shield, we conclude that there are still some limitations that prevent us from generating smart Shields from our models. We addressed some possible solutions to the current problems, but due to the limitation of time, this will be a subject for future work. We are not stating a Shield is a bad idea. In fact, we do see the potential in situations like the traffic light example and since the Shield fits our anti-corruption layer definition it is possible to generate Shields from an ACL interface machine, but like with the other types of the ACL we advise caution when applying it on a system.

7.3 Chocolate vending machine

As stated before, an ACL checks and/or enforces certain behavior. The Armor implementation for example, will prevent not-accepted actions by blocking them and returning an error code, it will also log the function call to enable the tracking of errors.

Using the previously learned model, an Armor can automatically be generated for our example application. As the outputs in the model are already represented by OK and NOK, converting the model to an ACL interface system is unnecessary. As a proof-of-concept, an Armor that contains all functions that are used to learn the model is generated. Each function checks if it is allowed in the current state and updates the current state accordingly. The function returns an error code (integer) if it is not allowed in the current state, otherwise the function will call the accompanying function in the original module. 'Illegal' inputs (inputs that are not present in the input language) are automatically protected by the compiler. Since the inputs represent function calls, and functions that do not exist simply cannot be called. The code snippet below shows the generated function in the Armor corresponding to the *twix* input of the vending machine:

```
int chocolateVendor_armor_twix(){
    switch(chocolateVendor_armor_state){
        case 0:
            chocolateVendor_armor_state = 0;
            System.out.println(String.format("Action \"twix\" not allowed in state %d",
                  chocolateVendor_armor_state));
            return -1;
        case 1:
            chocolateVendor_armor_state = 1;
            System.out.println(String.format("Action \"twix\" not allowed in state %d",
                  chocolateVendor_armor_state));
            return -1;
        case 2:
            chocolateVendor_armor_state = 2;
            System.out.println(String.format("Action \"twix\" not allowed in state %d",
                  chocolateVendor_armor_state));
            return -1;
        case 3:
            chocolateVendor_armor_state = 2;
            return twix();
        case 4:
            chocolateVendor_armor_state = 5;
            return twix();
        case 5:
            chocolateVendor_armor_state = 0;
            return twix();
        case 6:
            chocolateVendor_armor_state = 1;
            return twix();
        case 7:
            chocolateVendor_armor_state = 3;
            return twix();
    }
    return -1;
}
```

The *chocolateVendor_armor_state* variable is used to keep track of the state as described in the model. When applying the Armor, the original calls to the module need to be redirected to the ACL. In the chocolate vending machine example this was a bit more complicated, since the commands that were used to learn a model were not actual functions in the code. But if the original function calls are used as input this should, in object-oriented languages, be little more than replacing the original object for the generated Armor.

After implementing the ACL in the chocolate vending machine, the tests generated in Chapter 6 where executed. The tests confirmed the generated ACL was correct to the previously learned model. Additionally, a new model was learned including the Armor. This new model was compared to the previously learned model (Figure 11), confirming the behavior was indeed identical. By checking the logs (in this case the console) it was also confirmed that all illegal actions were blocked correctly.

7.4 Industrial case

For our industrial case at ASML, we generated the Observer and Armor in C++. The Observer is used to check if clients do not violate the desired behavior settled on by the domain experts.

To confirm the functioning of our anti-corruption layer, we first generated an Armor (Section 7.2.2) from the learned model. Since the model is fairly small, it was possible to manually check it and doing so we confirmed its correctness to the model.

This part is removed for confidentiality and can be found in the original version of this document.

After applying the Armor some regression tests were executed on the simulator (DevBench). These tests detected no errors, which was in line with our expectation since the same tests were used to generate the system traces that were used to guide our learner.

This part is removed for confidentiality and can be found in the original version of this document.

In Chapter 5, system traces were used to learn the model. This revealed the traces, and thus the regression tests, only cover a subset of the learned model. To test the entire Armor we also ran the generated unit-tests. The Armor not only blocks, but also logs behavior violations. By comparing the log file to the expected outputs of the unit-tests the Armor's behavior could be confirmed, assuming the tests are correct to the model. Additionally, the completeness of the generated unit tests could be confirmed by generating a coverage report for the Armor.

