Applying Model Learning and Domain Specific Languages to Refactor Legacy Software

Author: Natanael Adityasatria

Supervisors: prof. dr. J.J.M. Hooman
drs. ing. M.T.W. Schuts
Bart G.J. Knols, PhD MBA

July 24, 2017
Applying Model Learning and Domain Specific Languages to Refactor Legacy Software

Name               Natanael Adityasatria
Student number     4417992
Discipline         Master in Computing Science
                    Specialization: Science, Management, and Innovation (SMI)
Host organization  Philips
Start date         1 February 2017
Presentation date  19 July 2017
University coach   prof. dr. J.J.M. Hooman
Host organization coach  drs. ing. M.T.W. Schuts
Reader             Bart G.J. Knols, PhD MBA
Date of this document July 24, 2017
Abstract

Many organizations have serious challenges in maintaining complex legacy software systems to continue being competitive in today’s fast-changing business. The challenges include a lack of documentation, absence of original developers, and embedded business knowledge in the source code. The challenges to maintain legacy industrial software and the benefits of developing new software implementation are investigated. In addition, a combination of model learning, model transformations, and equivalence checking is applied to refactor complex legacy software and to improve future maintainability. This combination of approaches can produce a new software implementation that has the same behavior as its legacy version and reduces its complexity.

Keywords: model learning, domain specific language, model transformation, equivalence checking, legacy software, industrial application
Executive Summary

During software evolution, software modifications are inevitable. Software systems have to adapt to keep up with changing business requirements. Maintaining existing software systems is very crucial for organizations especially if the systems are already used by customers. As modifications are applied continuously, the complexity of software systems increases over time. Therefore, maintaining these so-called legacy software systems is difficult. Developing new software systems from scratch, however, is resource-intensive. In addition, modifications in legacy software are very risky because they can break and change the external behavior. Software refactoring is seen as an alternative to deal with legacy software systems. Code refactoring is a restructuring of internal software to improve internal quality without changing its external behavior.

A qualitative study was undertaken to investigate the challenges in maintaining legacy software. Interviews with software designers/architects and managers were chosen as a methodology to obtain primary data. The interviews also explored the benefits of developing a new software implementation compared to preserving legacy software. This qualitative study was combined with a literature study to explore how theories can be reflected in industrial applications.

Based on the interviews with maintainers and managers, there are several difficulties in maintaining legacy software. Limited documentation containing a technical description of how software component works with other software components is one of the challenges. Since the code-base is complex, the maintainers have to understand a very big scope of the software component. Consequently, maintenance becomes a time-consuming activity. Moreover, the use of multiple source code patterns makes the code difficult to understand. Over time, the current software component has to adapt to changes of its environment. Moreover, new features from new equipment force software modifications. Also, maintaining different software branches was mentioned as one of the challenges because there exist multiple software branches for different releases.

To produce a new software implementation from its legacy version, a methodology was developed. Some experiments were performed to apply this approach in order to refactor a legacy software component and to improve software maintainability. The main idea is to learn the behavior of a legacy software component and to produce a new software implementation. The methodology is a combination of several approaches: model learning, model transformations using domain-specific languages (DSLs), equivalence checking, and maintainability measurement using code metrics and domain-expert assessments.

An interventional X-ray system developed by Philips contains many software components. A particular software component named "Patient and Beam" (PandB) is used for positioning the C-arms and patient table. The PandB software component contains several layers, and the functional layer is the core layer containing the main functionality of the software component. The dynamic rotational angiography (DRA) functionality is one functionality that contains state-intensive behavior. Therefore, the DRA functionality was used as a case study for experimenting with this methodology.

Model learning was used to learn the legacy PandB software component. The outcome from this model learning is a model describing the behavior of the utilized interfaces of the PandB software component. In addition, a DSL named dotDSL was developed to perform model transformations. A model which results from model
learning (expressed in the standard graph description language Graphviz) is transformed into a ComMA (Component Modeling and Analysis) model describing the behavior of the software component. In addition, a code generator was developed in the ComMA framework to produce new code based on the ComMA model of the software component. In this way, a new software implementation can be produced from its legacy implementation. Model learning was used again to learn the behavior from the new code. The two models from model learning (the legacy code model and the refactored code model) were transformed into two mCRL2 (micro Common Representation Language 2) models using a code generator in the dotDSL. Equivalence checking was performed using these two mCRL2 models to verify whether the two software implementations describe the same behavior. Once the outcome from the equivalence checking shows that the behaviors of the two software implementations are equivalent, maintainability measurements were performed. By using several code properties and domain expert assessments, the maintainability characteristics of both the legacy and new software implementation were evaluated.

As the results from this combination of approaches, a new software implementation can be produced from its legacy version. From the maintainability assessments, the refactored code can reduce the complexity from the legacy code. However, the code duplication increases. Some improvements are possible in the code generator to improve the maintainability of the new code.

The models that are the result of model learning can also be used for model-based testing of the component. This would increase the existing test coverage. Concerning the maintainability measurement, an alternative is to run the code metrics on the ComMA models instead of the generated code.
Preface

I would like to thank all of my supervisors for guiding me through the process of finalizing this master thesis; Jozef Hooman who always actively gives me feedback and suggestions to improve my work. Mathijs Schuts who always supports me during software program creation and shares experiences on the previous research. Bart Knols who always reviews my works and guides me to finalize my thesis.

I would also like to thank all of my colleagues at Philips for helping me to solve technical questions and participating in this study.

To all of my friends at Radboud University and my families, thank you very much for always supporting my study. Also, thanks to Philips for giving me the opportunity to conduct this research. And last, but not least, I would like to thank Nuffic Neso Indonesia, for granting me Orange Tulip Scholarship (OTS) to do a Master study in the Netherlands.

Natanael - Best - 3 July 2017
Contents

Abstract v

Executive Summary vii

Preface ix

1 Introduction 1
  1.1 Software maintainability problems 1
  1.2 Legacy software problems 1
  1.3 Goal 2
  1.4 Model learning 2
  1.5 Domain specific language 3
  1.6 Research questions 3
  1.7 Methodology 3
  1.8 Thesis outline 5

2 Industrial case 7
  2.1 Interventional X-ray system 7
  2.2 Positioning software 8
  2.3 Dynamic Rotational Angiography 8

3 Qualitative study on maintenance 11
  3.1 Software maintainability 11
  3.2 Legacy Systems 12
    3.2.1 Software refactoring 12
    3.2.2 New product development 12
  3.3 Economic aspects 13
  3.4 Three aspects of software maintainability 14
  3.5 Software maintainability problems exploration (Step 1) 15
    3.5.1 Qualitative research preparation 15
    3.5.2 Selection criteria 15
    3.5.3 Selection 16
    3.5.4 Interview and analysis 16
    3.5.5 Content analysis 17
  3.6 Results 17
    3.6.1 The understanding of the software component 17
    3.6.2 Software component reuse 17
    3.6.3 Software component quality 18
    3.6.4 Software test 18
    3.6.5 Software branching 19
    3.6.6 Software environment changes 19
    3.6.7 New product development 19
    3.6.8 Product innovation 20
3.6.9 Making a decision on how to add new functionality 20

4 Applying model learning 21
4.1 Introduction to model learning 21
4.2 System under learning 22
4.3 Adapter for the SUL (Step 2) 22
4.4 Functional layer initialization 25
4.5 Trace files from system test cases 25
4.6 Handle multiple outputs 25
4.7 Handle non-determinism 26
4.8 Stateless Mealy machine 26
4.9 Encoding in input and output 27
4.10 Learned model 28

5 Model transformations 31
5.1 Transformation from model to a DSL instance (Step 3) 31
5.2 Code generator for new implementation (Step 4) 34

6 Quality of generated code 37
6.1 Equivalence checking (Step 5) 37
6.2 Maintainability measurement (Step 6) 39
6.2.1 Volume 40
6.2.2 Complexity per unit 40
6.2.3 Duplication 41
6.2.4 Unit size 41
6.2.5 Unit testing 42
6.2.6 Sub-characteristics of maintainability 42
6.3 Domain expert evaluation 43

7 Results 45
7.1 Generated model 45
7.1.1 First experiment 45
7.1.2 Second experiment 45
7.1.3 Third experiment 46
7.1.4 Fourth experiment 46
7.1.5 Fifth experiment 47
7.1.6 Interface utilization 48
7.2 Equivalence Checking 48
7.3 Maintainability metrics results (first experiment) 50
7.3.1 DRA state handler 50
7.3.2 DRA manager 50
7.3.3 Special procedure DRA 51
7.4 Maintainability metrics results (second experiment) 52
7.4.1 DRA state handler 53
7.4.2 DRA manager 54
7.4.3 Special procedure DRA 54
7.5 Domain expert assessments 55
7.6 Discussion of results 56
List of Figures

1.1 Methodology to combine model learning and DSLs .......................... 4
2.1 Interventional X-ray system [22] ................................................. 7
2.2 The comparison of lines of code and source files between the functional layer and all layers in the PandB software component ........... 9
3.1 Theoretical framework regarding software maintainability guidelines adapted from three circles model of sustainability [38] ................... 14
4.1 Active automata learning in a black-box reactive system [19] ............. 22
4.2 Interfaces between the functional layer and the other layers. 1: incoming IImpl interface; 2: outgoing IUUsed interface; 3: incoming IUUsed interface; 4: outgoing IImpl interface .......................... 23
4.3 The interaction between the learner and the SUL using a socket connection ................................................................. 24
4.4 The encoding of the interface property in the input and output string . 27
4.5 Graph representation from the learned model of the dot file in Listing 4.1 .............................................................. 29
5.1 Learned model from legacy implementation is a DSL instance ......... 31
5.2 The dotDSL abstract syntax to translate GraphViz model to ComMA model and mCRL2 model .................................................. 32
5.3 The mapping of inputs and outputs from DSL instance to the interface signatures in ComMA ....................................................... 33
5.4 The comparison between the legacy and the new functional layer in the PandB software component ........................................ 35
6.1 Equivalence checking between legacy and generated implementation [10] .............................................................. 38
7.1 Initial learned model .................................................................. 46
7.2 Initial learned model of the DRA functionality ............................. 47
7.3 Graphical representation from the model of the DRA state handler .. 48
7.4 Graphical representation from the model of the DRA manager ......... 48
7.5 Graphical representation from the model of the special procedure DRA 49
7.6 Graphical representation from the model of the DRA functionality .. 49
List of Tables

1.1 A mapping of defined steps to answer the research questions .......................... 5
3.1 A list of interview participants ........................................................................ 16
4.1 The number of available methods in the interfaces ........................................ 23
4.2 The interface utilization as the input or output for the learner which corresponds with Figure 4.2 ............................................................... 24
4.3 Some examples of the interaction which occurs between layers through interfaces. The numbers in this table correspond to the arrows depicted in Figure 4.2 ....................................................................................... 24
4.4 Some examples of input and output using the encoding .................................... 27
6.1 Mapping system of maintainability characteristics onto source code properties [25] ........................................................... 39
6.2 Ranking of total LOC [25] ........................................................................... 40
6.3 Risk evaluation of cyclomatic complexity per unit [25] ................................. 41
6.4 Ranking of cyclomatic complexity per unit [25] ............................................ 41
6.5 Ranking of code duplication [25] ................................................................ 41
6.6 Risk evaluation of unit size [49] ................................................................ 42
6.7 Ranking of unit size [49] ............................................................................ 42
6.8 Ranking of unit testing [25] ........................................................................ 42
6.9 Maintainability of the legacy implementation of the DRA manager .............. 43
7.1 The results of the learned models of the legacy DRA functionality .................. 47
7.2 Utilization of interfaces in the functional layer of the PandB software component ......................................................................................... 48
7.3 Maintainability of the legacy implementation of the DRA state handler ......... 50
7.4 Maintainability of the new implementation of the DRA state handler ......... 51
7.5 Maintainability of the legacy implementation of the DRA manager .......... 51
7.6 Maintainability of the new implementation of the DRA manager ............. 51
7.7 Maintainability of the legacy implementation of the special procedure DRA ............................................................ 52
7.8 Maintainability of the new implementation of the special procedure DRA ............................................................ 52
7.9 Maintainability of the legacy implementation of the DRA state handler ..... 53
7.10 Maintainability of the new implementation of the DRA state handler after the code generator improvement ........................................ 53
7.11 Maintainability of the legacy implementation of the DRA manager .......... 54
7.12 Maintainability of the new implementation of the DRA manager after the code generator improvement ........................................ 54
7.13 Maintainability of the legacy implementation of the special procedure DRA ............................................................ 55
7.14 Maintainability of the new implementation of the special procedure DRA after the code generator improvement .......................... 55

B.1 Comparison of the volume, duplication, and unit testing property between the legacy and the new implementation of the DRA state handler 65
B.2 Comparison of the cyclometric complexity property between the legacy and the new implementation of the DRA state handler 65
B.3 Comparison of the unit size property between the legacy and the new implementation of the DRA state handler 66
B.4 Comparison of the volume, duplication, and unit testing property between the legacy and the new implementation of the DRA manager 66
B.5 Comparison of the cyclometric complexity property between the legacy and the new implementation of the DRA manager 66
B.6 Comparison of the unit size property between the legacy and the new implementation of the DRA manager 66
B.7 Comparison of the volume, duplication, and unit testing property between the legacy and the new implementation of the special procedure DRA 67
B.8 Comparison of the cyclometric complexity property between the legacy and the new implementation of the special procedure DRA 67
B.9 Comparison of the unit size property between the legacy and the new implementation of the special procedure DRA 68

C.1 Comparison of the volume, duplication, and unit testing property between the legacy and the new implementation of the DRA state handler after the code generator improvement 69
C.2 Comparison of the cyclometric complexity property between the legacy and the new implementation of the DRA state handler after the code generator improvement 69
C.3 Comparison of the unit size property between the legacy and the new implementation of the DRA state handler after the code generator improvement 70
C.4 Comparison of the volume, duplication, and unit testing property between the legacy and the new implementation of the DRA manager after the code generator improvement 70
C.5 Comparison of the cyclometric complexity property between the legacy and the new implementation of the DRA manager after the code generator improvement 70
C.6 Comparison of the unit size property between the legacy and the new implementation of the DRA manager after the code generator improvement 71
C.7 Comparison of the volume, duplication, and unit testing property between the legacy and the new implementation of the special procedure DRA after the code generator improvement 71
C.8 Comparison of the cyclometric complexity property between the legacy and the new implementation of the special procedure DRA after the code generator improvement 71
C.9 Comparison of the unit size property between the legacy and the new implementation of the special procedure DRA after the code generator improvement 72
Chapter 1

Introduction

The topic of the thesis is introduced which explains software maintainability problems and challenges to maintain legacy software. Subsequently, this chapter includes the goal of this thesis and some information on previous research about model learning and Domain Specific Language (DSL). Next, some research questions are presented. Finally, at the end of this chapter, the research methodology is described.

1.1 Software maintainability problems

During the software life cycle, changes in both system’s environment and user requirements are inevitable [1]. As a consequence of those changes, the software components also have to change to keep up with business requirements. Modifications in the software component usually imply that the initial structure degrades and complexity increases. The result of this evolution is software design decay which is defined as the deterioration of the internal structure of system [2]. Over time, multiple people have applied changes to the software component. Since they do not understand the initial concepts, different code patterns and styles exist in the software component. Thus, modifications are harder to establish and maintainability of the software component becomes complex.

Some techniques have been developed to measure the software maintainability, for instance by measuring volume, structural complexity, and duplication [3]. Generally speaking, the availability of code analysis tools to measure software maintainability does not give a complete understanding of how maintainable the software is. To get a more complete overview of overall maintainability aspects of a system, one should apply a combination of different approaches. For instance, exploring maintainability aspects that are important for software maintainers [4] and using expert assessments to detect many potential maintainability problems [5]. Hence, it is important to explore the considerations and perspectives from both software designers/architects and managers on their choice for specific technologies to assess software maintainability. These considerations include social and economical aspects such as the cost of required resources and the compatibility of the technologies with the company culture. In addition, a qualitative analysis is perceived as a way to get the perception of the stakeholders on those aspects.

