EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Automatic code modernization with Rascal

Liu, T.

Award date:
2018

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Department of Mathematics and Computer Science
Software Engineering and Technology Research Group

Automatic Code Modernization with Rascal

Master Thesis

Tianyu Liu

Graduation committee:

prof. dr. Mark van den Brand (Supervisor)
dr. Mathijs Schuts (Company supervior from Philips)
dr. Julien Schmaltz (External committee member)

Eindhoven, September 2018

Abstract

Large-scale software systems are widely used in the past decades among all the industries. These software systems require frequent maintenance to ensure proper behaviors. Philips is one of the companies in the high-tech industry, producing sophisticated Image Guided Therapy (IGT) systems for decades. These IGT systems run with large-scale software systems. Due to the age of those software systems, they contain legacy code. Therefore, it is difficult and costly to maintain those legacy code. As a result, Philips intends to modernize the legacy code in their software system.

The purpose of this study is to investigate the possibilities to transform legacy code into a later version automatically. The research subject is a part of Philips' code base, namely Philips' homemade Operating System Abstraction Layer (OSAL).

This thesis proposed a Domain-Specific Language (DSL) based automatic transformation tool to handle the transformation. It contains the important steps and details to design the transformation tool. We started with an analysis on the existing code base of Philips, then made an inventory of the existing OSAL Application Programming Interface (API)s and comprehended the use of the existing OSAL. After that, we formalized the requirements from the analysis results, used model-based design approach and finally designed a transformation engine based on structure matching of source code. The input of that tool is a DSL to let the users define transformation details. Furthermore, we also made a working prototype based on the model design and the implementation details are also performed in this thesis.

We used the prototype to apply some transformations to Philips' code base and positive results have been obtained. We can conclude that the tool described in this thesis can be used to transform a large part of the OSAL functions.

Acknowledgment

This thesis is the result of my master graduation project at Eindhoven University of Technology and Philips, within Software Engineering and Technolgoy (SET) group. I have been working with SET group for a long time, started from course Software Evolution given by dr. Alexander Serebrenik and course Generic Language Technology given by prof. dr. Mark van den Brand. These well-designed courses let me find my real interests in software engineering and eventually graduated in SET group. In this long period of time, I wish to give my appreciations to a lot of people.

First of all, I would like to give my sincere appreciation to my supervisor prof. dr. Mark van den Brand, for taking me over in the mid-term of my graduation project due to the absence of my original supervisor prof. dr. Jurgen Vinju and giving me useful high-level guidances and advices during the project. Without his kind help, I would never be able to graduate.

Secondly, I wish to give my deep thanks to the following people. dr. Alexander Serebrenik, my previous supervisor during Capita Selecta project and Seminar SET, for guiding me during the projects and recommending me this internship position. dr. Mathijs Schuts, my company supervisor during my internship at Philips, for helping me and supporting me throughout the project. prof. dr. Jurgen Vinju, my original supervisor of this graduation project, in the first few times of meeting he showed me an overall look of the project, which has been helpful for the further study. Jurgen is absent during my graduation project due to illness, I wish all the best for his recovering.

Furthermore, I am very grateful to Rodin Aarssen, a PhD student at the TU/e together with Philips, who developed the fantastic tool ClaiR, an important component in my study. Without the help of ClaiR, it would be impossible to conduct my project. I wish him all the best in his PhD study. Also, I wish to show my thanks to my colleagues at Philips, including but not limit to Bora Akar, Clemens van Kempen, Jan Stevens, Luc Schouren, Peter Blom, Roel Kolman, my manager Paul Tielemans, and many other people, for the useful help during my internship. My thanks will also go to Weslley Silva Torres, for the happy lunch time we had at Philips, and Felipe Ebert, for the cooperation we had before in SET group. And I wish to thank my committee members: prof. dr. Mark van den Brand, dr. Mathijs Schuts and dr. Julien Schmaltz, for taking time to read and assess my thesis.

Finally, the most important ones, I would give my deepest thanks to my beloved parents, who have given me endless supports during the whole period of my study as well as my whole life time, both materialistic and spiritual. I would also give tons of thanks to my best friends, including but not limit to: ir. Huiyi Zhang, ir. Nan Yang, ir. Shidong Song and Yuyang Qi, for all the happy time we have spent; and also to ir. Yuguang Zhao and Ziyuan Zhao, for the amazing photography time.