This part is removed for confidentiality and can be found in the original version of this document.

the coverage report shows a 100% function coverage, but only a 79% condition/decision coverage. Upon closer inspection, this was caused by the exception handling in our Armor. For each function call, the Armor checks the state it is currently in. When this state is unknown for the function, which should not be possible with a correct model, it will return an error. This means one condition per function should not be triggered if the model is correct to the implementation and thus will not be covered by the generated tests.

The Armor also checks whether the log file could be opened before writing to it, this is to prevent system crashes. In our experiments, this problem never occurred. Thus, when not taking these exceptions into account, we can state we reach 100% test coverage for our Armor.

8 Refactoring the code

This chapter is removed for confidentiality and can be found in the original version of this document.

9 Conclusions

To conclude this research, we look back at the original research questions. In this thesis a method to assist engineers in refactoring legacy code is proposed. This method used a combination of model learning and test/code generation to reduce the risk of refactoring. To answer the first (main) question, *Can we provide engineers with a method to refactor legacy code based on model learning?*, we addressed the original research questions introduced in Section 1.5 as follows:

RQ2) Is model learning mature enough to be applied in an industrial setting?
In Section 5.1 we explain the principle of model learning using the vending machine example introduced in Section 3.4 and the L* algorithm. Before a model can be learned the System Under Learning (SUL) should be isolated from the rest of the software. Chapter 4 shows how we did this for our industrial case. The scope of the software that is isolated depends on what and how you wish to refactor. For refactoring an entire component, it might be useful to isolate more than a single module. In the case of the chocolate vending machine, we even learn the entire application. Engineers should keep in mind that larger scopes might result in larger models, thus a trade-off between the clarity of their models and the number of models that need to be learned must be made.
Using LearnLib, the L* algorithm and system traces extracted from a simulator of ASML's TWINSCAN machine we were able to learn a model of the legacy code of our industrial case. Chapter 5 discusses the current strengths and limitations of our learning method.
Since the learned behavior does not necessarily reflect the desired behavior of the code, engineers can refine the learned model. There are multiple tools that enable engineers to do so, and to support as much of them as possible we implemented multiple import and export formats for the models in our tool.
As [13], we also used a specific industrial case. From the experience gained during this research we conclude that, while model learning is relatively young, it is mature enough to be used for similar industrial applications. But, in the case of LearnLib, there are still some major limitations in, for example, the use of function parameters that strongly limit scalability.

RQ3) When model learning is applied for refactoring industrial software, what are the benefits and the limitations?
For our research, we used LearnLib to learn a behavioral model of our industrial case. By doing so we discovered some very useful benefits to the learned models, but unfortunately, there are still limitations to the methods we used. Table 7 briefly summarizes our findings, for a detailed description of all strengths and weaknesses we refer to Chapter 5.

Strengths	**Limitations**
Revealing potential flaws (bugs)	The SUL needs to be isolated from the system
Models form a clear representation of behavior	Not guaranteed to learn the complete behavior
Only a limited understanding of the code is needed	Models can grow very big, making manual modification nearly impossible
Combined with system traces the current behavior of the system can be shown	Models can be represented in many different formats
Learned models can be modified to assist engineers in fixing potential flaws when refactoring	LearnLib does not work with function parameters (yet)

Table 7: The strengths and limitations of model learning

As explained in Chapter 4, LearnLib still requires some manual labor before it can learn a model. Also, it is not guaranteed to learn the complete behavior of the legacy code. But using system traces we can at least make sure it contains the behavior that is currently used at run-time.

A very strong benefit of learning models of legacy systems, is that it may reveal flaws (bugs) in the behavior of the system. If such flaws are discovered, an engineer can modify the learned model to represent the desired behavior. This way potential problems can be solved while refactoring the code. Modifying the behavior of the code does, however, pose the risk of introducing new bugs. Using an anti-corruption layer, we gain confidence that our desired model does not conflict with the current usage of the system.