1.2 Legacy software problems

Maintaining software is not an easy task because it requires in-depth knowledge of the application. This knowledge is often available explicitly through documentation or the (original) developers. To be able to maintain software components for
future development, well-written and up-to-date documentation plays a significant role, especially the information on how and why software components work the way they do. However, when a software component was developed years ago without proper documentation and the original developers are no longer available, it is difficult to maintain and extend the use of this so-called legacy software. Typical legacy systems have been characterized as old – they were built on obsolete technologies – but they are still performing crucial work for their organizations [6]. With limited availability of documentation, source code is seen as the only reliable source of information about the system. Consequently, extracting information from source code is considered as the only way to gain sufficient understanding of the original requirements.

The cost to maintain legacy systems increases over time [7]. Finding qualified programmers to perform maintenance of obsolete technologies is expensive and difficult. On the other hand, developing a new system is resource-intensive [8], [9] and needs a massive amount of testing to ensure that it is compliant with the legacy system. Also, the modification of legacy systems is risky and requires a significant investment. Therefore, a lot of aspects influence the decision on how to deal with legacy systems.

### 1.3 Goal

The research described in this thesis is based on previous research at Philips on legacy software. It focuses mainly on finding a solution to the software maintainability and legacy software problems using a combination of model learning and Domain Specific Language (DSL) approach. The primary objective was to investigate whether DSLs and model learning can help to refactor legacy components and enhance future maintainability. This study also investigated when there is a good business case for software refactoring in the context of Image Guided Therapy (IGT) Systems. IGT Systems is a business unit that innovates in the development of image-guided interventional solutions at Philips.

### 1.4 Model learning

A study was undertaken to learn legacy software using model learning and equivalence checking at Philips [10]. This study used the model learning tool LearnLib [11] and suggested to use a faster algorithm for automata learning. The research described here also uses LearnLib to generate two models: one from the legacy implementation and one from the refactored implementation, but with a faster algorithm. These models will be compared to check if two implementations are equivalent using an equivalence checker to be sure that the new implementation has the same behavior as the legacy version. Moreover, the availability of logging functionality in the legacy software could improve the existing model learning algorithm by reducing the number of inputs required [12].

---

1. Refactor software code without changing its external behavior
2. Automata learning is a method to construct a finite-state machine
1.5 Domain specific language

Enhancing the maintainability of the software is one of the advantages of a well-designed DSL [13]. A DSL is a language that is designed to tackle a certain class of problems, called a domain [14]. A DSL is used to simplify the software’s complexity by separating its concern into high-level domain-specific abstractions. Advantages of using a DSL are that it is easier to modify as well as to understand [15]. In addition, DSLs provides a clean, customized, productive environment that allows non-programmers to work with languages that are closely aligned with the domain in which they work.

DSL development includes a meta-model and a grammar [16]. The meta-model (abstract syntax) identifies the fundamental concepts for modeling the component and the grammar (concrete syntax) defines the textual language used for describing instances of the language. Xtext [17] is used to develop the DSL in which parser and meta-model are generated automatically. Moreover, Xtend [18] is a programming language which facilitates the definition of generators that transforms the high-level abstraction in the DSL into text such as source code. In conclusion, Xtext and Xtend will be used to develop a DSL to refactor the legacy software.

1.6 Research questions

The main research questions addressed in this thesis are:

(RQ1) a) What are the maintainability challenges with the legacy software components in IGT Systems?
    b) What are the benefits (or disadvantages) of developing a new implementation compared with keeping the legacy software?

(RQ2) a) Which steps are required to implement model learning techniques to produce a model which is an instance of a DSL?
    b) How can model learning and DSLs be combined in an effective way?

(RQ3) a) How can software maintainability be characterized in the context of IGT Systems?
    b) What steps are required to implement a source code generator using a DSL that produces maintainable source code from legacy component?
    c) Is the generated source code from the DSL better maintainable than the legacy implementation?

1.7 Methodology

A combination of model learning and DSLs has been selected because model learning brings a (semi) automated way to extract valuable information from the source code [19] and DSLs can provide an abstraction and code generation tools. Some steps are required to find an effective way to make an automatic generation and to combine both model learning and DSLs. Specifically, the generation of the learned model of a legacy software to a DSL instance for a new software implementation could potentially require an (intermediate) transformation. The approach to answer the research questions can be described as follows:

Step 1 Conduct a literature study and interviews to investigate the challenges with legacy software components and how to deal with these;
Step 2 Learn the behavior of a legacy component using model learning; the outcome of this step is a model expressed in a graph description language namely GraphViz grammar;

Step 3 Design an instance of a DSL from the learned model of the legacy implementation (marked with triangle 2 in Figure 1.1); the ComMA (Component Modeling and Analysis) model is used for specifying the interface behavior of software component;

Step 4 Generate source code from a DSL instance to produce a new implementation (marked with triangle 3 in Figure 1.1);

Step 5 Validate the equivalence between the legacy implementation and the new implementation (marked with triangle 5, 6, and 7 in Figure 1.1); the mCRL2 (micro Common Representation Language 2) model is used for describing system behavior;

Step 6 Measure the software maintainability of the generated code using software maintainability tools as well as a qualitative analysis to get an understanding of the DSL contribution to the project (marked with triangle 8, 9, and 10 in Figure 1.1);

Step 7 Develop a business case for software refactoring and draw an overall conclusion.

![Diagram](image)

Figure 1.1: Methodology to combine model learning and DSLs
These seven steps will provide answers to all defined research questions. Table 1.1 shows how the steps correspond to the research questions.

<table>
<thead>
<tr>
<th>Step</th>
<th>Research questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>(RQ1) a, (RQ1) b</td>
</tr>
<tr>
<td>Step 2</td>
<td>(RQ2) a</td>
</tr>
<tr>
<td>Step 3</td>
<td>(RQ2) a, (RQ2) b</td>
</tr>
<tr>
<td>Step 4</td>
<td>(RQ2) b, (RQ3) b</td>
</tr>
<tr>
<td>Step 5</td>
<td>(RQ3) b</td>
</tr>
<tr>
<td>Step 6</td>
<td>(RQ3) a, (RQ3) c</td>
</tr>
<tr>
<td>Step 7</td>
<td>(RQ3) c</td>
</tr>
</tbody>
</table>

In Step 1, a literature review is used to explore if there are case studies about software maintainability regarding legacy software. Also, the purpose of this literature study is to investigate how the theory can be applied to deal with the problems of legacy systems. Next, literature study will be combined with the relevant findings from the interviews so that the analysis might result in a complete picture of the problems. Furthermore, the steps to find a method to refactor the legacy component using the combination of model learning and the DSLs will be applied in an iterative way in which the experiment will start from a small subset of a legacy software component and later will increase in complexity (Step 2 - Step 6). Furthermore, all steps will be elaborated on in detail in the next chapters.

1.8 Thesis outline

The remainder of this thesis is structured as follows. In Chapter 2, the industrial case for this study is explained. Subsequently, in Chapter 3, the theoretical framework and method for investigating software maintainability problems in combination with literature study are presented. This chapter also contains the primary data gathered during the qualitative study. Chapter 4 describes the application of model learning. Next, Chapter 5 explains the approach to generate a new software implementation using the DSLs. Chapter 6 describes the equivalence check between the legacy and the refactored software implementation. This chapter also explains the software maintainability measurement. In Chapter 7, the results of this study are presented and discussed in a broader context. Finally, Chapter 8 lists the conclusion and recommendations.
Chapter 2

Industrial case

This chapter introduces interventional X-ray systems and explains some applications of these systems in hospitals. Subsequently, it continues to the explanation on an interventional X-ray system developed by Philips and a particular software component that is responsible for controlling the movement of the system. Next, a particular functionality in the software component is explained, which is used for a case study in this thesis.

2.1 Interventional X-ray system

Interventional X-ray systems are used for minimally-invasive surgery. Minimally-invasive surgery involves the same procedure as conventional surgery but with minimum incisions. This is believed to decrease pain and have less physiologic impact on the patient. Some advantages of minimally-invasive surgery compared to open surgery are reduced recovery time, fewer complications, and shorter hospital stays resulting less hospital utilization [20]. Furthermore, the development of noninvasive devices and minimally-invasive surgical techniques can prevent and control nosocomial infections [21].

The most important aspect of minimally-invasive surgery is that it involves the visualization of internal organs. Also, the manipulation of the organ is executed using endoscopes, catheters, and needles through a small opening in the blood vessels/skin and guided by X-ray images. Images produced by this system allow physicians to guide instruments through the body to perform therapy. Figure 2.1 shows
an interventional X-ray system developed by Philips containing a C-arm with an X-ray tube and a detector, an L-arm to hold the C-arm to the ceiling, a patient table, a touch screen module, an UI module for controlling the movement and the monitor for visualization. This equipment is placed in the so-called intervention room where the patient lays down on the table and physicians perform the treatment. Another room involved is the so-called control room where other staff, such as, nurses support physicians for controlling the movement of the equipment for the visualization on the monitor. In addition, there are many treatments which are supported by these system variants such as cardiology intervention, neuroradiology intervention, vascular surgery, and oncology intervention.

The interventional X-ray system developed by Philips for minimally-invasive interventions contains a large number of software components. The system has been maintained for more than twenty years and the increasing amount of software makes it difficult to maintain. In addition, when the system has been released to the customers, it has to be maintained for about 10 years.

2.2 Positioning software

During treatment, high-quality X-ray images of the patient’s internal organ are very critical for surgeons. To produce high-quality images, some parts of the interventional X-ray system such as the C-arms, the X-ray detectors, and the patient table, may move to obtain optimal projections. The movement in the system is controlled by users and embedded software is responsible for operating the motion. This software is not only for operating the motion of the system but it is also responsible for controlling the priority of the inputs to the system. For example, there are some sources of input for this software to decide the movement that it is going to be made. The inputs can be from switches in the intervention room or switches in the control room. A different source of input has a different priority to the system and the software processes the inputs based on the priority. One particular software component for positioning both C-arms and patient table was used as a case study in this thesis namely the Patient and Beam (PandB) software component.

The software architecture in the PandB software component consists of some layers in which every layer has its purpose. One particular layer, the functional layer, contains the functional behavior of the software component. This layer is the core of the software component containing the main functionality. Since we are interested in the most essential part of the software component which contains the crucial business information, we concentrated on the functional layer and eliminated other layers. As can be seen in the Figure 2.2, the total lines of code (LOC) of the software component is around 850 thousands. In addition, the total LOC in the functional layer alone is about one-third of the total LOC in the software component. Furthermore, the PandB software component contains about one thousand source files in which the functional layer is only about one-third of it.

2.3 Dynamic Rotational Angiography

To start with a small subset of the functional layer, this research focuses on one particular feature in the functional layer. This feature is called Dynamic Rotational Angiography (DRA). DRA is driven by a combination of a quick movement of the C-arm and an X-ray image acquisition. One purpose of this DRA run is, for example,
2.3. Dynamic Rotational Angiography

![LOC, Lines of Code vs Source files](image)

**Figure 2.2**: The comparison of lines of code and source files between the functional layer and all layers in the PandB software component

To visualize an internal organ of a patient in a three-dimensional picture in which the C-arm rotates and at the same time some X-ray images are captured.

In running the interventional X-ray system, there are some pre-conditions to run the DRA procedure. For instance, there is a pre-condition in which a DRA run is not supported and it needs an activation. Another pre-condition is, for example, that the start and end position of the C-arm should be defined before the actual movement. In addition, there is a condition in which the C-arm reaches its final position and finishes the movement, so the DRA run should be stopped and deactivated. Intuitively, these conditions in the system will construct some states and transitions when it is translated into a formal model, such as in a state machine representation. With the aim to produce a model describing the behaviour of the DRA functionality, given these conditions, it is expected that the generated model will have some states and transitions.
Chapter 3

Qualitative study on maintenance

This chapter elaborates on the maintainability problem following the first research question. This background information includes the definition of software maintainability, legacy systems, and a perspective from the economics of software engineering. Then, it continues to explain the methodology used to conduct the interviews. The results of the interviews of IGT Systems are presented in the last part of this chapter.

3.1 Software maintainability

According to the IEEE software engineering terminology (2002), the definition of 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" [23]. In addition, maintainability is a quality factor of the software to produce more productive maintenance phase [24]. In other words, software maintainability is strongly related to software quality. A variety of models exists for assessing the quality of software product. One example is the ISO/IEC 9126 standard (replaced by the ISO/IEC 25010) for software engineering product quality. This ISO standard addresses software maintainability as one criterion of the quality structures. Thus, to improve the quality of the software product, one should take into account the maintainability aspect.

Assessing software maintainability involves many aspects of software that affect maintenance activity. For instance, software maintainability in the ISO 9126 standard is refined by the attributes analyzability, changeability, stability, and testability. For only one attribute, one might use different properties to evaluate the attribute, either from properties that are (not) source-code related. An example of a source-code property is code duplication and an example of a non-source-code property is the deployment process [25]. For the source-code related properties, one can use static code analysis tools to compute the measurement automatically. This requires a precise definition of the property so that the measurement can be performed easily. For instance, one should define clearly what code duplication means, e.g., how many lines of code are classified as duplication. Furthermore, one should also specify precise definition of maintainability attribute in terms of properties so that the overall maintainability assessment can give a clear picture of the maintainability of the software.

Creating a new system is not a warranty to improve the system maintainability. A new system which was set up with new structured development techniques may be easier to maintain at the beginning, but the size and complexity of that system will probably also increase. A good measure of software maintainability is needed because it can help to manage the effort during the software maintenance phase.
better. It has been argued that the cost to maintain a software system is related to its maintainability; a software system with poor maintainability will be more expensive in the maintenance stage than one with higher maintainability [26].

### 3.2 Legacy Systems

Legacy systems are aging (business) software systems that are increasingly difficult to modify and to evolve because they tend to break easily when modified [27]. Many legacy systems are critical to businesses, contain embedded business knowledge, and are difficult to adapt in case of the changing business requirements. Nowadays, many organizations are facing the problem of managing, maintaining, and rejuvenating legacy systems. They must overcome serious challenges to continue being competitive in today’s fast-changing business and technological environments. Managing and maintaining a legacy system involves an environment where knowledge on how to proceed is scanty. Some examples of the factors related to this scantiness of knowledge are the size and complexity of the system, high employee turnover, limited documentation, and the long period these systems must be maintained. Most of the time, people who maintain the system are not the same as the people who developed the system and previously undetected errors should be fixed by the maintainers.

Over time, legacy systems often obtained additional layers and wrapped to provide new features. One of the reasons for these layers is that these are low cost and safe; since there is no software modification, original functionality is preserved [28]. For large software systems where the legacy part and modern part are mixed, it has become imperative to phase out the legacy part and substitute these with modern architectures that support the rapidly-changing business condition of today. However, modernization of legacy software remains a significant cost item.

In practice, there are two ways to deal with legacy systems: by refactoring source code and by starting with a new product.

#### 3.2.1 Software refactoring

Instead of adding more layers to an existing legacy system, one can also refactor the source code. Refactoring is defined as the process of changing a software system to improve its internal structure without altering the external behavior of the code [29]. When a software system is revised repeatedly, such as adding and changing functionality, the code is becoming more complex and the maintainability of this evolving software system will decrease over time [30]. Consequently, the quality of the code decreases [31]. Given this situation, software refactoring plays an important role because by improving the internal quality of the code, it will help to reduce the complexity. Since code modifications in legacy software are very risky, one should find an appropriate way to refactor the software.

#### 3.2.2 New product development

When software is difficult to maintain, why should it be preserved? We should not declare "old" software obsolete [32]. The reason is that this collection of functions is an important asset containing a rich information of experiences, "ideas", and knowledge to discover building blocks that can be usable for future systems [32].

Developing a new product from scratch without involving the current legacy system is very costly. This not only applies to the resources needed to develop new
product but also to the maintenance after its release. A practical approach to introduce development of a new technology and techniques into an existing production environment is to apply them to a small-scale project [33]. For instance, a small extension to some legacy systems. This approach can leverage the previous investment as well as minimize the risk of breaking the existing system.

From a reusability standpoint, the interface of a software component is essential to be more understandable and clean because the integration into a new system will be much easier. The goal of software reuse is to use the existing software to build new software systems aiming to reduce time-to-market and produce high-quality software [34]. Instead of building software from scratch, software reuse increases productivity, reduces costs, and faster time to production. In addition, from a modularity standpoint, the legacy software component needs to be self-contained and not overly coupled with other software components so reusing the legacy software component can be easier by detaching it from its original system.