Tianyu Liu
Sunday 30th September, 2018
Eindhoven

Contents

Contents **vii**

List of Figures **ix**

List of Tables **xi**

Listings **xiii**

List of Algorithms **xv**

Acronyms **xvii**

1 Introduction **1**
1.1 Problem Description . . . 2
1.2 Thesis Outline . . . 4

2 Background and Related Work **5**
2.1 Software Modernization . . . 5
2.1.1 Early Usage . . . 5
2.1.2 State of the Art . . . 5
2.1.3 Exemplary of Modernization Techniques . . . 6
2.1.4 Code Modernization and Refactoring . . . 6
2.2 OSAL and Philips' TOS OSAL . . . 7
2.3 Rascal and its Extensions . . . 9
2.3.1 Example of Rascal Code . . . 9
2.3.2 ClaiR . . . 11
2.3.3 Abstract Syntax Tree (AST) and ClaiR AST . . . 11

3 Analysis of Philips' Operating System Abstraction Layer **13**
3.1 Analysis Steps . . . 13
3.2 Categories of the Philips' OSAL APIs . . . 14
3.3 Distribution of Operating System Abstraction Layer Application Programming Interfaces . . . 16
3.4 Distribution of Application Programming Interface (API) Usage . . . 18
3.5 Operating System Abstraction Layer (OSAL) APIs Use Cases . . . 19
3.6 Preprocessor Statements . . . 22
3.7 Conclusion of the Analysis . . . 23

4 Tool Design **25**
4.1 Design Approach . . . 25
4.2 Study of Code Use Cases . . . 26
4.2.1 Study of Atomic Cases . . . 26
4.2.2 Study of Paired Cases . . . 28

4.3 Requirement Analysis . . . 30
4.3.1 Additional Description About the Requirements . . . 31
4.4 Design Results . . . 32
4.4.1 Architecture Design . . . 32
4.4.2 Data Structure . . . 35
4.4.3 C++ Modernization Language - the DSL . . . 37
4.4.4 Work Flow . . . 40
4.4.5 Algorithms . . . 42
4.5 Limitations and Future Work . . . 48
4.5.1 Limitation of Supported Transformation Type . . . 48
4.5.2 Limitations of the Format of Transformation Rules . . . 48
4.5.3 Future Work . . . 48
4.6 Conclusion of Tool Design . . . 48

5 Application and Case Study **51**
5.1 Apply an Atomic Transformation to Philips' Code Base . . . 51
5.2 Apply a Simple Paired Transformation to Philips' Code Base . . . 53
5.3 Discussion . . . 57
5.3.1 Discussion of the Correctness . . . 57
5.3.2 Discussion of the Performance . . . 57
5.4 Conclusion of Case Study . . . 58

6 Conclusions and Suggestions **59**
6.1 Contributions of this Thesis . . . 59
6.2 Suggestions to Philips . . . 60

Bibliography **61**

Appendix **65**

A Rascal Script Used for Searching Functions **65**

B User Manual of the Transformation Tool **67**
B.1 Installation . . . 67
B.2 Write the DSL . . . 73
B.3 Apply a Transformation . . . 74

C Implementation Examples in Rascal **75**
C.1 Exemplar Implementation to Match a Single Statement . . . 75
C.2 Exemplar Implementation to Match a Pair of Statements . . . 76

List of Figures

1.1 A picture of Philips' Allura IGT system. 2
1.2 A generic system architecture of software system in the industry. 2
1.3 An abstract illustration of Philips positioning system. 3
1.4 An abstract illustration of the project goal. 4

2.1 An abstract model of components in a general computer system [26]. 7
2.2 An abstract model of multi-platform system. 8
2.3 The abstract model of Philips' multi-platform system. 9
2.4 The Rascal development environment in Eclipse. 10
2.5 An example of a standard AST. 12

4.1 An illustration of the approach work flow. 25
4.2 The three-layer model of this study. 33
4.3 An high-level view of the transformation tool. 34
4.4 Data flow model of the transformation tool. 35
4.5 The illustration of the data structure of the transformation tool. 36
4.6 The model of the DSL. 38
4.7 The workflow design of the transformation tool. 41