Another strong feature of the learned model is that it visualizes the behavior of the system in a readable way. Marking the model with system traces gives a clear view of the current behavior of the system. However, we know from previous experience these models can grow quite big and become very unreadable.

Model learning is a black-box method for learning a system, this means the learned model represents the behavior from an external perspective. It also means that, because the model learner does not look at the internal code of the system it is very hard to learn the complete behavior of a system. When for example a system responds identical to the first 100 times an action is called but changes it's behavior the 101st time, most model learning algorithms will already have stopped learning and not discover this behavior. Counterexamples can be used to guide the model learner into learning this behavior, but this requires knowledge of the system and that is usually scarce when considering legacy systems.

The biggest inconvenience we experienced while learning our industrial case, is that LearnLib does not support function parameters yet. We solved this problem by using multiple messages for some actions. For example, function *foo* contains a boolean parameter. This can be resolved by using two different messages: *foo_true* and *foo_false*.

RQ4) Based on experience gained in the industry, can we provide engineers with general guidelines for software refactoring?
Section 1.2 refers to 6 activities needed for software refactoring, namely:

- Identify where the software should be refactored.
- Determine which refactoring(s) should be applied to the identified places.
- Guarantee that the applied refactoring preserves behavior.
- Apply the refactoring.
- Assess the effect of the refactoring on quality characteristics of the software (e.g., complexity, understandability, maintainability) or the process (e.g., productivity, cost, effort).
- Maintain the consistency between the refactored program code and other software artifacts (such as documentation, design documents, requirements specifications, tests, etc.).

To help identify where software should be refactored, we opted to determine the state of the software using the following 11 guidelines from [4] and [5] as discussed in Section 1.1:

1. Write short units of code (functions should be <= 15 lines).
2. Write simple units of code.
3. Write code once (no duplicate code).
4. Keep unit interfaces small.
5. Separate concerns in modules.
6. Couple architecture components loosely.
7. Keep architecture components balanced.
8. Keep your codebase small.
9. Automate tests.
10. Write clean code.
11. Functions take at most 3 parameters.

This can be (partially) automated by using static analysis tools like CodeSonar [10] or Better-CodeHub [11].

Furthermore, we propose a method for refactoring using model learning, test and code generation. This method assists in gaining a better understanding of the to be refactored code and increases the engineers confidence in the refactored code. Chapter 2 shows all the steps needed for our proposed method. Section 1.2 explains there are 3 types of refactoring, for ASML refactoring means a combination of redesigning and restructuring code in such a way that it is easier to understand and maintain. Our method however, is applicable for any of these types.

RQ5) Can we automatically generate tests from a learned model to assist engineers in refactoring legacy code in a test driven way? If so, what is the most convenient test framework to use?
Chapter 6 shows us there are multiple possible ways of testing when using models. For our research, we chose to generate tests that can be implemented in the company's current test suite. The generated tests are used to increase the engineer's confidence in the behavior preservation of their refactored code. By generating unit-tests we enable the possibility of test-driven development when refactoring the code, this means that while refactoring commonly used unit testing frameworks (like JUnit and GTest) can be used to confirm the code's behavior.
To find a suitable tool to generate test sequences multiple test generation tools were studied. For our proof of concept, we use the HADS [1] tool to generate test sequences that cover all transitions of the model. These test sequences are then translated to the JUnit and GTest formats. Using the Bullseye Code Coverage Analyzer, we generated metrics to show the function and statement coverage of the generated tests. While they did not reach 100% we are confident it is possible to increase these statistics by refactoring the code and improving our learner to handle more function parameters.

RQ6) Can we use models to guarantee behavior preservation when deploying the refactored code?
We explained in ***RQ3*** that the learned model might expose flaws in the current behavior. Thus for this research, we opted to preserve the desired behavior rather than that of the legacy system. ***RQ5*** dealt with generating tests to increase the engineer's confidence in the behavior of the refactored code. Using the HADS [1] tool we generated a set of tests that guarantee either the system is correct according to the model, or the system contains more states. Unfortunately, this does not completely guarantee behavior preservation from the refactored code. Also, it proved to be complicated to learn complete and correct models from big systems like our industrial case because of the current limitations of the model learning tool LearnLib. Thus it might prove useful to look into other tools for model learning.