### 3.3 Economic aspects

From an economics point of view of software engineering, the introduction of new software from scratch into an organization for efficiency purpose is not an attractive investment. The average number of Lines of Code (LOC) produced by professional software R&D staff is about 700 lines per month [35]. For instance, suppose a new functionality needs to be added to software with one million LOC which is built with about 119 man-years. To implement this new functionality, one needs three man-years. In a case of building the software system from scratch, this new feature can be added with only one man-year assuming that it is very easy to add new functionality to the new software system. Therefore, the option to add new functionality by building a software system from scratch will cost 120 man-years (a total from creating a new implementation with 119 man-years plus a new functionality with one man-year). By comparing between 120 man-years versus three man-years, it is clear that the less expensive way to add a new feature is the one that requires three man-years. It becomes a default approach in the industry. To a large extent, this is due to the effort needed to reproduce the software from scratch when its documentation is lacking, the original implementers are no longer available, and the business logic is hidden in the programs. As long as the existing software implementation is adequate to satisfy the current requirements, the introduction of new software implementation despite its improved efficiency is less attractive. However, in a case whereby multiple new features need to be added to software, one new feature might require more effort than others. Also, to implement new features will become more difficult as the complexity increases. Therefore, one should consider long-term consequences during the development of new features in software.

Another point of view is presented by Schach [36] about introducing a new coding technique which is faster than the old coding technique. Practically, the faster (new) technique is the technique of choice, but from the economics of software engineering, it might lead to a different choice. There are two reasons for this situation. Firstly, suppose that the new software technology is 10 percent faster than the current technology. However, the fact that the cost incurred to develop new technology might be far larger than the more rapid functionality’s advantages, then this situation becomes less attractive. The cost incurred includes the training in which the staff needs to get used to the new technology and the learning curve might be high. Hence, to complete a project using a new technology might take far longer than if
the organization had continued to use old technology [36]. Secondly, another reason of keeping the old implementation is the cost of maintenance. Although, the new implementation is 10 percent faster than the existing implementation, it could be the case that the new application is harder to maintain. Consequently, the cost to maintain is higher over the life of the product. Sometimes the consideration to choose new technology that is faster than the old one is driven by a goal to produce a product as quickly as possible. The long-term consequence of using a particular technique is often ignored in the interest of short-term benefits [36].

There are several factors affecting maintenance costs such as maintenance team stability, staff skills, software age, and structure. It is believed that the cost to maintain a legacy system will decrease if the same persons of the maintenance team are involved for a prolonged period of time [37]. Also, the more experience the staff has with the system and domain knowledge, the lower the maintenance cost will be.

3.4 Three aspects of software maintainability

Research on software maintainability suggests that one should not only focus on technical but also nontechnical issues [39]. We categorize three aspects that contribute to the software maintainability. The first aspect concerns the maintainability factors of the software such as the structure and the complexity of the code. This aspect is sourced from the point of view of the software product. The second aspect is the socio-economic aspect of software maintainability such as the number of people that do the maintenance job, the maintenance cost and time required to maintain the software. The third aspect is the corporate aspect for guiding maintenance activity. One example of this aspect is the perception from the maintainers who deal with the maintenance situation on a daily basis and have the knowledge and experience about the domain. Since they understand the technical characteristics of the software product in detail, they also understand what future improvements are needed. Another example is the management decision. A study explained some guidelines for improving the management in maintenance activity. This study suggested the management team should involve maintainers in the design and testing phase, and

Figure 3.1: Theoretical framework regarding software maintainability guidelines adapted from three circles model of sustainability [38]
rotate personnel between design and maintenance phase [32]. These three aspects are illustrated in Figure 3.1.

3.5 Software maintainability problems exploration (Step 1)

This section elaborates on the methodology shown in Step 1 for conducting the interview (Section 1.7). The interview was chosen to answer the first research question about the investigation on the maintainability difficulties with the legacy software at IGT Systems.

The perceptual differences between people who deal with maintenance issues on daily basis and people from management about the maintainability aspects could potentially give a different understanding of the real challenges with the legacy components. Therefore, qualitative research was conducted with a group of software developers and managers that are affiliated with legacy software components at IGT Systems. Interviews were used to explore information that could not be measured by software tools about software maintainability, for example, how the software developers perceived the technical maintainability of the legacy components.

3.5.1 Qualitative research preparation

For this qualitative research, it is not the goal to generalize the result to the whole study population. Instead, this research explored some circumstances on what was happening within a small group of people as this might provide insights into the behavior of the wider research population. Therefore, a purposive sampling technique was taken which is a sampling technique for finding more on description rather than generalization [40]. A small number of people was invited to participate in this research. A quota sample was selected targeting six people from two job levels, three software designers, and three managers. Different people from various roles might give a different perspective on what the problems are, and it resulted in a variation of viewpoints of the problems concerning legacy software. The interview was preferred as a method for this research for gaining some accurate information based on an individual’s views and experiences.

3.5.2 Selection criteria

The interviewees were selected based on certain criteria. The scope of the first research question is within the IGT Systems at Philips which is a business unit consisting of some departments. Therefore, the first criterion to find some interviewees was that they had to be based in one of the departments within the IGT Systems. The second criterion was, as mentioned above, that the participants should have a job role as either a software designer/architect or a manager. The investigation on the maintenance problems of legacy software requires the third selection criterion in which the interviewees should be involved in a maintenance project, so they had actual working experience related to the legacy software. The fourth criterion was optional and expressed that the respondents also participated in the development of new functionality in the software. This last criterion was added for finding a comparison of different difficulties between persons only working with legacy software and developing new functionality whilst doing the maintenance. A wide range of working experience with legacy software was also a consideration to choose the interviewees because new people and experienced people might give a different perspective.

In summary, the participants were selected based on these criteria as follows:
Chapter 3. Qualitative study on maintenance

1. Work in one of the departments in IGT Systems business unit
2. Has a job role either as a software designer/architect or a manager
3. Involved in a maintenance project of legacy software
4. Involved in a development of new functionality (optional criterion)

3.5.3 Selection

Based on the defined selection criteria, the participants were selected by the main researcher of this thesis with support from Philips’ supervisor. Next, the participants were invited for the interview. Software designers were selected from a different type of responsibility and years of working experience on a single software component. Also, three managers were interviewed from different management levels. One person is a manager of a particular department, one person is responsible for coordinating two departments, and one person is a manager in the business cluster. The result of the selection and some characteristics of the participants are shown in Table 3.1.

<table>
<thead>
<tr>
<th>Person</th>
<th>Position</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Software designer</td>
<td>Maintain legacy software (2 years)</td>
</tr>
<tr>
<td>2</td>
<td>Software architect</td>
<td>Maintain legacy software &amp; develop new features (1.5 years)</td>
</tr>
<tr>
<td>3</td>
<td>Software designer</td>
<td>Maintain legacy software &amp; develop new features (2.5 years)</td>
</tr>
<tr>
<td>4</td>
<td>Manager</td>
<td>Manage 35 people in two departments which maintain legacy software &amp; develop new features</td>
</tr>
<tr>
<td>5</td>
<td>Manager</td>
<td>Manage 35 people in a department which maintain legacy software &amp; develop new features</td>
</tr>
<tr>
<td>6</td>
<td>Manager</td>
<td>Manage several software departments in a business cluster</td>
</tr>
</tbody>
</table>

3.5.4 Interview and analysis

Since there are two target groups of interviewees which are managers and software designers, the questions were designed differently for each target group. The reason is that it might be interesting to ask more high-level questions to the managers, for instance, about the strategy to deal with legacy software. The type of questions for the software designers is more on the difficulties they found when maintaining legacy software on a daily basis. The interview was set up in a semi-structural form and lasted for about one hour. A semi-structured interview was picked because it would help to get specific information and still provide flexibility to adapt the interview to the situations so that other valuable insights can be obtained. Also, the interviews were recorded in audio format file and transcribed verbatim for further analysis. Most of the questions used are open-ended questions. The list of questions is shown in Appendix A.
3.5.5 Content analysis

After all the interviews were fully transcribed, content analysis was performed systematically. All the transcripts were read to find a list of categories from the data concerning the difficulties in maintaining legacy software and benefits of developing new software implementation. In this way, relevant findings from the interview data can be extracted to answer the first research question. Coding by content was performed by examining each transcript and assigning codes per words or specific characteristics within the text. The program atlas.ti\(^1\) was used for coding. This list of code words was used as a basis to categorize the results.

3.6 Results

The results from the interviews are described in this section. Some verbatim quotations from the interview data are shown to support the objectivity of the results.

3.6.1 The understanding of the software component

The three software designers were asked about their general understanding of how legacy software works. According to some, a high-level concept on how the software component works is clear for the software maintainers. This is supported by the availability of high-level documentation containing class and sequence diagrams. However, when it comes to the interaction of the software component with other software components, for instance, the communication through external interfaces, from maintainer’s point of view, it is difficult to see how the software components work together. As one said "I learned from embedded level till high-level software ... the scope is very big." This is the typical problem with distributed systems where a lot of software components interact with each other in a real-time environment. It was also mentioned that the software component contains many code design patterns. Hence, when one is not familiar with the use of design pattern, then the code becomes difficult to understand. In addition, there is no detailed documentation about the communication between components. Therefore, debugging and tracing are the main activities to find a solution if there is a defect in the software component. Moreover, a trace file can be useful to see how the interaction of different processes from various components. This is currently not available for every interaction in the external interfaces.

There was very detailed documentation in the past. However, it turned out that the documentation is outdated because developers update the code but they do not update the documentation accordingly. One participant indicated the importance of finding the right balance when writing documentation, i.e., writing not many documentation and also not too less. The availability of comments in the code as part of the documentation is helpful for the maintainers. However, a careful action needs to be taken as mentioned by a maintainer that "comments can be old ... sometimes you change the code, but the comment does not change, the comment does make sense anymore."

3.6.2 Software component reuse

The concept of software reuse is applied during a new product development. According to two managers, writing code from scratch will be triggered only if there

\(^1\)http://atlasti.com/
is a new requirement that cannot be implemented in the old architecture. As long as the old architecture can still facilitate the new requirement, then it will be added from the old system by default. Furthermore, to add new functionality to the legacy software is not easy because of its complexity. Not only because of the size of a single layer is enormous but also how different layers communicate with each other. When the architecture of the legacy software does not fit anymore with a new software feature, it might trigger a rewrite of the code or add a new layer. In a case of adding a new layer, this makes the software even become more complex. Given this situation, a manager pointed out that "the more code you have, the more errors in there... it is actually not linear", so reducing the lines of code (LOC) might be a good solution. For instance, code refactoring to remove the unused/dead code. This is currently a process that is still ongoing in the department.

3.6.3 Software component quality

The quality of the code is currently maintained using static code analysis. Static code analysis gives some insights which code needs to be refactored, for example, dead code. The use of the current static code analysis is very beneficial. The measurements of code coverage, total lines of code, and code complexity, for instance, are valuable to maintain the quality of the software. One software maintainer mentioned that the code understandability and readability are two important aspects of supporting the maintenance job in which the static code analysis does not provide any help. The person mentioned: "So if you see a piece of code, name makes sense to what it really does, which is not the case" meaning that it is important to give meaningful names. The person referred to class names which should describe its responsibility and purpose.

It was found that sometimes the class names do not describe the actual utility of the classes. Identifier names are very crucial; if one miss-uses the identifier name, then it will be hard to understand the intended use of it in the later stage [41]. Meaningful identifier names are basic ingredients for reasoning about program behavior since the developers chose them for describing the purpose of the class or the method [42]. Class responsibilities are not always clear, for instance, it was found one class that has a role of two responsibilities in which this class should have a proper name describing two responsibilities, but it is not the case. When faced with class inheritance, both the child name and parent class name should make sense and be compatible with each other giving the maintainers the meaning what these classes do and what the responsibilities are. However, measuring the code readability and understandability is hard and can be subjective as it needs a human judgment of how easy a text is to understand. One study has introduced a method to automatically determine a functionally descriptive name of a method or class based on the information in the body of the method or class [42]. Notwithstanding, this approach assumes the body of the method or class is descriptive.

3.6.4 Software test

Incomplete test coverage is one of the problems when doing the software refactoring. One manager mentioned that in case that one needs to refactor a part of the source code, testing is obligatory: "After the refactor you have to test whether it is still performing and function," otherwise it is difficult to be sure that the applied code changes are still performing well and do not break the other parts of the code. Although the software system has a full code coverage meaning every part of the
software is covered by testing tools, introducing a big refactoring might still have some risks, for instance, it introduces new software blocks which are not covered by the current tests or it misses some critical parts in the software. However, the benefit of software refactoring was justified by a manager by a case of where to apply code changes, in the past one should modify in 10 or 20 places in the software but now it only requires changes in one or two places. In case of software refactoring, one should choose the priority whether taking the risk to refactoring code in a very small release and limited test time or putting it in the backlog for the next software release.

According to the maintainers, software needs to adapt to new hardware and new security requirements. Improving the quality of code is a continuous process in the department. In addition, when there is a defect in the software, the maintainers need to figure it out what the cause of the defect is. This includes the work of discovering the code-base using tracing and debugging tools. Sometimes when the software was created in the past, the design was not intended to be maintained for a long time, like 30 or 40 years. Therefore, the complete documentation did not exist for that reason. The design decision in the past is completely different with how software is created today.

3.6.5 Software branching

According to a maintainer, currently, there exist multiple branches of software releases because every release has its separate branch. A new software branch is added with every release, and every branch should be supported because they are used in the field by the customers. In case a modification is made in one software branch, the maintainer needs to be aware of the other software branches. Often software changes in a software branch require changes in the other branches. This is double work for the maintainers because this does not have any additional value. Also, there is a potential that the maintainers overlook to trigger changes in the other software branches. This software branching problem is also confirmed by a manager who stated that ideally one should maintain a small number of software branches.

3.6.6 Software environment changes

According to a manager, it was found that the PandB software component is very hard to decouple at the system level design. This situation results in a high unit dependency in which it is hard to modify a part of the system without testing the whole system. Consequently, once the software infrastructure changes then it will force all software units to change as well. The obsolescence of hardware forces an architecture change in which the software needs to adapt to the new architecture. Also, operating system obsolescence will force software modification, for example, the obsolescence of Windows XP leads to large maintenance effort.

3.6.7 New product development

In the new product development, the concept of software reuse was applied by combining the old architecture with the part containing new functionality. The development of the new functionality, in this case, was more on adapting the external interfaces to fit with a new architecture. The new architecture was designed because of
the utilization of new equipment involved in the system. Therefore, the new product development was focusing on designing the communication between different software systems through external interfaces to support new equipment.

Based on the experience of one maintainer, there exist a steep learning curve at the beginning of new product development project to discover how the new software feature will suit best with the new architecture and framework design. The challenges for a new product development include the backward compatibility, whether the software component in the new product development will still support the old architecture.

3.6.8 Product innovation

For high-risk software projects involving state-of-the-art methodology, it was mentioned by two managers that the company normally collaborates with academic partners through, for instance, subsidy projects. These greenfield projects investigate how to improve the existing software systems and make the maintenance process efficient. One example is the use of model learning to understand the behavior of the state-intensive legacy system as well as to produce a model in which from the model, the automatic testing can be produced. This type of cooperation combines two domains: a domain knowledge of the product from the employees and a domain knowledge of the high-tech approaches from the academic researchers.

A manager indicated one considerable advantage that can be gained by the company is that this type of collaboration will not take many current resources. Instead, the existing resources are utilized to focus on the product development while at the same time assisting the academic partner to achieve the goal of the research. A manager also pointed out that it is not only about the product as the outcome of this cooperation project but also the approach and the knowledge that the company can learn from the approach. In this way, the company can gain more confidence and comprehension on a new way of working whether to apply the option as the company standard or to explore other options.