5.1 A file comparison for adding inclusion. 52
5.2 A file comparison for passing a pointer access. 52
5.3 The test result for transforming the sleep function. 52
5.4 File comparison of the open pattern. 54
5.5 File comparison of the close pattern. 55

B.1 Get Eclipse for Java Developers. 68
B.2 A screen of installing Eclipse CDT. 69
B.3 A screenshot of a new Rascal console of ClaiR. 70
B.4 Check ClaiR as reference project. 71
B.5 The Java build path. 72
B.6 Console example of a successful installation of transformation tool. 72
B.7 The structure of a Domain-Specific Language (DSL) script. 73
B.8 The structure of transformations. 74

List of Tables

3.1 Philips'OSAL category and description. 14
3.2 List of OSAL APIs 16
3.3 Distribution of OSAL function calls by functionality. 19
3.4 Distribution of OSAL function calls by general category. 19

4.1 The criteria to choose an example for atomic cases. 26
4.2 Detailed information about the starting point: sleep function. 28
4.3 Details of simple paired scenario example. 29

5.1 A summary of running time in different transformation. 57

Listings

2.1 A Rascal code example to declare a new data type. 10
2.2 An example code fragment to demonstrate an AST. 11
2.3 An example statement to be parsed in ClaiR. 11
2.4 An example of ClaiR AST. 12
3.1 A minimal example of atomic function use. 20
3.2 A minimal example of simple paired function use. 20
3.3 A minimal example of complex paired function use with value passing. 21
3.4 A minimal example of an independent function implementation. 21
3.5 A minimal example of a dependent function. 21
3.6 An exmaple of conditional complication. 22
4.1 A transformation example of the mutex functions. 29
4.2 The Rascal code to define the top-level data structure. 37
4.3 An example of a DSL script. 40
5.1 The DSL script to transform the Sleep function on Philips' code base. 52
5.2 The DSL script to transform the Sleep function on Philips' code base. 53
5.3 The DSL script to transform the mutex construct on Philips' code base. 54
5.4 The DSL script to transform the mutex construct on Philips' code base in several steps. 56
A.1 A Rascal code fragment for searching existing functions. 66
C.1 The implementation of matching a single statement. 76
C.2 The implementation of matching a pair of statements. 77

List of Algorithms

1 Top level transform. `transform(l)` 42
2 Parse a DSL and generate transformation. `generateTransformation(l)` 43
3 Transform a list. `transformList(listOfTransformation)` 43
4 Transformation algorithm for an atomic transformation. `transformBasedOnType(t, startLocation)` 44
5 Modify files for an atomic transformation. `modifyFileAtomicVersion(source, target, file, metadata)` 45
6 Transform a simple paired transformation. `transformBasedOnType(t, startLocation)` 46
7 Modify files for a simple paired transformation. `modifyFileSimplePairedVersion(pairSource, pairTarget, file, metadata)` 47

Acronyms

API Application Programming Interface.

AST Abstract Syntax Tree.

ClaiR C(++) language analysis in Rascal.

CML C++ Modernization Language.

Cpp C Preprocessor.

DSL Domain-Specific Language.

IDE Integrated Development Environment.

IGT Image Guided Therapy.

JDK Java Development Kit.

JVM Java Virtual Machine.

LOC Lines Of Code.

OS Operating System.

OSAL Operating System Abstraction Layer.

TIA Test Impact Analyzer.

UI User Interface.

UML Unified Modeling Language.

URL Uniform Resource Identifier.

Chapter 1

Introduction

In high-tech industries, software plays a very important role in their products, the products that are constructed using cutting-edge technologies [24]. In order to catch the cutting-edge technologies, the software must be updated from time to time. Thus, millions of Lines Of Code (LOC) were created and kept updating in the past decades in order to operate those high-tech products properly. As a result, many LOC in a high-tech system have become **legacy code**. Legacy code means the code came from someone else which is not in the current team anymore; the code belongs to someone else; the original author of the code is not reachable or the code came from other companies [8].

In large-scale software development, it is important to create the software in a proper manner and to ensure it fully functioned [21], [1]. However, the software system has a tendency to defect due to the increasing number of LOC [12], resulting in accumulated maintenance difficulty. The difficulties originate from the low readability of the code, the incompatibility of the latest Operating System (OS), the poor understandability and other unexpected reasons.