To protect the module from *illegal* action sequences at run-time we also introduced an anti-corruption layer in Chapter 7. This anti-corruption layer can be generated from the model and forms a protective wrapper around the learned code. We showed three types of the anti-corruption layer:
The *Observer* is a type of anti-corruption layer that logs behavior violations and can be used to confirm the system's behavior when modifying a model.
The *Armor* is a type of anti-corruption layer that both logs and prevents behavior violations by throwing an error instead of forwarding the call to the module.
The *Enforcer* is a type of anti-corruption layer that logs behavior violations, throws an error instead of forwarding the call to the module and brings the module back to the initial state.
Additionally, we also experimented with a *Shield*. This type corrects the given input by adding a prefix to bring the system to a state where the input is allowed. Unfortunately, there are still some limitations in the current models that cause problems when using the Shield in practice.

The techniques we use, increase an engineers confidence in the behavior of the refactored code, but they are not enough to completely guarantee behavior preservation. As stated before, the generated tests will show the software contains *at least* the behavior described in the model, but they are not able to detect whether the software contains more states. The anti-corruption layers can prevent *illegal* behavior from external clients, but this also assumes the model is correct for the implementation of the software.

By answering the questions above we can answer the main question of this thesis: ***Can we provide engineers with a method to refactor legacy code based on model learning?*** Combining the answers to the research questions above we came up with a method to refactor code by combining model learning, test and code generation.

To learn a model, the to be refactored part of the system needs to be isolated from the rest of the system. Using this isolated system multiple model learning techniques can be applied to extract a model from the system. For this research, we used the L* and TTT algorithms that are implemented in LearnLib. After a behavioral model is learned from the system, potential flaws in it can be corrected. From this corrected model tests and an anti-corruption layer can be generated. The generated tests can be used to refactor the code in a test-driven way. By using the 10 guidelines discussed in ***RQ4*** the quality, and therefore the quality improvement, of the code can be measured. This guides engineers in developing maintainable code and increases their confidence in the behavior of the refactored code. After refactoring, the generated anti-corruption layer can be applied to protect the refactored code from 'illegal' behavior. The anti-corruption layer can also be used to check if the current system does not violate the desired behavior.

A detailed description of this method can be found in Chapter 2. Using our method we aim at helping companies reduce the risks posed by refactoring, increase their test coverage and even solve potential flaws in their systems. To prove that the method works we applied it to our industrial case in Chapter 8.

9.1 Recommendations and Future work

This research was meant to explore a possible method for refactoring software using model learning. During the limited time-frame of this research, we had to make choices on which ideas to execute. In this section, we will discuss the most promising ideas that unfortunately where out of our scope.

9.1.1 Using strongly connected components to detect cyclic dependencies

ASML is in possession of a tool that generates dependency graphs of existing code. We used this tool to identify cyclic dependencies when isolating the module. Identifying strongly connected components in these graphs can prove useful to assist engineers in isolating modules from this code since these strongly connected components reveal the cyclic connections to other modules that need to be severed. We recommend looking into extending this tool with this functionality, in order to simplify the isolation of the SUL in the future.

9.1.2 Static code analysis

Refactoring software usually means working with a white-box SUL, though the model learning techniques used in this research are created for black-box learning. This means we learned the code's behavior from an external interface (client perspective). It might prove useful to look into static code analysis tools like CPAchecker[25] (previously known as BLAST[46]) to learn models or sequences that can be combined with the model learned by LearnLib or used to provide counterexamples to improve the learned model.