3.6.9 Making a decision on how to add new functionality

Although developing a new software implementation requires a lot of effort and is very risky, it might be needed in some situations. As mentioned by a manager, at a certain moment in time the department has to decide either to start with a clean software design or to add new functionality in current software design. For instance, in a case of introducing new hardware in a product, one should balance the advantages and additional value obtained by introducing new functionality in the current design or by developing a subset of the software system from scratch.
Chapter 4

Applying model learning

This chapter explains the model learning approach in a legacy software component as used in Step 2 of the methodology (Section 1.7), including the preparation and problems encountered during the setup. This chapter starts with a brief introduction on model learning. Next, it continues to the explanation on the system under learning (SUL) and the adapter application. Subsequently, it explains some problems during the setup and its solution including initialization problem, non-determinism problem, and stateless Mealy machine problem. Finally, this chapter explains the encoding used for handling parameter data.

4.1 Introduction to model learning

Model learning aims to construct a state diagram model containing the behavior of a software system by providing inputs and observing outputs [19]. We refer to [19] for an overview of model learning. In active automata learning, a model is constructed by active interaction and by reasoning on the observed output behavior. In the so-called minimally adequate teacher (MAT) framework [43], the construction of models involves a "learner" and a "teacher" in which a "learner" has the task to learn a model of a system by actively asking queries to a "teacher". There are two types of queries: membership queries (MQs) and equivalence query (EQ). The "learner" infers a model by giving inputs to the system under learning (SUL) and observing outputs via MQs. The "learner" asks the "teacher" what the output is in response to an input sequence using MQs. Given a hypothesized model, the "learner" asks the "teacher" whether the model is correct via EQ. The "teacher" uses a conformance testing (CT) tool to answer that query using a finite number of test queries (TQs). If the hypothesized model is incorrect, then the "teacher" will supply a counterexample to the "learner" that distinguish the constructed hypothesized model and the SUL model. Based on this counterexample, the "learner" may construct an improved hypothesis by giving more input sequences via MQs. The learning process ends if the "teacher" does not return a counterexample to the "learner" anymore. The process of active automata learning in a black-box system is illustrated in Figure 4.1. Furthermore, during the learning process, the "learner" can reset the "teacher" at any point. Reset is used to change the current state of the SUL back to its initial state. Reset is important because active learning requires membership queries to be independent [44]. Hence, the reset command is executed after every query.

For this model learning implementation, we used LearnLib\(^1\), an open-source framework for active automata learning. This library provides a rich set of model learning algorithms. In addition, we applied a particular algorithm called the TTT algorithm which currently the most efficient algorithm for active learning [19].

\(^1\)https://learnlib.de/
LearnLib uses the AutomataLib\(^2\) library for representing and manipulating automata, for example, in a form of Mealy machines [45]. Mealy machines are characterized by inputs which are always enabled. This means the transition function is defined for all input symbols and their response to an input (sequence) is uniquely determined [44]. LearnLib has the functionality to provide a hypothesis in the form of a Mealy machine which is represented in the standard graph description language GraphViz. Moreover, AutomataLib mainly focuses on deterministic automata, so model learning implementation using LearnLib assumes the SUL is deterministic.

### 4.2 System under learning

Model learning will be applied to the PandB software component, as explained in Chapter 2. The aim is to learn a model of the functional layer which is a subset of the PandB software component since this layer contains business logics and core functionality. Since this study focuses on learning the DRA functionality, the system under learning is the DRA functionality of the functional layer. This DRA functionality is implemented by a number of classes. To generate a relatively simple model, our experiments were limited to three different DRA classes of the DRA functionality, namely: DRA state handler, DRA manager, and special procedure DRA.

### 4.3 Adapter for the SUL (Step 2)

To learn the behavior of the SUL, we need to understand the architecture of the PandB software component. As can be seen in Figure 4.2, the functional layer is surrounded by other layers namely the representation layer and the abstraction layer. In the running software, the functional layer needs to communicate with these two layers to connect to external interfaces of the functional layer, for instance, for data exchange.

To construct a model from the SUL, understanding the available interfaces is required. There are two interfaces which are related to the functional layer namely

---

\(^2\)http://www.automatalib.net/
4.3. Adapter for the SUL (Step 2)

IImpl interface, to connect with the representation layer and IUsed interface, to connect with the abstraction layer (Figure 4.2). Table 4.1 shows the number of accessible methods in each interface.

<table>
<thead>
<tr>
<th></th>
<th>IImpl</th>
<th>IUsed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessible</td>
<td>1742</td>
<td>1292</td>
</tr>
</tbody>
</table>

Each of the two interfaces (IImpl and IUsed) consists of two parts and each part has a source layer and a target layer. There are four categories of events based on that separation. This is drawn by the arrow in Figure 4.2. (1) Incoming IImpl interface which is sourced from the representation layer and is directed to the functional layer. (2) Outgoing IUsed interface which is sourced from the functional layer and is directed to the abstraction layer. (3) Incoming IUsed interface which is sourced from the abstraction layer and is directed to the functional layer. (4) Outgoing IImpl interface which is sourced from the functional layer and is directed to the representation layer.

To learn the behavior of the SUL regarding the interaction between the functional layer and the other layers, an adapter application was created such that the interactions between layers through interfaces were captured. Therefore the adapter was connected with both the representation layer and the abstraction layer. Furthermore, the inputs and outputs for the model learning were specified. Table 4.2 which corresponds with Figure 4.2 shows how the interfaces are utilized as either input or output for the learner.

Based on the observation of the interaction between layers in the PandB software component, an input to the SUL can lead to an output to the abstraction layer, or an output to the representation layer, or a combination of these outputs, or no output at all. Moreover, input may come from the representation layer or from the abstraction layer. In other words, one input presented as arrow 1 or arrow 3 in Figure 4.2 could return no output, outputs as arrow 2 and/or arrow 4. Table 4.3 shows some examples of this layers interaction through interfaces.
Table 4.2: The interface utilization as the input or output for the learner which corresponds with Figure 4.2

<table>
<thead>
<tr>
<th>Arrow</th>
<th>Interface name</th>
<th>As input or output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IImpl</td>
<td>Input</td>
</tr>
<tr>
<td>2</td>
<td>IUsed</td>
<td>Output</td>
</tr>
<tr>
<td>3</td>
<td>IUsed</td>
<td>Input</td>
</tr>
<tr>
<td>4</td>
<td>IImpl</td>
<td>Output</td>
</tr>
</tbody>
</table>

Table 4.3: Some examples of the interaction which occurs between layers through interfaces. The numbers in this table correspond to the arrows depicted in Figure 4.2

<table>
<thead>
<tr>
<th>Input and outputs (Arrow)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: 1</td>
</tr>
<tr>
<td>Output: 2</td>
</tr>
<tr>
<td>Input: 1</td>
</tr>
<tr>
<td>Output: 4</td>
</tr>
<tr>
<td>Input: 1</td>
</tr>
<tr>
<td>Outputs: 2, 4</td>
</tr>
<tr>
<td>Input: 3</td>
</tr>
<tr>
<td>Output: 2</td>
</tr>
<tr>
<td>Input: 1</td>
</tr>
<tr>
<td>No output</td>
</tr>
</tbody>
</table>

To capture which interfaces are called in response to a certain interface call, a singleton code pattern was created in the adapter application for collecting the outputs. A singleton pattern is used because it gives the capability to collect the outputs to both layers around the functional layer as a collection of outputs. This is convenient for collecting multiple outputs with different directions. Once an interface is called either from IImpl or IUsed, this interface call is observed as output in the collection. Moreover, to observe an interface call, we removed the body of the method in the outgoing IImpl interface and outgoing IUsed interface and saved the observation in the output collection.

To connect the LearnLib library with the SUL, we need to develop an adapter application. A socket connection of TCP/IP communication was used for the connection between the learner and the SUL (Figure 4.3). In response to a particular input from the learner, the adapter application will do the actual interface call to the SUL and capture the output from the SUL response. This response is sent back to the learner by the adapter application as the output for the learning process.
One important part of the model learning is a reset command which is a command to bring back the system to its initial state. In a case of the functional layer, when the system is running, the reset command should remove the object from the memory and re-initialize the object again so that the functional layer can return to its initial state. At the beginning of the experiment, it became clear that this way of reset is not adequate to bring the functional layer to its initial state. This reset results in the non-determinism problem. We refer to Section 4.7 for more details about this.

4.4 Functional layer initialization

During the experiments it turned out that proper initialization of the functional layer is essential. Without proper initialization, we encountered a run-time error in the adapter application when making actual interface calls. This is because, for instance, the actual interface call needs a particular object which, without object initialization, will not exist in memory. For a legacy software component, it is not so clear what proper initialization means and how to achieve it.

4.5 Trace files from system test cases

PandB software component has existing system test cases for regression testing. These test cases generate some trace files when the test is run. These trace files describe the sequence of interface calls as well as the object initialization occurred during the test. The collected trace files are useful to obtain insight on how different layers in the software component interact with each other and what data is sent from one layer to the other. In addition, the availability of existing trace files can be useful as a guideline to understand which objects in the functional layers should be initialized and in which order. By following these trace files, a proper initialization of the SUL can be performed.

4.6 Handle multiple outputs

As mentioned earlier, there is a possibility that one input given to the SUL will return multiple outputs. Given this situation, the learner has to know when it should stop receiving the expected outputs. Once the input is sent to the SUL, the output observed is not immediately sent back to the learner. Instead, the output will be saved in a collection of outputs temporarily. When the expected outputs are completed by a reply to the input, the adapter can send them as a collection of outputs to the learner and the learner will process them. Before giving new input stimuli to the SUL, this collection of outputs from the previous input stimuli will be discarded. In this way, the learner can handle the outputs properly for every input to the SUL even when it contains multiple outputs (or no output). Furthermore, in a case of multiple outputs, the order of the output collection observed by the adapter application should be the same order as the output collection sent to the learner. For instance, when the learner sends an input sequence $a$ to the SUL via the adapter application, then the adapter application observes what the outputs are from the SUL. Suppose the output sequence collected is $c d$, then the output sequence sent back to the learner should be in the same order $c d$. The reason is because the construction of a model of a Mealy machine distinguishes single input and single output so sequence $c d$ is different from sequence $d c$. 
4.7 Handle non-determinism

At the beginning of the model learning experiments, the learner could not run properly because of the non-deterministic behavior of the SUL. Non-determinism was caused by a single input which gave two different outputs in different queries. For instance, an input sequence \( a b \) gives output sequence \( c \) in the first observation but in the next observation a similar input sequence gives output sequence \( c \ d \). This results in inconsistency in the learning process and no model could be constructed. A cause of this inconsistency is the use of a singleton code pattern in the functional layer where values or class objects are shared within the application. The use of re-initialization of the adapter object was not enough to reset the whole SUL because the shared objects were not removed from the memory completely. A solution of this issue is to use another method to reset the SUL by exiting the running adapter process in the operating system and starting the process of the SUL again. This way of reset is executed in the learner.

4.8 Stateless Mealy machine

After solving the previous problems, the first learning experiments showed that the preliminary model generated from the Learner contains only one state. There are three reasons for this situation. The first reason is the lack of input stimuli, so we need to add more input stimuli. The second reason is that some errors occur in the adapter application that we need to solve. For instance, when the adapter application runs the actual interface call, a run-time error occurs and makes that the adapter application crashes. In this situation, it is a necessity to prevent the error in every interface call because one interface call might contain an essential process in the running software component, for example, to check the preconditions or to update the state to allow an individual process in the software component.

There are two reasons for these errors. The first reason of the errors is the lack of initialization in the functional layer as we mention in Section 4.4. The initialization is not only about object creation in the adapter application, for example, by calling the constructor but it also needs to run method "initialize" in the object. The second reason for these errors is that some calls from the functional layer to the abstraction layer require return class objects. To prevent a run-time error in the adapter application, the creation of dummy object class is needed in the abstraction layer so that the running adapter application can continue. In some cases, the object classes are not trivial to initialize because of the inherited classes in which we have to initialize some objects which are not part of the SUL.

The third reason of the one-state Mealy machine is the use of single parameter data in the interface calls. Some methods in the IImpl and the IUsed interfaces require one or more data parameters for the actual interface call. These parameters are used to update values or state inside the SUL. Some examples of the parameters are boolean type value, enumerate type value, and event handler object values. The learning process requires a combination of all possible values in the parameters to make a model representing a complete behavior of the SUL. Therefore, every interface call which requires parameter(s) will need to use all possible parameter values.

To deal with parameter data, we need to create an encoding so that the inputs from the learner can bring information about parameter data. In other words, the parameter data will be encoded in a string of input from the learner. Later, the
encoded input sequence can be translated into an actual interface call in the adapter application. The encoding will be explained in the next section.

## 4.9 Encoding in input and output

The adapter was created to do the actual call to the interfaces in the SUL. Since each interface call has its direction to the SUL, the generated model needs some information to distinguish the difference between, for instance, the input that is coming from either IImpl interface or IUsed interface. Later, this information is required to produce new code from the model. Therefore, every input and output to the learner contains information about the direction and in which interface it occurs. By adding a prefix to every input and output, the transitions in the generated model will have the information about the direction and the interface it belongs to.

Since LearnLib can only handle string data type as input and output, we need to encode the property as a prefix in the string. Figure 4.4 shows the encoding of the property in input and output string.

![Figure 4.4: The encoding of the interface property in the input and output string](image)

For generating a new code implementation, we also need information about the class name and the method name. Together with the parameter data, we encoded the complete input and output string as follows:

```
[interface_name]_[direction]_[class_name]_[method_name]_[parameter_data]
```

Table 4.4 shows some examples of input and output using the defined encoding.

<table>
<thead>
<tr>
<th>Input and output examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Impl_In_FuncDRAStateHandler_Execute_ENDPOSSTORED&quot;</td>
</tr>
<tr>
<td>&quot;Impl_In_FuncDRAStateHandler_Execute_STARTPOSSTORED&quot;</td>
</tr>
<tr>
<td>&quot;Impl_Out_CPBGenOverallUIModel_SetAPCReadyForRecall&quot;</td>
</tr>
<tr>
<td>&quot;Used_In_FuncDRAManager_HandleEvent_CPBAbstEventScenarioStarted&quot;</td>
</tr>
<tr>
<td>&quot;Used_Out_CPBAbstEventAMRStatusChanged_GetAMRStatusFulfilled&quot;</td>
</tr>
</tbody>
</table>
4.10 Learned model

A model produced by LearnLib is expressed in the GraphViz grammar in which the textual representation is expressed in a file with file extension "dot". This file contains a Mealy machine representation of the model which is given by states and transitions. Listing 4.1 shows an example of a learned model in a dot file. This model is the outcome from model learning the DRA manager. As can be seen in Listing 4.1, every transition in the model contains information about input and outputs in its label. Input information is denoted by "In" text and output information is denoted by "Out" text. In addition, an input which has no output is denoted by an "Empty" text. Also, the information about the interface location in the model is straightforward because every action in inputs or outputs contains the interface location information denoted by the "Used" or "Impl" text.

Listing 4.1: A fragment of a dot file in GraphViz syntax representing a model of the DRA manager

```
digraph g {
  _ _ start0 [label="" shape="none "] ;
  s0 [shape="circle " label="s0 "];  
  s1 [shape="circle " label="s1 "] ;  
  ...  
  s0 -> s1 [label="Impl_In_FuncDRAManager_HandleStateEndPosNotDefined /
           Used_Out_CPBAbsManagerSingleton_GetDataModelFacade ;
           Impl_Out_CPBGemOverallUIModel_SetAPCPositionStoredAStatus ;
           Impl_Out_CPBGemOverallUIModel_SetAPCPositionStoredBStatus ;
           Used_Out_CPBAbsManagerSingleton_GetGeometryFacade ;
           Used_Out_CPBAbsGeometryFacade_GetUIMessageListObserver ;
           Used_Out_CPBAbsGeometryFacade_CreateUIMessageListObserver ;
           Used_Out_CPBAbsUIMessageListObserver_Initialize ;
           Used_Out_CPBAbsObserverBase_Initialize ;
           Used_Out_CPBAbsUIMessageListObserver_ActivateUIMessageID_PBGenXCNEnum ;
           Used_Out_CPBAbsUIMessageListObserver_GetUIMessageList ;
           Used_Out_CPBAbsUIMessageListObserver_GetUIMessageList [" "] ;
           s0 -> s0 [label="Used_In_FuncDRAManager_HandleEvent_CPBAbsEventLockPreempted /
            Used_Out_CPBAbsManagerSingleton_GetGeoAcquisitionScenarioFacade ;
            Used_Out_CPBAbsGeoLogicalResourceBase_IsLocked [" "] ;
           ...  
           _ _ start0 -> s0 ;
}
```

Figure 4.5 illustrates the graph representation of the dot file in Listing 4.1. There are six states in this graph representation in which the initial state is drawn in green. Also, to simplify the graph representation, some self-transitions are not shown and not all labels of transitions are displayed. For the results of the other DRA functionality, we refer to Chapter 7.
Figure 4.5: Graph representation from the learned model of the dot file in Listing 4.1
Chapter 5

Model transformations

This chapter explains the approach to transform a GraphViz model into another model from which code can be generated. Moreover, the code generation itself is described.