Philips is a high-tech company, producing high-end medical equipment in the past decades. Image Guided Therapy (IGT) system is one of the advanced medical systems designed by Philips, using images provided by interventional radiology technologies to treat diseases. This interventional radiology is different from the diagnostic radiology, containing both diagnosis and treatment of diseases with minimal invasion. There are two product lines in Philips' IGT system, namely Allura and Azurion. Allura is the current maintained product line and Azurion the next generation product line. To operate this system properly, both product lines require a large-scale (more than 1 million LOC) software. In such a software system, legacy code is inevitable. The positioning team[1] of Philips planned to investigate the possibilities to modernize their positioning software in the Allura product line. **Figure 1.1** is a picture of the Allura IGT system.

Philips' IGT system has a three-layer architecture as shown in **Figure 1.2**. The intermediate layer, also known as the Operating System Abstraction Layer (OSAL), is employed to provide cross-platform support and allow same top-level application to run on different platforms. The Operating System Abstraction Layer (OSAL) operates very similarly to the Java Virtual Machine (JVM) of Java, acting as a bridge between the upper level applications and the OS or hardware. Hence, the top-level applications can keep intact to support different versions of OS or hardware platforms. With OSAL, the top-level applications can always call the same Application Programming Interface (API) for the corresponding behavior regardless the OS or hardware [19], [23], [33]. This design results in less maintenance regarding time and cost. All the operations to control the lower level OS or hardware are implemented in the intermediate layer. The concept of this intermediate layer also applies to many embedded and mobile developing tools in the industry like Rhodes, PhoneGap DragonRad and MoSync [20].

[1] The positioning team creates and maintains the software that controls all the movable parts on the IGT system.

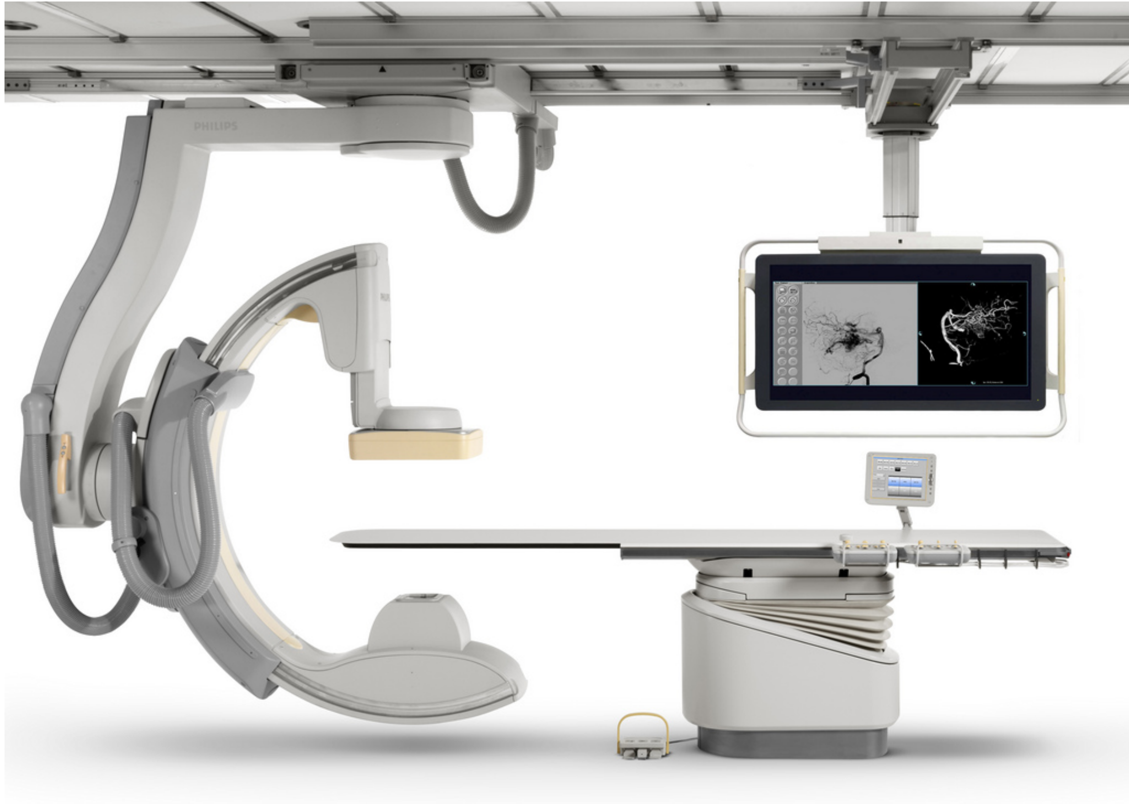

Figure 1.1: A picture of Philips' Allura IGT system.