9.1.3 Consider more variables when learning models

During this research, the behavior of the system consisted mainly of the order of system calls. However, the real behavior is also be affected by other variables (like function parameters). Applying our techniques in Chapter 8 showed us that without this the generated tests do not cover all conditions/decisions and cannot guarantee behavior preservation. LearnLib only works with string representations of inputs and outputs, so it is not possible to work with function parameters or variables unless these are being 'mimicked' by using multiple messages to call functions with different parameters. To improve this work we recommend looking into different behavior aspects like function parameters or variables and how to influence them to learn their impact on the behavior more effectively.

9.1.4 Potential problems with big models

When refactoring entire systems, models can become quite big. The models we used for our research where very small, thus we did not encounter problems concerning their size. However, as stated before, the scope of the SUL depends on the project. A bigger SUL, or a SUL containing more behavior, can result in very big models. If this is the case modifying these models by hand poses problems. A possible solution might be to split the model into multiple smaller models. In [47] a method for splitting models is described. Additionally, we planned to look into ways to modify models in a semi-automatic way but due to time limitations, we had to exclude this idea.

Semi-automatically modifying models can be done using, for example, predicate logic. For example: if we would give our tool a predicate stating that only one Twix is allowed at a time, the tool should be able to modify the model in such a way that this is the case. Model checking algorithms can also be used to check the model for these predicates, in this way certain behavior (like deadlocks or specific sequences) are easier to detect. There are a lot of tools available to check and/or modify models.

9.1.5 Tools for test generation

In Section 6.1, multiple interesting tools for test generation are discussed. For the sake of time not all of these tools could be tested, in the future it might profit to dive deeper into these and other tools.

Some of the investigated tools do not allow for exporting test sequences, but only support online testing (generate tests while executing them). A method for extracting test sequences from these tools could be creating a simulator that reflects the behavior of the model and logs all inputs as a way of retrieving the test sequences from these tools. Extending our prototype tool with this functionality allows us to embed more test generation tools.

9.1.6 Add weights to the model

For this research we used models as generated by LearnLib, these models considered the weight of every operation (transition) as equal, while in practice this might not be the case. It might be profitable to consider the running-time of each operation while generating the shortest routes between states. This way we might be able to reduce the running time of the test sequences. This weight can then also be used to generate more effective anti-corruption layers like the *Enforcer* or the *Shield*.

The Shield Synthesis approach discussed in [2] showed us how to create Shields from liveness and safety properties. In Section 7.2.4, we experimented with generating a Shield from a model. Instead of a 'random' output, we opted to trigger a sequence that brings us in a state where the requested action is allowed. If, for example, there is a lot started and we want to start another lot before finishing the current one. A sequence of actions can be called to first finish the current lot, and then start the new one. Our experiment however, showed that sometimes this means resetting the system and thus losing the current progress. However, using weighted transitions we can guide the Shield in which action to take. While this is still tricky, looking into safer and more efficient ways to generate this kind of anti-corruption layers might prove worthwhile.

10 Appendix A: Vending Machine

This appendix contains the source code of the vending machine example used to explain the model learner. The example is based on a JavaScript application used by Jan Tretmans for his Testing Techniques lectures.