5.1 Transformation from model to a DSL instance (Step 3)

An intermediate step is required to generate a new code implementation from a learned model of the legacy implementation (Figure 5.1). The intermediate transformation includes the creation of a DSL grammar which can translate a model generated by LearnLib into other models, for instance, an mCRL2 (micro Common Representation Language 2) model and a model in an interface language framework named ComMA (Component Modeling and Analysis). We created a DSL grammar named dotDSL to define these model transformations. Figure 5.2 depicts the class diagram design of the dotDSL grammar.

By using this dotDSL grammar, a learned model from LearnLib is utilized as an instance of dotDSL. Listing 4.1 demonstrates an example of a learned model which will be converted to a ComMA model and an mCRL2 model.

A model generator was developed for the dotDSL grammar to produce an interface specification in ComMA. The ComMA framework was successfully applied to describe the behavior of interfaces of software components in industrial applications [46]. By using ComMA, engineers can define the signatures of interfaces as...
Chapter 5. Model transformations

The model transformation involves a mapping of incoming and outgoing interface from the DSL instance to either commands or notifications in ComMA. Commands and notifications are different in a way that commands can be called synchronously from client to server and notifications are designed to be sent asynchronously from server to client. In the functional layer, the `HandleEvent` method is categorized as a callback from the abstraction layer (incoming IUsed interface). Therefore, it is classified as a notification in the IUsed interface in ComMA. Figure 5.3 illustrates the mapping of the incoming and outgoing interface for both the IImpl and IUsed interface from the DSL instance to the commands and notifications specified in ComMA.

An instance of dotDSL produces two files: a file containing the interface signatures as groups of commands and notifications (Listing 5.1), and a file containing interface behavior specification (Listing 5.2). As can be seen in the interface signatures (Listing 5.1), there are two interfaces produced: IImpl interface and IUsed interface. All information about the action in inputs or outputs from the dotDSL instance is classified into one of these two interfaces by using a mapping mentioned earlier.

Listing 5.2 shows the software component behavior in a state machine containing

---

**Figure 5.2:** The dotDSL abstract syntax to translate GraphViz model to ComMA model and mCRL2 model

---
states, transitions, and triggers. Transition bodies require a trigger, actions, and a next state. All transition bodies are constructed from the state machine in the dotDSL instance. In Listing 5.2, there is an example of a transition with some actions and a transition with no action. Also, there is a transition with a self-loop and a transition between different states.

Listing 5.1: Interface signatures in ComMA transformed from dotDSL instance in Listing 4.1

```java
interface IImpl1 {
    commands
    void FuncDRAManager_HandleStateEndPosNotDefined
    void FuncDRAManager_StopExposure
    ...
    notifications
    CPBGenOverallUIModel_SetAPCPositionStoredAStatus
    CPBGenOverallUIModel_SetAPCPositionStoredBStatus
    ...
}

interface IUsed1 {
    commands
    void CPBAbstManagerSingleton_GetDataModelFacade
    void CPBAbstDataModelFacade_BGOStandOverrideEnable
    void CPBAbstManagerSingleton_GetGeometryFacade
    void CPBAbstGeometryFacade_GetUIMessageListObserver
    void CPBAbstGeometryFacade_CreateUIMessageListObserver
    void CPBAbstUIMessageListObserver_Initialize
    void CPBAbstObserverBase_Initialize
    void CPBAbstUIMessageListObserver_ActivateUIMessageID_PBGenXCNEnum
    void CPBAbstUIMessageListObserver_GetUIMessageList
    void CPBAbstManagerSingleton_GetGeoAcquisitionScenarioFacade
    void CPBAbstGeoLogicalResourceBase_IsLocked
    ...
    notifications
    FuncDRAManager_HandleEvent_CPBAbstEventLockPreempted
    ...
}
```
Chapter 5. Model transformations

LISTING 5.2: A state machine specification in ComMA transformed from dotDSL instance in Listing 4.1

```csharp
import "Dot_interface.if"
behavior machine machine1 provides IImpl1 requires IUsed1 {
    initial state s0 {
        transition trigger: IImpl1::FuncDRAManager_HandleStateEndPosNotDefined
            do: IUsed1::CPBAbstManagerSingleton_GetDataModelFacade
                IUsed1::CPBAbstDataModelFacade_BGOStandOverrideEnable
                IImpl1::CPBGenOverallUIModel_SetAPCPositionStoredAStatus
                IUsed1::CPBAbstManagerSingleton_GetGeometryFacade
                IUsed1::CPBAbstGeometryFacade_GetUIMessageListObserver
                IUsed1::CPBAbstGeometryFacade_CreateUIMessageListObserver
                IUsed1::CPBAbstUIMessageListObserver_Initialize
                IUsed1::CPBAbstObserverBase_Initialize
                IUsed1::CPBAbstUIMessageListObserver_ActivateUIMessageID_PBGenXCNEnum
                IUsed1::CPBAbstUIMessageListObserver_GetUIMessageList
                IUsed1::CPBAbstUIMessageListObserver_GetUIMessageList
                reply next state: s1
        transition trigger: IImpl1::FuncDRAManager_StopExposure
            do: reply next state: s0
        transition trigger: IUsed1::FuncDRAManager_HandleEvent_CPBAbstEventLockPreempted
            do: IUsed1::CPBAbstManagerSingleton_GetGeoAcquisitionScenarioFacade
                IUsed1::CPBAbstGeoLogicalResourceBase_IsLocked
                next state: s0
    }
}
```

5.2 Code generator for new implementation (Step 4)

The goal in this step is to produce new code from the generated interface and state machine specification expressed in ComMA. There is a general requirement that the software component definition in the new software implementation should be the same as the software component definition in the legacy software implementation. For instance, the interface names, the parameters used, and the return values. The reason is that when the legacy implementation is replaced with the new implementation, there is no need to also update the other layers in the software component. In other words, the new implementation can just replace the legacy implementation without having to adapt other layers in the software component.

The ComMA framework contains several tools to support different tasks via model transformations [46]. We developed an additional code generator in the ComMA framework to generate new code from the software component behavior defined in ComMA. The code generator produces two files: one header file and one C++ implementation file. Moreover, the new code was developed following the state code pattern because the software component behavior is expressed as a state machine representation. Also, the singleton code pattern was used in the new code implementation because it needs to share a single value of the current state consistently across the functional layer. Figure 5.4 illustrates the comparison between the legacy and the new code implementation in the functional layer. The difference between the legacy and the new code implementation is that there is an additional class of a state machine in the functional layer and all three DRA classes dependent on this state machine class. Moreover, every method in the three DRA classes utilizes this state machine class to perform interface calls.
5.2. Code generator for new implementation (Step 4)

![Diagram](image1)

(A) The legacy functional layer  
(B) The new functional layer

**Figure 5.4:** The comparison between the legacy and the new functional layer in the PandB software component

The code fragment from the generated header file can be seen in Listing 5.3. Also, the code fragment from the generated C++ implementation file can be seen in Listing 5.4.

**Listing 5.3:** A code fragment of the header file produced by the code generator

```c++
enum Triggers
{
    FuncDRAManager_StopExposure,
    FuncDRAManager_HandleStateEndPosNotDefined,
    ...
};

class CPBFuncBaseState
{
public:
    virtual CPBFuncBaseState* GetNextState(Triggers trigger) = 0;
};
class CPBFuncState0 : public CPBFuncBaseState
{
public:
    virtual CPBFuncBaseState* GetNextState(Triggers trigger);
};

class CPBFuncStateMachine
{
public:
    static CPBFuncStateMachine& Instance()
    {
        static CPBFuncStateMachine instance;
        return instance;
    }
    voidStateChanged(Triggers trigger);
};

private:
    CPBFuncBaseState* m_pCurrentState;
    ...
};
```
Listing 5.4: A code fragment of the C++ implementation file produced by the code generator

```cpp
CPBFuncStateMachine::CPBFuncStateMachine()
{
    m_pCurrentState = new CPBFuncState0();
}

void CPBFuncStateMachine::StateChanged(Triggers trigger)
{
    if (m_pCurrentState)
    {
        CPBFuncBaseState* pState = m_pCurrentState->GetNextState(trigger);
        m_pCurrentState = pState;
    }
}

CPBFuncBaseState* CPBFuncState0::GetNextState(Triggers trigger)
{
    CPBFuncBaseState* nextState = NULL;
    switch (trigger)
    {
        case FuncDRAManager_HandleStateEndPosNotDefined :
        {
            // IUsed1::CPBAbstManagerSingleton.GetDataModelFacade
            CPBAbstDataModelFacade* pCPBAbstDataModelFacade =
                &CPBAbstManagerSingleton::Instance().GetDataModelFacade();
            ...
            nextState = new CPBFuncState1;
        }
        break;
        ...
    }
    return nextState;
}
```

The new code is generated using a pre-defined specification in the code generator. This pre-defined specification is used to specify which C++ action is required to substitute the action in ComMA. As can be seen from Listing 5.4, the code generator substitutes a ComMA action "GetDataModelFacade" with a C++ action. Listing 5.5 shows a fragment of the mapping which is written in a text file. The code generator reads this mapping and substitutes the action in ComMA with the action in C++.

Listing 5.5: Pre-defined specification in the code generator

```plaintext
### IUsed1::CPBAbstManagerSingleton.GetDataModelFacade
= CPBAbstDataModelFacade* pCPBAbstDataModelFacade =
    &CPBAbstManagerSingleton::Instance().GetDataModelFacade();
###
### IUsed1::CPBAbstDataModelFacade_BGOStandOverrideEnable
= pCPBAbstDataModelFacade->BGOStandOverrideEnable(TRUE);
###
```
Chapter 6

Quality of generated code

This chapter describes the equivalence checking approach to verify the equivalence of the behavior of two software implementations. In addition, this chapter explains the software metrics used to measure software maintainability characteristics. The maintainability measurement also includes the subjective assessments from software maintainers.

6.1 Equivalence checking (Step 5)

A previous study has shown the use of an equivalence checker to compare the external control behavior between a refactored software implementation and its legacy software implementation [10]. The goal of this equivalence checking is to get confidence that the two models from two implementations are equivalent. The equivalence checking tool from the mCRL2 tools [47] was used to compare the two models. This tool also provides the functionality to provide a counterexample in a case the two models are not equivalent.

Firstly, we developed additional functionality in dotDSL to transform a GraphViz model into an mCRL2 model (see triangle 5 and 6 in Figure 1.1). By creating this functionality, a Mealy machine expressed in Graphiz model can be transformed automatically into an mCRL2 process. Secondly, model learning is used to produce two GraphViz models both from the legacy implementation and the refactored implementation (Figure 6.1).

Thirdly, the transformation from GraphViz model to mCRL2 model is executed on the dotDSL instances (of both learned models). Fourthly, once the two mCRL2 processes are produced, we used the ltscompare tool in the mCRL2 toolset to do the equivalence checking of these two processes. Various notions of equivalence have been defined in the literature. Based on the arguments of [10], we chose (strong) branching bisimulation equivalence. Furthermore, Figure 6.1 shows an overview on how the equivalence checking is executed.

It was found that sometimes the two mCRL2 models are not equivalent in the first equivalence check. Then, the next step is to find the counterexample and investigate the cause of this counterexample in the new code implementation. There are several reasons for this non-equivalence situation:

1. Missing or incorrect specification in the code generator that substitutes the action in ComMA with the action in the C++ implementation. To solve this problem, we need to check if the C++ action replacement is correct by referring to the legacy implementation.

2. The legacy model contains not only the behavior of the functional layer but also the behavior of the abstraction layer. This was caused by a singleton class in the abstraction layer that calls another method in the abstraction layer so
Chapter 6. Quality of generated code

that the behavior of the abstraction layer was included in the model. Some adjustments were required in the adapter application to observe only the very first interface call from the functional layer to the abstraction layer and not to include the observation that is happening inside the abstraction layer.

3. There is some method overloading implemented in the IUsed interface, for example, some methods have the same name but have different parameters. We found that there is no information to differentiate this method overloading from the action expressed in ComMA. Therefore, this separation needs to be performed earlier in the process by means of the encoding of inputs and outputs into strings.

Since the new software implementation is generated from the ComMA model, some adjustments are needed in the code generator of the ComMA framework. After some improvements in the code generator, the equivalence check is run again until the two mCRL2 processes do not give any counterexample. Listing 6.1 shows an example of an mCRL2 process that is transformed from a GraphViz model. The equivalence checker is helpful to compare the behavior of two implementations because the textual differences of two GraphViz models do not always mean that the behaviors are different.
6.2. Maintainability measurement (Step 6)

The literature contains a practical approach to measure software maintainability which allows root-cause analysis [25]. This approach is named the SIG maintainability model. This maintainability model is framed following the ISO 9126 standard for software engineering product quality. There are some characteristics of maintainability in this ISO standard and the SIG maintainability model provides a mapping that transforms these characteristics into some source code properties (Figure 6.1). Therefore, one can determine which maintainability characteristics are affected by a software system change. The measurement of the source code properties described in this section follows the SIG maintainability model.

**TABLE 6.1: Mapping system of maintainability characteristics onto source code properties [25]**

<table>
<thead>
<tr>
<th>ISO 9126 maintainability</th>
<th>source code properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>analysability</td>
<td>x x x x x x</td>
</tr>
<tr>
<td>changeability</td>
<td>x x x x x x</td>
</tr>
<tr>
<td>stability</td>
<td>x x x x x x</td>
</tr>
<tr>
<td>testability</td>
<td>x x x x x x</td>
</tr>
</tbody>
</table>

As the first step for measuring the source code properties, one should specify all source code properties precisely. Therefore, we define source code properties as follows.
Chapter 6. Quality of generated code

**Volume** is defined as the total size of a system which is calculated by the total lines of code (LOC) without comments.

**Complexity per unit** is calculated using McCabe’s cyclomatic complexity (CC) [48] per unit. In addition, a unit is defined as the smallest part of the software which can be executed and tested independently. In our case, since both the legacy code implementation and the refactored code implementation use C++ code, a unit is a method or a function.

**Duplication** of source code is calculated as the percentage of LOC without comments which are duplicate over the total LOC. Here, code duplication is specified as code blocks of at least six lines which occur more than once.

**Unit size** is calculated as the total LOC without comments per unit.

**Unit testing** is calculated as the percentage of units which are tested or covered by test cases over the total number of units.

For each property, we use a simple scale for ranking: ++ / + / o / - / --. For instance, ranking ++ is better than ranking +, ranking + is better than ranking o, ranking o is better than ranking -, and ranking - is better than ranking --. Also, we used some tools to help to measure these source code properties. The explanation on how to measure each property in more details is presented in the following section.

### 6.2.1 Volume

We used the CCCC\(^1\) tool to calculate the total LOC of both the legacy code implementation and the refactored code implementation. The legacy code implementation includes a header file and a cpp file of the related DRA class. Also, the refactored code implementation includes the same file as the legacy version but with two additional files: a header and a source file of the state machine class. The number of the total LOC generated by the tool is categorized in the ranking in Table 6.2.

<table>
<thead>
<tr>
<th>rank</th>
<th>KLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>0-66</td>
</tr>
<tr>
<td>+</td>
<td>66-246</td>
</tr>
<tr>
<td>o</td>
<td>246-665</td>
</tr>
<tr>
<td>-</td>
<td>655-1310</td>
</tr>
<tr>
<td>--</td>
<td>&gt; 1310</td>
</tr>
</tbody>
</table>

### 6.2.2 Complexity per unit

The CCCC\(^1\) tool was utilized to compute the complexity per unit. Firstly, the complexity for every unit is categorized based on its risk evaluation (Table 6.3). Secondly, we counted the total LOC in each category and computed the percentage for each category. For example, given that a software system with 100 LOC which has 10 LOC with moderate risk, then the percentage of the code with moderate risk is 10%. Next, based on the percentage in every category, we determined the ranking using Table 6.4.

\(^1\)http://cccc.sourceforge.net/
### 6.2. Maintainability measurement (Step 6)

#### Table 6.3: Risk evaluation of cyclomatic complexity per unit [25]

<table>
<thead>
<tr>
<th>CC</th>
<th>risk evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-10</td>
<td>simple, without much risk</td>
</tr>
<tr>
<td>11-20</td>
<td>more complex, moderate risk</td>
</tr>
<tr>
<td>21-50</td>
<td>complex, high risk</td>
</tr>
<tr>
<td>&gt; 50</td>
<td>very complex, very high risk</td>
</tr>
</tbody>
</table>

As can be seen in Table 6.4, the ranking ++ allows 21% of the code with moderate risk and no code with high and very high risk at all. Additionally, to have a ranking +, a software system can have no more than 30% of code with moderate risk, no more than 5% of code with high risk, and no code with very high risk. The lowest ranking -- is given if a software system has more than 50% of the code with moderate risk or more than 15% of code with high risk or more than 5% of code with very high risk.

#### Table 6.4: Ranking of cyclomatic complexity per unit [25]

<table>
<thead>
<tr>
<th>rank</th>
<th>maximum relative LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>moderate</td>
</tr>
<tr>
<td>++</td>
<td>21%</td>
</tr>
<tr>
<td>+</td>
<td>30%</td>
</tr>
<tr>
<td>o</td>
<td>40%</td>
</tr>
<tr>
<td>-</td>
<td>50%</td>
</tr>
<tr>
<td>--</td>
<td>-</td>
</tr>
</tbody>
</table>

### 6.2.3 Duplication

We used the Simian\(^2\) tool to measure code duplication in both the legacy and refactored version. Based on the number of code duplication found by the tool, a percentage of duplication is calculated and the ranking is determined using Table 6.5.

#### Table 6.5: Ranking of code duplication [25]

<table>
<thead>
<tr>
<th>rank</th>
<th>duplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>0-3%</td>
</tr>
<tr>
<td>+</td>
<td>3-5%</td>
</tr>
<tr>
<td>o</td>
<td>5-10%</td>
</tr>
<tr>
<td>-</td>
<td>10-20%</td>
</tr>
<tr>
<td>--</td>
<td>20-100%</td>
</tr>
</tbody>
</table>

### 6.2.4 Unit size

Since the definition of unit size is the total LOC per unit, we used a similar way to calculate the complexity per unit, but the risk evaluation is determined using Table 6.6.

The threshold for determining the ranking is shown in Table 6.7. Based on this table, a software system is ranked ++ if it has no more than 19.5% of code with

\(^2\)http://www.harukizaemon.com/simian/
moderate risk, no more than 10.9% of code with high risk, and no more than 3.9%
of code with very high risk. Moreover, the lowest ranking -- is given if a software
system have more than 45.9% of code with moderate risk or more than 31.4% of code
with high risk or more than 18.1% of code with very high risk.

6.2.5 Unit testing

The BullseyeCoverage tool was used to measure unit test coverage. The measurement
was performed while the existing test cases were running. The percentage of
the code coverage as the outcome from the tool was used to determine the ranking
(Table 6.8).

6.2.6 Sub-characteristics of maintainability

Based on the rankings obtained for all source code properties, we mapped back
the rankings to all sub-characteristics of maintainability according to the ISO 9126
standard (Table 6.1) by aggregation. Every sub-characteristic of maintainability is
computed by using a weighted average according to the cross-marks in the matrix.
Moreover, the weights are equal for every source-code property. This can be illus-
trated in an example of maintainability of the DRA manager in Table 6.9. As can be
seen from the table, the DRA manager has ranking -- for the unit testing. Based on

---

http://www.bullseye.com/
the matrix, this ranking affects the analysability, stability, and testability characteristic of maintainability. Since the only code-level property that affects the stability characteristic is unit testing, stability is ranked --. In addition, the complexity per unit of the DRA manager is ranked ++ and the code duplication is ranked o. Hence, the changeability characteristic is ranked + based on the average between the ranking on the complexity and the ranking on the duplication.

<table>
<thead>
<tr>
<th>ISO 9126 maintainability</th>
<th>source code properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>volume</td>
</tr>
<tr>
<td>analysability</td>
<td>x</td>
</tr>
<tr>
<td>changeability</td>
<td>x</td>
</tr>
<tr>
<td>stability</td>
<td></td>
</tr>
<tr>
<td>testability</td>
<td>x</td>
</tr>
</tbody>
</table>

6.3 Domain expert evaluation

Besides measuring the maintainability characteristics using some source code properties, we also conducted a meeting session to ask some domain experts about the maintainability of the new code implementation in comparison with the legacy version. We invited some software maintainers, the same participant as in the interview sessions. Since one of the participants is no longer available in the company, then we invited the remaining two software designers who work with the PandB software component. Since they have experiences with the software component, this session aims to obtain their perceptions and opinions whether the refactored code is better maintainable. The new code implementation was shown to them and they evaluated based on their judgments. The results of this evaluation will be presented in the next chapter.
Chapter 7

Results

This chapter presents the results of the experiments including the learned models and the equivalence checking results. Additionally, this chapter describes the result from the software maintainability metrics as well as the subjective views from the software maintainers.

7.1 Generated model

This section presents the generated model from the legacy implementation of the DRA functionality including the preparation steps and problems encountered during the experiments.

7.1.1 First experiment

Experiment settings  An adapter application was created to learn a class in the functional layer. We picked one simple class from the functional layer for this initial model learning. The reset strategy was by re-initialize that particular class object (for example, call the destructor and constructor) so that the SUL could back to its initial state.

Results  The result of this model learning was a Mealy machine containing only one state. The one-state Mealy machine was generated because of the lack of input stimuli. Therefore, in the next experiment, the number of input stimuli was added by adding more interface calls for the learning.

7.1.2 Second experiment

Experiment settings  The number of input stimuli increased by adding more interface calls so that the learner could learn more behavior in the functional layer. In the constructor of the adapter application, the software component initialization was executed to ensure the adapter could perform actual interface calls.

Results  Model with three states containing an initial state, an error state and a constructor state (Figure 7.1). From the actual interface calls performed by the adapter application, some runtime errors occur. This can be seen from the generated model as some errors were printed in the output string in the model. One reason of the errors was the existence of uninitialized objects which did not exist in the memory but are required for executing interface calls.
7.1 Third experiment

Experiment settings In this experiment, we fixed more errors that occur in the actual interface calls. An example is the removal of the body of the methods in the abstraction layer. It was found that there are some methods in the abstraction layer contains calls to other method in the abstraction layer and contains some COM calls in which both of these calls are not the part of the intended observation for the model generation. By removing them, we can avoid some errors in the adapter application. Another example was that some function calls in the abstraction layer need a return object. Hence, the return of a dummy object was added to make sure the function calls can continue in the adapter application. A third error was solved by implementing different way of resetting the SUL by shutting down the SUL process in the OS and re-run the SUL process again. This new way of reset was performed because there was a singleton object in the SUL which could not be removed by the previous way of resetting.

Results Figure 7.2 depicts the model generated by the Learner with an initial state in a green circle. Using 79 input events resulted in a model with 16 states. These 79 input events were selected from three DRA classes. It became clear that by removing errors in the function calls, a model can have more states. Therefore, in the next experiment, more errors were solved before learning the SUL again.

7.1.4 Fourth experiment

Experiment settings All errors in the adapter application regarding the DRA functionality were fixed in this experiment. Some experiments were performed to see how the model was generated with different inputs selected from each DRA class. In this experiment, we added a prefix to each input, so the learned model had more information about the source of the interface and the direction of the event. For instance, information about whether the input/output was from the IImpl or IUsed interface.

Results Four models were learned in this experiment. Three models were produced from each of three DRA classes (Figure 7.3, 7.4, 7.5) and one model was produced from a combination of three DRA classes (Figure 7.6). To simplify

---

1COM (Component Object Model) technology in the Microsoft Windows-family of Operating Systems enables software components to communicate
7.1. Generated model

Figure 7.2: Initial learned model of the DRA functionality

the graphical representation of the learned models, the self-transitions are not shown in the figures. In addition, the initial state is drawn in green. Table 7.1 shows the comparison of the number of input stimuli, the number of generated states, the number of intermediate hypothesis, and the learning time of the DRA functionality.

Table 7.1: The results of the learned models of the legacy DRA functionality

<table>
<thead>
<tr>
<th>class name</th>
<th>stimuli</th>
<th>states</th>
<th>intermediate hypothesis</th>
<th>time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRA state handler</td>
<td>61</td>
<td>8</td>
<td>5</td>
<td>2191</td>
</tr>
<tr>
<td>DRA manager</td>
<td>72</td>
<td>7</td>
<td>3</td>
<td>1726</td>
</tr>
<tr>
<td>Special procedure DRA</td>
<td>88</td>
<td>9</td>
<td>4</td>
<td>3440</td>
</tr>
<tr>
<td>A combination of three DRA classes</td>
<td>221</td>
<td>6</td>
<td>2</td>
<td>3653</td>
</tr>
</tbody>
</table>

Problems There were some methods with the same name (method overloading). We added a supplementary prefix of the class name in the input/output string so that the produced state machine in ComMA model can distinguish the difference of methods. In the next experiment, the model will be reproduced again with this additional information encoded in the input/output to the learner.

7.1.5 Fifth experiment

Experiment settings With the additional class name encoded in the prefix of the input/output string to distinguish between methods with the same name.

Results The generated model is the same as in the previous experiment except that all the input/output strings now had information about the class name.
7.1.6 Interface utilization

In both the IImpl and IUsed interface, there are many available methods. Table 7.2 shows the number of methods utilized for model learning in our experiments.

<table>
<thead>
<tr>
<th>interface</th>
<th>utilization</th>
<th>percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>IImpl</td>
<td>111 of 1742</td>
<td>6.4%</td>
</tr>
<tr>
<td>IUsed</td>
<td>86 of 1292</td>
<td>6.6%</td>
</tr>
</tbody>
</table>

7.2 Equivalence Checking

The equivalence check performed for all three DRA classes did not give an equivalent result in the initial experiments. As mentioned earlier in the previous chapter,
7.2. Equivalence Checking

we had to solve some problems to get an equivalent result. This includes the inves-
tigation of the counterexample found and the inspection what the cause of this counterexample is. Most of the time, the cause is the incorrect/missing specification to substitute the ComMA action with C++ code. The counterexample obtained can be applied to the SUL to check if, after the correction, the refactored code will lead to an equivalent result. Finally, after some improvements in the code generation, the outcome from the model checking tools is that for all three DRA classes, the legacy software implementation and its refactored version are equivalent.
7.3 Maintainability metrics results (first experiment)

This section presents the results from the maintainability measurement of both the legacy and the refactored software implementation. All three DRA classes are compared using the source code properties defined in the previous chapter. We present the rankings for all DRA classes in the next sub-sections. For a more detailed calculation, we refer to Appendix B.

7.3.1 DRA state handler

Table 7.3 shows the measurement results from the legacy version of the DRA state handler and Table 7.4 shows the measurement results from its refactored version. As can be seen from both tables, the volume and the unit testing are in the same ranking (−). Therefore, the stability characteristic is also the same. Moreover, the rankings for the other source code properties drop in its refactored version. The complexity, duplication, and unit size code property are ranked − in the refactored implementation. Since those rankings decline, the other maintainability characteristics are influenced. For instance the ranking of analysability declines from ranking o to −, the ranking of the changeability declines from ranking o to ranking --, and the ranking of testability declines from ranking - to --. This means that the analysability, changeability, and testability characteristic of the refactored software implementation are worse than its legacy version.

<table>
<thead>
<tr>
<th>ISO 9126 maintainability</th>
<th>source code properties</th>
<th>volume</th>
<th>complexity per unit</th>
<th>duplication</th>
<th>unit size</th>
<th>unit testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysability</td>
<td>++</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Changeability</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>Stability</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>--</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testability</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

7.3.2 DRA manager

The maintainability results from the legacy implementation of the DRA manager is shown again in Table 7.5 to make the comparison easier with the maintainability results from the refactored implementation in Table 7.6. As can be seen from both tables, similar with the DRA state handler, the volume and the unit testing code property of the DRA manager are in the same ranking between the legacy and the refactored implementation. However, the other three code properties: complexity, duplication, and unit size drop to the lowest ranking -- in the refactored version.
7.3. Maintainability metrics results (first experiment)

Table 7.4: Maintainability of the new implementation of the DRA state handler

<table>
<thead>
<tr>
<th>ISO 9126 maintainability</th>
<th>source code properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>volume</td>
</tr>
<tr>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>analysability</td>
<td>x</td>
</tr>
<tr>
<td>changeability</td>
<td>x</td>
</tr>
<tr>
<td>stability</td>
<td>-</td>
</tr>
<tr>
<td>testability</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 7.5: Maintainability of the legacy implementation of the DRA manager

<table>
<thead>
<tr>
<th>ISO 9126 maintainability</th>
<th>source code properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>volume</td>
</tr>
<tr>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>analysability</td>
<td>x</td>
</tr>
<tr>
<td>changeability</td>
<td>x</td>
</tr>
<tr>
<td>stability</td>
<td>-</td>
</tr>
<tr>
<td>testability</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 7.6: Maintainability of the new implementation of the DRA manager

<table>
<thead>
<tr>
<th>ISO 9126 maintainability</th>
<th>source code properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>volume</td>
</tr>
<tr>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>analysability</td>
<td>x</td>
</tr>
<tr>
<td>changeability</td>
<td>x</td>
</tr>
<tr>
<td>stability</td>
<td>-</td>
</tr>
<tr>
<td>testability</td>
<td>x</td>
</tr>
</tbody>
</table>

7.3.3 Special procedure DRA

Table 7.7 shows the rankings of the source code properties from the legacy version of the special procedure DRA. Additionally, Table 7.8 shows the rankings of source code properties from its refactored version. Similar to the other DRA classes, the refactored software implementation of this special procedure DRA is ranked -- for the complexity, duplication, and unit size property. The ranking drops from ++ to --.
in complexity, from + to -- in duplication, and from - to -- in unit size. As a consequence, the rankings of the analysability, changeability, and testability characteristic drop as well.

### Table 7.7: Maintainability of the legacy implementation of the special procedure DRA

<table>
<thead>
<tr>
<th>ISO 9126 maintainability</th>
<th>source code properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>volume</td>
<td>++ ++ + - --</td>
</tr>
<tr>
<td>complexity per unit</td>
<td>x x x x o</td>
</tr>
<tr>
<td>duplication</td>
<td></td>
</tr>
<tr>
<td>unit size</td>
<td></td>
</tr>
<tr>
<td>unit testing</td>
<td></td>
</tr>
</tbody>
</table>

### Table 7.8: Maintainability of the new implementation of the special procedure DRA

<table>
<thead>
<tr>
<th>ISO 9126 maintainability</th>
<th>source code properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>volume</td>
<td>++ -- -- -- --</td>
</tr>
<tr>
<td>complexity per unit</td>
<td>x x x x -</td>
</tr>
<tr>
<td>duplication</td>
<td></td>
</tr>
<tr>
<td>unit size</td>
<td></td>
</tr>
<tr>
<td>unit testing</td>
<td></td>
</tr>
</tbody>
</table>

#### 7.4 Maintainability metrics results (second experiment)

From the software maintainability measurements in the first experiment, the rankings of the four code properties are low for all DRA classes in the new code. Since we used the state code pattern for the new software implementation and a state is represented as a class, the switch-case statement in a class is utilized to differentiate the triggers in the state. The ranking of the complexity property drops in all DRA cases because of the huge number of these switch-case statements written in the state classes. Moreover, the huge number of lines in switch-case statements leads to a lower ranking of the unit size property. After analyzing the limitation in the new code, it became clear that the improvement in the code generator is possible by removing the use of switch-case statements.