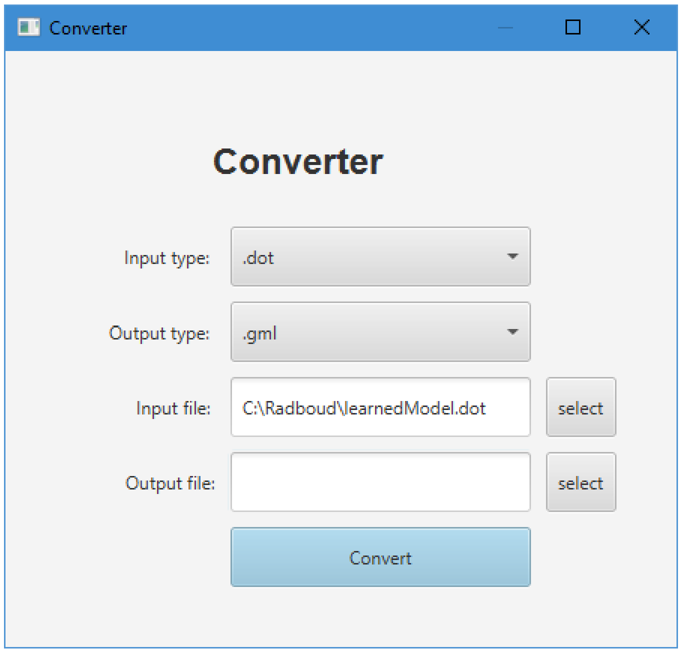
```java
public class ChocolateVendorExample {
  private static final boolean verbose = false;
  private static final int port = 1026;
  private static int money = 0;
  private static HashMap<String, Integer> chocs = new HashMap<>();

  public static void main(String[] args) {
    chocs.put("mars", 10);
    chocs.put("twix", 15);
    chocs.put("snickers", 25);

    int emptyMsgAmount = 0;

    // Get commands from socket.
    try {
      log("Setting up communication...");
      Communicator com = new Communicator("localhost", port);
      log("System connected!");
      while(emptyMsgAmount < 3){
        //Handle incoming message
        String msg = com.receiveMessage().replaceAll(System.lineSeparator(), "");
        String log = "received: "+msg+" |pre-amount: "+money;

        boolean success = false;
        if(chocs.containsKey(msg)){
          success = getChoc(msg);
        }else if(msg.equals("reset")){
          reset();
          success = true;
        }else if(msg.equals("5ct")){
          success = addMoney(5);
        }else if(msg.equals("10ct")){
          success = addMoney(10);
        }else if(msg.equals("")){
          emptyMsgAmount += 1;
        }

        //Send response
        if(success && !msg.equals("") && !msg.equals("reset")){
          log(log+" |post-amount: "+money+" |sending: OK");
          com.sendMessage("OK");
        }else if(!msg.equals("") && !msg.equals("reset")){
          log(log+" |post-amount: "+money+" |sending: NOK");
          com.sendMessage("NOK");
        }
        TimeUnit.MILLISECONDS.sleep(100);
      }
      log("Not receiving input anymore.");
    } catch (IOException | InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("THE END.");
  }

  private static boolean addMoney(int x){
    if(money <= 25){
      money += x;
      return true;
    }
    return false;
  }

  private static boolean subtractMoney(int x){
    if(money < x)
      return false;
    money -= x;
    return true;
  }

  private static void reset(){
      money = 0;
      log("resetting");
   }

   private static boolean getChoc(String choc){
      log("buying "+choc);
      if(chocs.containsKey(choc)){
         return subtractMoney(chocs.get(choc));
      }
      return false;
   }

   private static void log(String msg){
      if(verbose)
         System.out.println(msg);
   }
}
```

11 Appendix B: Model Converters

For this research, different tool have been explored and used. Unfortunately, the world of model learning and -checking has not settled on a universal format for models yet. To be able to use tools like Jumbl, MCRL2, GraphViz, LearnLib, yEd, CADP and more, we had to convert between format's a lot. At the start of this research, we used mCRL2 [48] [49] for converting between different modeling formats. Since this was a bit inconvenient in combination with our own tool, we extended our tool with a converter as well. Our tool contains an internal model class that is used for most operations, we extended our tool with a 'converter' interface to read and write this model from and to different file formats. This way, our tool can be used in combination with a big number of different model learning, model checking and model based testing tools. Currently our tool can read and write the following formats:

- .aut
- .dot
- .tml
- .gml
- ASML traces (input only).

We also implemented scripts to convert ASML traces to a MCRL2 model and partially generate a stub file from a folder with header files.


Converter

Input type: .dot

Output type: .gml

Input file: C:\Radboud\learnedModel.dot

Output file:

Figure 28: Converter tool

12 Appendix C: generated tests

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class UnitTests {
    public void test1(){
        reset();
        assertEquals(twix(),"NOK");
        assertEquals(twix(),"NOK");
        assertEquals(mars(),"NOK");
        assertEquals(5ct(),"OK");
        assertEquals(mars(),"NOK");
    }
    public void test2(){
        reset();
        assertEquals(twix(),"NOK");
        assertEquals(mars(),"NOK");
        assertEquals(5ct(),"OK");
        assertEquals(mars(),"NOK");
    }
    public void test3(){
        reset();
        assertEquals(mars(),"NOK");
        assertEquals(twix(),"NOK");
        assertEquals(mars(),"NOK");
        assertEquals(5ct(),"OK");
        assertEquals(mars(),"NOK");
    }
    public void test4(){
        reset();
        assertEquals(snickers(),"NOK");
        assertEquals(twix(),"NOK");
        assertEquals(mars(),"NOK");
        assertEquals(5ct(),"OK");
        assertEquals(mars(),"NOK");
    }
    public void test5(){
        reset();
        assertEquals(5ct(),"OK");
        assertEquals(twix(),"NOK");
        assertEquals(twix(),"NOK");
        assertEquals(mars(),"NOK");
        assertEquals(5ct(),"OK");
        assertEquals(mars(),"OK");
    }
    public void test6(){
        reset();
        assertEquals(5ct(),"OK");
        assertEquals(twix(),"NOK");
        assertEquals(mars(),"NOK");
        assertEquals(5ct(),"OK");
        assertEquals(mars(),"OK");
    }

    ...