We improved the code generator in the ComMA framework and removed the use of switch-case statements in the state classes by a map of function pointers and separating the C++ actions for every trigger into a class. The fragments of the new code
produced by the code generator after the improvement are presented in Appendix D. The equivalence checking was conducted again to check if the two software implementations describe the same behavior after improving the code generator. Checking confirms that the two software implementations have the same behavior. Next, the maintainability measurements were performed again to see whether the new code generator has better rankings. The results of maintainability metrics after the improvement for the three DRA classes are presented in the following subsections. In addition, we refer to Appendix C for more detailed calculations.

### 7.4.1 DRA state handler

We presented the results of the maintainability measurements of the legacy DRA state handler again in Table 7.9 to compare it with the results from the new code after the code generator improvement in Table 7.10.

#### Table 7.9: Maintainability of the legacy implementation of the DRA state handler

<table>
<thead>
<tr>
<th>ISO 9126 maintainability</th>
<th>source code properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>volume</td>
</tr>
<tr>
<td>++</td>
<td>-</td>
</tr>
<tr>
<td>analysability</td>
<td>×</td>
</tr>
<tr>
<td>changeability</td>
<td>x</td>
</tr>
<tr>
<td>stability</td>
<td></td>
</tr>
<tr>
<td>testability</td>
<td>x</td>
</tr>
</tbody>
</table>

**Table 7.9: Maintainability of the legacy implementation of the DRA state handler**

#### Table 7.10: Maintainability of the new implementation of the DRA state handler after the code generator improvement

<table>
<thead>
<tr>
<th>ISO 9126 maintainability</th>
<th>source code properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>volume</td>
</tr>
<tr>
<td>++</td>
<td>+</td>
</tr>
<tr>
<td>analysability</td>
<td>x</td>
</tr>
<tr>
<td>changeability</td>
<td>x</td>
</tr>
<tr>
<td>stability</td>
<td></td>
</tr>
<tr>
<td>testability</td>
<td>x</td>
</tr>
</tbody>
</table>

**Table 7.10: Maintainability of the new implementation of the DRA state handler after the code generator improvement**

As can be seen from two tables, the new code has better rankings in both of the complexity and unit size properties. In contrast, the ranking in the duplication property drops from + to o. In addition, the rankings from the other code properties remain the same. As a consequence of these rankings, the changeability and testability characteristics improve, and the analysability and stability characteristics remain the same.
7.4.2 DRA manager

Table 7.11 shows the result of the maintainability measurements from the legacy DRA manager and Table 7.12 shows the results from the new code after the improvement in the code generator.

<table>
<thead>
<tr>
<th>ISO 9126 maintainability</th>
<th>source code properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>volume</td>
</tr>
<tr>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>analysability</td>
<td>x</td>
</tr>
<tr>
<td>changeability</td>
<td>x</td>
</tr>
<tr>
<td>stability</td>
<td>x</td>
</tr>
<tr>
<td>testability</td>
<td>x</td>
</tr>
</tbody>
</table>

As shown in two tables, the code duplication property is the only ranking that changes and the other code properties remain the same. The code duplication property has the lowest ranking -- in the new code. The ranking drops from o to --. This situation affects the lower rankings in the changeability characteristic in the new code. In addition, the other characteristics in the ISO standard remain the same.

7.4.3 Special procedure DRA

Table 7.13 shows the results from the maintainability measurements from the legacy special procedure DRA and Table 7.14 shows the results from its refactored version after the improvement in the code generator.

As can be seen from Table 7.13 and Table 7.14, the code property rankings remain the same except the code duplication property. The ranking in the duplication property drops from + to the lowest ranking --. This leads to the lower ranking in the analysability and changeability characteristics in the ISO standard.
TABLE 7.13: Maintainability of the legacy implementation of the special procedure DRA

<table>
<thead>
<tr>
<th>ISO 9126 maintainability</th>
<th>source code properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>volume</td>
</tr>
<tr>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>analysability</td>
<td>x</td>
</tr>
<tr>
<td>changeability</td>
<td>x</td>
</tr>
<tr>
<td>stability</td>
<td>x</td>
</tr>
<tr>
<td>testability</td>
<td>x</td>
</tr>
</tbody>
</table>

TABLE 7.14: Maintainability of the new implementation of the special procedure DRA after the code generator improvement

<table>
<thead>
<tr>
<th>ISO 9126 maintainability</th>
<th>source code properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>volume</td>
</tr>
<tr>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>analysability</td>
<td>x</td>
</tr>
<tr>
<td>changeability</td>
<td>x</td>
</tr>
<tr>
<td>stability</td>
<td>x</td>
</tr>
<tr>
<td>testability</td>
<td>x</td>
</tr>
</tbody>
</table>

7.5 Domain expert assessments

This section contains observations obtained from the assessments by the software designers. By reviewing the new software implementation, they gave the following remarks:

1. Software tracing or logging functionality is very helpful from their point of view. Logging functionality will record the current state and the activities performed in the running software component. Since the new software implementation is produced by the code generator from the learned model, adding logging functionality in the code generator is relatively easy because one can place a piece of code for logging in every generated C++ action.

2. The software maintainers asked whether the ComMA model or the generated code should be maintained. In the case of maintaining the ComMA model, then it will add extra effort to maintain and understand the models next to maintaining the existing software component.

3. It is important that names are meaningful. They saw difference between class names in the conceptual explanation and the new code implementation of the state machine. For instance, in the initial refactored code, we used name "machine" instead of "state machine" as the class name. These names are modified in the later version of the refactored implementation following their suggestions and the existing code convention.
4. There are a lot of unnecessary code duplications in the C++ actions as the result from the substitution of actions in ComMA (see Chapter 5.2 in page 36). These C++ actions can be removed in the refactored code because most of them are actions to get an object from the abstraction layer. These object calls can be performed only once. Since the goal of this study is to learn the utilization of the IImpl and IUsed interfaces and these object calls utilize these interfaces, these object calls are also part of the learning results. Although removing duplication of object calls is relatively easy in the code generator in ComMA by checking if an object call already occurs during C++ action substitution, this will result in a refactored software implementation that is not equivalent to its legacy version. To demonstrate this, we performed the equivalence checking to see whether eliminating the duplication of object calls in the new software implementation is feasible. As a result, the different specification in the software component behavior in ComMA leads to refactored code that is not equivalent to the legacy version.

5. By viewing the refactored implementation from a state machine perspective, on the one hand, the code becomes harder to understand especially if the number of states increases. On the other hand, by using model learning, the state machine representation is explicitly present in a state machine class of the refactored implementation whereas it is originally embedded in its legacy version.

7.6 Discussion of results

As mentioned in [44], one of the challenges of implementing model learning in an embedded system is to establish the connection of the membership queries to the SUL. At the beginning of the experiments, we encountered some problems that we had to solve to establish the connection of the learner to the SUL. Since the PandB software component is complex and it is difficult to decouple one layer from the others (based on the interviews), learning a subset of the software component (the functional layer) becomes even more difficult. Once the connectivity is established, the effort to scale the learning, for instance adding stimuli, becomes relatively easy.

In the initial experiments, we included the initialization of the PandB software component as the part of model learning. For instance, by adding the constructor methods and "initialize" methods as input stimuli for the learner. Later, it became obvious that this initialization was not interesting to learn and gave some problems for the new software implementation generation. Therefore, the initialization part is no longer part of learning and the adapter application performs the software component initialization before the learning process.

Furthermore, for each code property from the software metrics, there are several remarks as follows:

- volume; As can be seen from Appendix C, the number of lines of code of the new implementation increases because of the two additional files containing the state machine. Since the number is still in the same range of the threshold, the ranking remains the same.
- complexity per unit; The improvement in the code generator by removing the use of switch-case statements improves the ranking in the complexity property. Hence, the new code is less complex than its legacy version.
- duplication; Since an input to the learner can give multiple outputs and these outputs are converted into C++ actions during code generation, this situation
can lead to code duplication in the new code. Especially, if two output sequences have the same order. For instance, an output sequence produced by an input is a subset of another output sequence from different input. If these output sequences have the same order and the generated C++ actions contain more than 5 lines, then this is categorized as code duplication in the code metrics.

- **unit size:** The improvement in the code generator also affects the ranking improvement in the unit size property in the DRA state handler class. However, when the number of input stimuli increases, in the DRA manager and special procedure DRA, the ranking in the new code is the same as its legacy version. There are two reasons for this. Firstly, it is because several triggers in the states have a big number of actions. These actions are sourced from the outputs observed by the adapter application. Secondly, the use of a map of function pointers affects the number of lines in the constructors of every state class. If the number of triggers increases, then the number of lines to register function pointers increases as well.

- **unit testing:** Executing the existing test cases on the legacy implementation give the same results as executing them on the new implementation for all three DRA classes. However, late in this research, we found out that the execution of the test cases stops before performing the DRA test cases. This might be caused by the errors that occur in the test cases and lead to an unobserved exit of the test cases. By not executing the test cases specifically for the DRA functionality, the unit testing of the three DRA classes is ranked -- in the legacy version. Similarly, running the same test cases on the refactored implementation result in ranking --. Because of time limitations, we were only able to run the DRA test cases on the legacy version. For each of the DRA classes, this leads to ranking + in the unit testing property.
Chapter 8

Conclusions

8.1 Conclusion

In this study, a methodology to refactor legacy software is presented by applying a combination of model learning, model transformations, and equivalence checking. By using this combination of approaches, a new software implementation can be produced from its legacy version. Moreover, the behavior of the new software implementation is equivalent to its legacy version. The initial questions from Section 1.6 are addressed as follows:

(RQ1)  a) What are the maintainability challenges with the legacy software components in IGT Systems?
The challenges faced to preserve legacy software in industrial settings were investigated by using qualitative analysis from the interviews. According to the investigation, there are several difficulties found in maintaining legacy software. A limited amount of documentation containing a technical description of how the software component works with other software components is one of the challenges. As time goes, the software environment is changing and the software architecture becomes different from one in the past. New hardware support is one of the reasons for changes in the software environment. Since the code-base is already complex, the maintainers have to understand a very big scope of the software component and the maintenance activity becomes time-consuming. According to the interviews, the problems of the legacy software is not only caused by its huge size but also by the many code design patterns applied. Consequently, the code is difficult to understand. Furthermore, maintaining different software branches was mentioned as one of the challenges because there exist multiple software branches for different releases.

b) What are the benefits (or disadvantages) of developing a new implementation compared with keeping the legacy software?
The benefits of developing a new implementation for a part of the software component is that it can support new features which could not be added in the old software design easily. The disadvantages include the huge resources required to develop new software, because documentation is limited and a lot of knowledge is embedded in the legacy software.

(RQ2)  a) Which steps are required to implement model learning techniques to produce a model which is an instance of a DSL?
To implement model learning, an adapter application was created to
Chapter 8. Conclusions

connect the learner to the SUL. For learning a part of a software component, the adapter application runs the initialization of the software component before the learning process. The creation of the dotDSL language is required to interpret the results of model learning as an instance of a DSL.

b) How can model learning and DSLs be combined in an effective way?

The dotDSL language was created to perform the transformation of the outcome from model learning into other models. To combine model learning and DSLs, the dotDSL requires the capability of capturing the GraphViz syntax. Also, the code generator created in the dotDSL can perform the transformations from GraphViz model to ComMA model and mCRL2 model.

(RQ3) a) How can software maintainability be characterized in the context of IGT Systems?

The investigation from the interviews shows that the use of the existing static code analysis, such as cyclomatic complexity and lines of code, is very beneficial to assess software maintainability. In addition, the SIG maintainability model was applied to measure the code-related property of both the legacy and new software implementations. Equally important, the assessment from domain experts was performed to obtain their judgments of the maintainability of the new software.

b) What steps are required to implement a source code generator using a DSL that produces maintainable source code from legacy component?

By creating the dotDSL, the GraphViz model from learning can be utilized as a dotDSL instance. Then, this DSL instance performs a transformation of this GraphViz model into a ComMA model. A code generator was created in the ComMA framework to produce new code implementation. To produce maintainable source code, some improvements are needed in this code generator.

c) Is the generated source code from the DSL better maintainable than the legacy implementation?

In addition to the equivalence result, the maintainability metrics show that the new code has a better ranking in the complexity property. Removing a huge number of switch-case statements is a key to achieve this. The greatest limitation of the new code is that the percentage of the code duplication increases. Future version of the code generator may reduce duplicate code and improve the rankings in these maintainability metrics (see Section 8.3).

8.2 Recommendations

Regarding the results from the maintainability metrics, maintaining the ComMA model is an alternative for maintaining the refactored software implementation. Then, the maintainability metrics could be applied to ComMA models. This means it is not necessary to run maintainability metrics in the generated code anymore; only the ComMA models are maintained. Hence, it may improve the results of the maintainability measurements.

The participants of the interviews provided several suggestions on how to deal with legacy software. Firstly, the process of refactoring legacy software that is still ongoing should be continued, for example, the activities to reduce code size and to
remove unused code. Secondly, one suggests to keep adding new backlogs on which activities have to be performed in the future. In this way, new personnel knows what and where to improve in the legacy software.

8.3 Future work

As future work, improving the code generator is required, especially to reduce code duplication. The use of template functions and template classes may reduce code duplication. In addition, the use of template constructors may reduce unit size in the constructors of the state classes. Another future direction is to implement different behavioral code design patterns to improve the results in maintainability characteristics. Furthermore, future work includes the execution of the complete test cases including the test cases for the DRA functionality as the part of maintainability measurements.

For model learning with a big number of input stimuli, the selection of testing algorithm to find a counterexample is crucial. During our experiments of learning the combination of the three DRA classes with 221 input stimuli, the use of Wp-method algorithm has given 19 states as the intermediate hypothesis. The result from this testing algorithm is different with the result with the random walk algorithm that gave only 6 states as the final model (Table 7.1). A study presented the importance of test strategy selection for finding counterexample [50]. Therefore, further work includes the investigation of models produced with different testing algorithms.

Concerning model transformations, the learned model might be transformed into a hierarchical finite-state machine for better understandability. A hierarchical finite-state machine is a finite-state machine where a state may contain another finite-state machine. Since the number of states increases for learning a complex legacy software, hierarchical finite-state machine will group states into a composite state. This is a very useful construct to structure and represent large systems. Further study is needed to investigate the approach of constructing a hierarchical finite-state machine from an ordinary (flat) finite-state machine.
Appendix A

Interview questions

A.1 Software designer

Target participants: the maintainers of the legacy software in the context of positioning software

Goal: to answer the question "what are the problems with the legacy software?" and to investigate what the maintainers perspective are

These questions are about the maintainability aspects of existing systems
1. What software/systems are you working right now?
2. How long have you been working with the current software?
3. What is your role?
4. What are technologies used?
5. Are you familiar with those technologies (at the beginning of the project and now)?
6. How long do you think the learning curve to get familiar with this software?
7. Do you find it is difficult to get familiar with the current software pattern?
8. Do you know how many lines or a number of functions that are exist in this software?
9. If there is a defect in the software, how long it takes to fix it (in average)?
10. How easy to fix the defect without causing an error in some other part of the software?
11. Are there new features in the development at the moment?
12. How easy to develop new feature?
13. How to measure the current maintainability of the system?
14. Are there difficulties to maintain the system?
15. What are the maintainability criteria that do you think you can use to assess the software? For example, LOC, Unit Size, duplication, unit complexity, etc. are something can be measured but are there something you consider (maybe cannot be measured by tool) but that are very important for the maintainability?
16. Do you think the current software is a well-maintainable? Why?
17. What are important aspects that will enhance the maintainability?

A.2 Manager

Target participants: people from management in the context of IGT Systems (to explore what the problems are in this business unit and how to enhance future maintainability)
Appendix A. Interview questions

Goal: to answer the question "what are the benefits to keep the legacy software?"
This means only doing bug fixing, without adapting with new technology/new implementation.

Interview questions:
1. What software/systems are you responsible with right now?
2. How long have you been responsible with those systems?
3. Do you know how old are those systems? Are they still in development stage or maintenance stage? How many percentage
4. How many people maintain this software?
5. Are there any drawbacks/disadvantages with the current system? For example, to meet business requirements.
6. Do you think the current systems are flexible to adapt with new requirements?
7. What are the maintainability aspects do you think that are important from management point of view?
8. What are the benefits to keep the legacy software? Can you explain to me on how much is the cost from the company to maintain them?
9. How much are resources or time do you need to keep the legacy software running over time?