    public void test38(){
        reset();
        assertEquals(10ct(),"OK");
        assertEquals(10ct(),"OK");
        assertEquals(10ct(),"OK");
        assertEquals(10ct(),"NOK");
        assertEquals(twix(),"OK");
        assertEquals(twix(),"OK");
        assertEquals(5ct(),"OK");
        assertEquals(mars(),"NOK");
    }
}
```

13 Literature

1. Hybrid adaptive distinguishing sequences for fsm-based complete testing. `https://gitlab.science.ru.nl/moerman/hybrid-ads`. Accessed: 2018-06-21.
2. Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. Shield synthesis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 533–548. Springer, 2015.
3. Edward E Ogheneovo. On the relationship between software complexity and maintenance costs. *Journal of Computer and Communications*, 2(14):1, 2014.
4. Joost Visser, Sylvan Rigal, Rob van der Leek, Pascal van Eck, and Gijs Wijnholds. *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code.* " O'Reilly Media, Inc.", 2016.
5. Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.
6. Robert S Arnold. An introduction to software restructuring. *Tutorial on Software Restructuring*, pages 1–11, 1986.
7. Martin Fowler, Kent Beck, J Brant, William Opdyke, and Don Roberts. Refactoring: Improving the design of existing programs, 1999.
8. Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.
9. Natanael Adityasatria, JJM Hooman, MTW Schuts, and Bart GJ Knols. Applying model learning and domain specific languages to refactor legacy software. 2017.
10. Grammatech codesonar. `https://www.grammatech.com/products/codesonar`. Accessed: 2018-06-26.
11. Better Code Hub spend less time fixing bugs. and more time shipping new features. `https://bettercodehub.com`. Accessed: 2018-06-30.
12. George H Mealy. A method for synthesizing sequential circuits. *Bell Labs Technical Journal*, 34(5):1045–1079, 1955.
13. MTW Schuts. *Industrial Experiences in Applying Domain Specific Languages for System Evolution*. PhD thesis, Sl: sn, 2017.
14. Particle Sizes sizes of airborne particles as dust, pollen bacteria, virus and many more. `https://www.engineeringtoolbox.com/particle-sizes-d_934.html`. Accessed: 2018-06-21.
15. Arie J den Boef. Optical wafer metrology sensors for process-robust cd and overlay control in semiconductor device manufacturing. *Surface Topography: Metrology and Properties*, 4(2):023001, 2016.
16. Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.
17. Onderwijs Catalogus testing techniques. `https://sis.ru.nl/osiris-student/OnderwijsCatalogusSelect.do?selectie=cursus&collegejaar=2016&cursus=NWI-I00110`. Accessed: 2018-06-21.
18. Radboud university nijmegen. `https://www.ru.nl/`. Accessed: 2018-06-26.
19. Frits Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, 2017.
20. Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.
21. Mathijs Schuts, Jozef Hooman, and Frits Vaandrager. Refactoring of legacy software using model learning and equivalence checking: an industrial experience report. In *International Conference on Integrated Formal Methods*, pages 311–325. Springer, 2016.
22. Malte Isberner. Foundations of active automata learning: an algorithmic perspective. 2015.
23. Harald Raffelt, Bernhard Steffen, and Therese Berg. Learnlib: A library for automata learning and experimentation. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 62–71. ACM, 2005.
24. Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, (3):178–187, 1978.

25. Dirk Beyer and M Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*, pages 184–190. Springer, 2011.
26. Wouter Smeenk, Joshua Moerman, Frits Vaandrager, and David N Jansen. Applying automata learning to embedded control software. In *International Conference on Formal Engineering Methods*, pages 67–83. Springer, 2015.
27. Jan Tretmans. Model based testing with labelled transition systems. In *Formal methods and testing*, pages 1–38. Springer, 2008.
28. Machiel Van Der Bijl, Arend Rensink, and Jan Tretmans. Compositional testing with ioco. In *International Workshop on Formal Approaches to Software Testing*, pages 86–100. Springer, 2003.
29. Kathi Fisler and Moshe Y Vardi. Bisimulation and model checking. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 338–342. Springer, 1999.
30. Axel Belinfante, Lars Frantzen, and Christian Schallhart. 14 tools for test case generation. In *Model-Based Testing of Reactive Systems*, pages 391–438. Springer, 2005.
31. Stacy J Prowell. Jumbl: A tool for model-based statistical testing. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 9–pp. IEEE, 2003.
32. Kristian Karl. Graphwalker. *URL: www. graphwalker. org [accessed: 2015-12-18]*, 2013.
33. Junit 5. `https://junit.org/junit5/`. Accessed: 2018-07-03.
34. Lydie du Bousquet and Nicolas Zuanon. An overview of lutess a specification-based tool for testing synchronous software. In *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 208–215. IEEE, 1999.
35. Nicolas Halbwachs and Pascal Raymond. Validation of synchronous reactive systems: from formal verification to automatic testing. In *Annual Asian Computing Science Conference*, pages 1–12. Springer, 1999.
36. Bruno Marre and Agnes Arnould. Test sequences generation from lustre descriptions: Gatel. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 229–237. IEEE, 2000.
37. Alexander Pretschner, Heiko Lotzbeyer, and Jan Philipps. Model based testing in evolutionary software development. In *Rapid System Prototyping, 12th International Workshop on, 2001.*, pages 155–160. IEEE, 2001.
38. Michael Barnett, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Validating use-cases with the asml test tool. In *Quality Software, 2003. Proceedings. Third International Conference on*, pages 238–246. IEEE, 2003.
39. Michael Schmitt, Anders Ek, Jens Grabowski, Dieter Hogrefe, and Beat Koch. Autolink—putting sdl-based test generation into practice. In *Testing of Communicating Systems*, pages 227–243. Springer, 1998.
40. Paul A Gagniuc. *Markov Chains: From Theory to Implementation and Experimentation.* John Wiley & Sons, 2017.
41. William Rowan Hamilton. Travelling salesman problem.
42. MP Vasilevskii. Failure diagnosis of automata. *Cybernetics*, 9(4):653–665, 1973.
43. Arpan Sen. A quick introduction to the google c++ testing framework. *IBM DeveloperWorks*, page 20, 2010.
44. About xunit.net. `https://xunit.github.io/`. Accessed: 2018-07-03.
45. Bullseye testing technologies. `https://www.bullseye.com/`. Accessed: 2018-07-03.
46. Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker b last. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.
47. Corina S Pasareanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M Cobleigh, and Howard Barringer. Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008.

48. Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck Van Weerdenburg. The formal specification language mcrl2. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.
49. Sjoerd Cranen, Jan Friso Groote, Jeroen JA Keiren, Frank PM Stappers, Erik P De Vink, Wieger Wesselink, and Tim AC Willemse. An overview of the mcrl2 toolset and its recent advances. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 199–213. Springer, 2013.