Goal: to answer the question "what are the benefits to introduce new technology?"

Interview questions:
1. Is the IGT Systems trying to invest to a new technology in the context of refactoring the legacy software?
2. Have they ever been refactored/replaced with new technology?
3. How often that the current legacy software is refactored? For example, when there is a new technology or a new way to make it efficient.
4. How much is the investments spend by the company to implement new implementation/technology?
5. How much are resources (time) needed to refactor the legacy component using that new technology? Do you have a (rough) calculation on it?
Appendix B

Maintainability measurements

B.1 DRA state handler

| Legacy | PBFuncDRAStateHandler.h | 815 | 32 | 11 |
| Legacy | PBFuncDRAStateHandler.cpp | | | |
| Percentage of all | | | | 4 |
| New | PBFuncDRAStateHandler.h | 3638 | 810 | 18 |
| New | PBFuncStateMachine.h | | | |
| New | PBFuncStateMachine.cpp | | | |
| Percentage of all | | | | 22 |

Table B.1: Comparison of the volume, duplication, and unit testing property between the legacy and the new implementation of the DRA state handler

| CC per unit | Legacy | New |
| LOC | Percentage | LOC | Percentage |
| low (1-10) | 416 | 51 | 345 | 9 |
| moderate (11-20) | 399 | 49 | 302 | 8 |
| high (21-50) | 0 | 0 | 0 | 0 |
| very high (>50) | 0 | 0 | 2991 | 82 |

Table B.2: Comparison of the cyclometric complexity property between the legacy and the new implementation of the DRA state handler
### Table B.3: Comparison of the unit size property between the legacy and the new implementation of the DRA state handler

<table>
<thead>
<tr>
<th>Unit size</th>
<th>Legacy</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number</td>
<td>Percentage</td>
</tr>
<tr>
<td>low risk (0-30)</td>
<td>416</td>
<td>51</td>
</tr>
<tr>
<td>moderate risk (30-44)</td>
<td>69</td>
<td>8</td>
</tr>
<tr>
<td>high risk (44-74)</td>
<td>231</td>
<td>28</td>
</tr>
<tr>
<td>very high risk (&gt;74)</td>
<td>99</td>
<td>12</td>
</tr>
</tbody>
</table>

### B.2 DRA manager

### Table B.4: Comparison of the volume, duplication, and unit testing property between the legacy and the new implementation of the DRA manager

<table>
<thead>
<tr>
<th>Legacy</th>
<th>volume (LOC)</th>
<th>Duplication (LOC)</th>
<th>Unit testing (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBFuncDRAManager.h</td>
<td>1332</td>
<td>138</td>
<td>14</td>
</tr>
<tr>
<td>PBFuncDRAManager.cpp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percentage of all</td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>New</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PBFuncDRAManager.h</td>
<td>3389</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PBFuncDRAManager.cpp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PBFuncStateMachine.h</td>
<td></td>
<td>1253</td>
<td>20</td>
</tr>
<tr>
<td>PBFuncStateMachine.cpp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percentage of all</td>
<td></td>
<td></td>
<td>37</td>
</tr>
</tbody>
</table>

### Table B.5: Comparison of the cyclomatic complexity property between the legacy and the new implementation of the DRA manager

<table>
<thead>
<tr>
<th>CC per unit</th>
<th>Legacy</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOC</td>
<td>Percentage</td>
</tr>
<tr>
<td>low (1-10)</td>
<td>1260</td>
<td>95</td>
</tr>
<tr>
<td>moderate (11-20)</td>
<td>72</td>
<td>5</td>
</tr>
<tr>
<td>high (21-50)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>very high (&gt;50)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
**B.3 Special procedure DRA**

**TABLE B.6:** Comparison of the unit size property between the legacy and the new implementation of the DRA manager

<table>
<thead>
<tr>
<th>Unit size</th>
<th>Legacy</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOC</td>
<td>Percentage</td>
</tr>
<tr>
<td>low risk (0-30)</td>
<td>726</td>
<td>55</td>
</tr>
<tr>
<td>moderate risk (30-44)</td>
<td>237</td>
<td>18</td>
</tr>
<tr>
<td>high risk (44-74)</td>
<td>273</td>
<td>20</td>
</tr>
<tr>
<td>very high risk (&gt;74)</td>
<td>96</td>
<td>7</td>
</tr>
</tbody>
</table>

**TABLE B.7:** Comparison of the volume, duplication, and unit testing property between the legacy and the new implementation of the special procedure DRA

<table>
<thead>
<tr>
<th>Volume (LOC)</th>
<th>Duplication (LOC)</th>
<th>Unit testing (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Legacy</td>
<td>PBFuncSpecialProcedureDRA.h</td>
<td>1844</td>
</tr>
<tr>
<td></td>
<td>PBFuncSpecialProcedureDRA.cpp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Percentage of all</td>
<td></td>
</tr>
<tr>
<td>New</td>
<td>PBFuncSpecialProcedureDRA.h</td>
<td>7415</td>
</tr>
<tr>
<td></td>
<td>PBFuncSpecialProcedureDRA.cpp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PBFuncStateMachine.h</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PBFuncStateMachine.cpp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Percentage of all</td>
<td>84</td>
</tr>
</tbody>
</table>

**TABLE B.8:** Comparison of the cyclometric complexity property between the legacy and the new implementation of the special procedure DRA

<table>
<thead>
<tr>
<th>CC per unit</th>
<th>Legacy</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOC</td>
<td>Percentage</td>
</tr>
<tr>
<td>low (1-10)</td>
<td>1670</td>
<td>91</td>
</tr>
<tr>
<td>moderate (11-20)</td>
<td>174</td>
<td>9</td>
</tr>
<tr>
<td>high (21-50)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>very high (&gt;50)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
TABLE B.9: Comparison of the unit size property between the legacy and the new implementation of the special procedure DRA

<table>
<thead>
<tr>
<th>Unit size</th>
<th>Legacy</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOC</td>
<td>Percentage</td>
</tr>
<tr>
<td>low risk (0-30)</td>
<td>924</td>
<td>50</td>
</tr>
<tr>
<td>moderate risk (30-44)</td>
<td>290</td>
<td>16</td>
</tr>
<tr>
<td>high risk (44-74)</td>
<td>316</td>
<td>17</td>
</tr>
<tr>
<td>very high risk (&gt;74 )</td>
<td>314</td>
<td>17</td>
</tr>
</tbody>
</table>
Appendix C

Maintainability measurements after improvement

C.1 DRA state handler

Table C.1: Comparison of the volume, duplication, and unit testing property between the legacy and the new implementation of the DRA state handler after the code generator improvement

<table>
<thead>
<tr>
<th>Legacy</th>
<th>PBFuncDRASStateHandler.h</th>
<th>PBFuncDRASStateHandler.cpp</th>
<th>Percentage of all</th>
</tr>
</thead>
<tbody>
<tr>
<td>volume (LOC)</td>
<td>815</td>
<td>32</td>
<td>11</td>
</tr>
<tr>
<td>duplication (LOC)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unit testing (%)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percentage of all</td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>New</th>
<th>PBFuncDRASStateHandler.h</th>
<th>PBFuncDRASStateHandler.cpp</th>
<th>PBFuncStateMachine.h</th>
<th>PBFuncStateMachine.cpp</th>
<th>Percentage of all</th>
</tr>
</thead>
<tbody>
<tr>
<td>volume (LOC)</td>
<td>6185</td>
<td>430</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>duplication (LOC)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unit testing (%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percentage of all</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.2: Comparison of the cyclometric complexity property between the legacy and the new implementation of the DRA state handler after the code generator improvement

<table>
<thead>
<tr>
<th>CC per unit</th>
<th>Legacy</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>Percentage</td>
<td>LOC</td>
</tr>
<tr>
<td>low (1-10)</td>
<td>416</td>
<td>51</td>
</tr>
<tr>
<td>moderate (11-20)</td>
<td>399</td>
<td>49</td>
</tr>
<tr>
<td>high (21-50)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>very high (&gt;50)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
TABLE C.3: Comparison of the unit size property between the legacy and the new implementation of the DRA state handler after the code generator improvement

<table>
<thead>
<tr>
<th>Unit size</th>
<th>Legacy</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOC</td>
<td>Percentage</td>
</tr>
<tr>
<td>low risk (0-30)</td>
<td>416</td>
<td>51</td>
</tr>
<tr>
<td>moderate risk (30-44)</td>
<td>69</td>
<td>8</td>
</tr>
<tr>
<td>high risk (44-74)</td>
<td>231</td>
<td>28</td>
</tr>
<tr>
<td>very high risk (&gt;74)</td>
<td>99</td>
<td>12</td>
</tr>
</tbody>
</table>

C.2 DRA manager

TABLE C.4: Comparison of the volume, duplication, and unit testing property between the legacy and the new implementation of the DRA manager after the code generator improvement

<table>
<thead>
<tr>
<th></th>
<th>volume (LOC)</th>
<th>duplication (LOC)</th>
<th>unit testing (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Legacy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PBFuncDRAManager.h</td>
<td>1332</td>
<td>138</td>
<td>14</td>
</tr>
<tr>
<td>PBFuncDRAManager.cpp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percentage of all</td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>New</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PBFuncDRAManager.h</td>
<td>10018</td>
<td>3285</td>
<td>3</td>
</tr>
<tr>
<td>PBFuncDRAManager.cpp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PBFuncStateMachine.h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PBFuncStateMachine.cpp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percentage of all</td>
<td></td>
<td>33</td>
<td></td>
</tr>
</tbody>
</table>

TABLE C.5: Comparison of the cyclometric complexity property between the legacy and the new implementation of the DRA manager after the code generator improvement

<table>
<thead>
<tr>
<th>CC per unit</th>
<th>Legacy</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOC</td>
<td>Percentage</td>
</tr>
<tr>
<td>low (1-10)</td>
<td>1260</td>
<td>95</td>
</tr>
<tr>
<td>moderate (11-20)</td>
<td>72</td>
<td>5</td>
</tr>
<tr>
<td>high (21-50)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>very high (&gt;50)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
C.3 Special procedure DRA

**TABLE C.6:** Comparison of the unit size property between the legacy and the new implementation of the DRA manager after the code generator improvement

<table>
<thead>
<tr>
<th>Unit size</th>
<th>Legacy LOC</th>
<th>Percentage</th>
<th>New LOC</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>low risk (0-30)</td>
<td>726</td>
<td>55</td>
<td>4472</td>
<td>68</td>
</tr>
<tr>
<td>moderate risk (30-44)</td>
<td>237</td>
<td>18</td>
<td>516</td>
<td>8</td>
</tr>
<tr>
<td>high risk (44-74)</td>
<td>273</td>
<td>20</td>
<td>937</td>
<td>14</td>
</tr>
<tr>
<td>very high risk (&gt;74)</td>
<td>96</td>
<td>7</td>
<td>621</td>
<td>9</td>
</tr>
</tbody>
</table>

**TABLE C.7:** Comparison of the volume, duplication, and unit testing property between the legacy and the new implementation of the special procedure DRA after the code generator improvement

<table>
<thead>
<tr>
<th>Legacy</th>
<th>PBFuncSpecialProcedureDRA.h</th>
<th>volume (LOC)</th>
<th>PBFuncSpecialProcedureDRA.cpp</th>
<th>duplication (LOC)</th>
<th>PBFuncStateMachine.h</th>
<th>PBFuncStateMachine.cpp</th>
<th>unit testing (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1844</td>
<td>80</td>
<td>13</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>New</td>
<td></td>
<td>16147</td>
<td>5912</td>
<td>3</td>
<td></td>
<td></td>
<td>37</td>
</tr>
</tbody>
</table>

**TABLE C.8:** Comparison of the cyclometric complexity property between the legacy and the new implementation of the special procedure DRA after the code generator improvement

<table>
<thead>
<tr>
<th>CC per unit</th>
<th>Legacy LOC</th>
<th>Percentage</th>
<th>New LOC</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>low (1-10)</td>
<td>1670</td>
<td>91</td>
<td>16018</td>
<td>99</td>
</tr>
<tr>
<td>moderate (11-20)</td>
<td>174</td>
<td>9</td>
<td>229</td>
<td>1</td>
</tr>
<tr>
<td>high (21-50)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>very high (&gt;50)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Table C.9: Comparison of the unit size property between the legacy and the new implementation of the special procedure DRA after the code generator improvement

<table>
<thead>
<tr>
<th>Unit size</th>
<th>Legacy LOC</th>
<th>Legacy Percentage</th>
<th>New LOC</th>
<th>New Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>low risk (0-30)</td>
<td>924</td>
<td>50</td>
<td>11408</td>
<td>71</td>
</tr>
<tr>
<td>moderate risk (30-44)</td>
<td>290</td>
<td>16</td>
<td>366</td>
<td>2</td>
</tr>
<tr>
<td>high risk (44-74)</td>
<td>316</td>
<td>17</td>
<td>1861</td>
<td>12</td>
</tr>
<tr>
<td>very high risk (&gt;74)</td>
<td>314</td>
<td>17</td>
<td>2512</td>
<td>16</td>
</tr>
</tbody>
</table>
Appendix D

Code implementation after improvement

LISTING D.1: A code fragment of the header file produced by the code generator after improvement

```cpp
enum Triggers {
    FuncDRAManager_StopExposure,
    FuncDRAManager_HandleStateEndPosNotDefined,
    ...
};

class CPBFuncBaseState {
    public:
        virtual CPBFuncBaseState* GetNextState(Triggers trigger) = 0;
    }

class CPBFuncBaseTrigger {
    public:
        virtual CPBFuncBaseState* PerformAction(CPBGenOverallUIModel* OverallUIModel) = 0;
    }

class CPBFuncState0_FuncDRAManager_StopExposure : public CPBFuncBaseTrigger {
    public:
        CPBFuncBaseState* PerformAction(CPBGenOverallUIModel* OverallUIModel);
    }

class CPBFuncState0_FuncDRAManager_HandleStateEndPosNotDefined : public CPBFuncBaseTrigger {
    public:
        CPBFuncBaseState* PerformAction(CPBGenOverallUIModel* OverallUIModel);
    }

class CPBFuncState0 : public CPBFuncBaseState {
    public:
        virtual CPBFuncBaseState* GetNextState(Triggers trigger);
        CPBFuncState0() {
            m_pTriggers[FuncDRAManager_StopExposure] = new CPBFuncState0_FuncDRAManager_StopExposure();
            m_pTriggers[FuncDRAManager_HandleStateEndPosNotDefined] = new CPBFuncState0_FuncDRAManager_HandleStateEndPosNotDefined();
            ...
        }
    private:
        map<Triggers, CPBFuncBaseTrigger*> m_pTriggers;
    }
    ...
```
LISTING D.2: A code fragment of the C++ implementation file produced by the code generator after improvement

```cpp
CPBFuncStateMachine::CPBFuncStateMachine()
    : m_pCurrentState(new CPBFuncState0())
{}

void CPBFuncStateMachine::StateChanged(Triggers trigger)
    if (m_pCurrentState)
        CPBFuncBaseState* pState = m_pCurrentState->GetNextState(trigger);
        m_pCurrentState = pState;
        ...
    }

CPBFuncBaseState* CPBFuncState0::GetNextState(Triggers trigger)
    return m_pTriggers[trigger]->PerformAction(OverallUIModel);

CPBFuncBaseState* CPBFuncState0_FuncDRAManager_HandleStateEndPosNotDefined::PerformAction(CPBGenOverallUIModel* OverallUIModel)
    // IUsed1::CPBAbstManagerSingleton_GetDataModelFacade
    CPBAbstDataModelFacade* pCPBAbstDataModelFacade =
        &CPBAbstManagerSingleton::Instance().GetDataModelFacade();
        ...
    CPBFuncBaseState* nextState = NULL;
    nextState = new CPBFuncState1;
    ...
    return nextState;
```
Bibliography


[38] M. Keiner, “History, definition (s) and models of sustainable development”, 2005.
[47] mCRL2, version 201409.1, Sep. 2014.