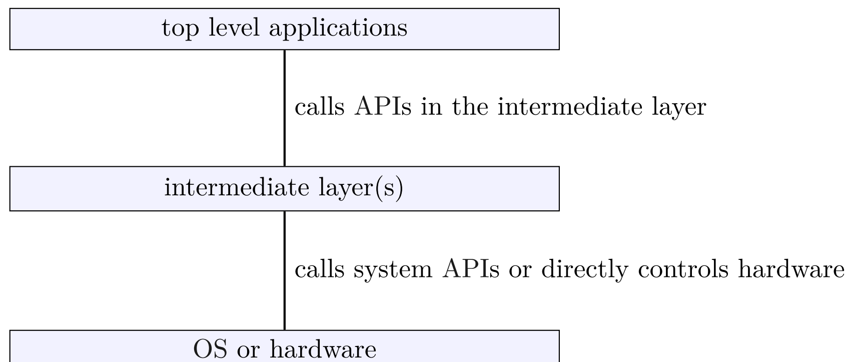

Figure 1.2: A generic system architecture of software system in the industry.

The goal of this project is to design and to build an automated transformation tool to modernize the OSAL in Philips' code base. For this aim, several research questions need to identified.

1.1 Problem Description

In software engineering domain, legacy code has been accumulated inevitably along time [8]. Philips' software system is no exception. The existence of legacy code makes the whole-life-cycle maintenance even more time-consuming and costly [1].

Introduction to the case

The main focus of this project is Philips' intermediate layer, which was originally created decades ago and acts as a OSAL. In this OSAL, there are different implementations to fulfill the same functionality on different OSs. Philips' OSAL is an important part of positioning software, as shown in **Figure 1.3**. Philips' OSAL is in a library named public library with different components. The applications also use the functions provided by other components in this public library. The OSAL has several variants of implementation to support both Microsoft Windows and Wind River VxWorks[2].

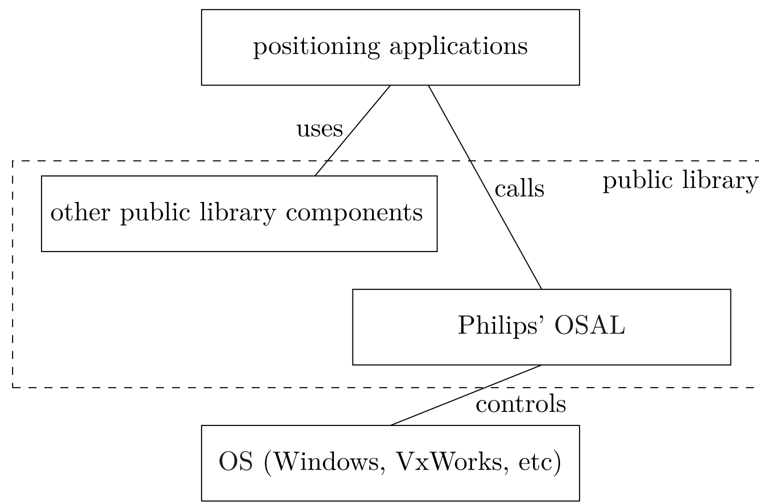

Figure 1.3: An abstract illustration of Philips positioning system.

The top-level positioning application contains the positioning logic. To apply this logic to the real machine, these positioning applications use both the OSAL and other components in the public library. The OSAL handles the operations related to OS, while the other public library components handle the non-OS-related logic. Due to the upgrade of Windows and VxWorks, it is necessary to maintain Philips' OSAL, which can be very time-consuming. Furthermore, the larger number of LOC in Philips' OSAL might lead to higher possibilities of defect [12], resulting in a higher maintenance cost [27].

Given the aforementioned situation, the positioning team of Philips IGT decided to find a solution to reduce the maintenance cost in the future. According to previous investigations by Philips, there are two existing standard libraries providing an OSAL, namely the C++11 standard library and POSIX standard library. Both of the libraries are supported by Microsoft Windows 10 and VxWorks7, which are the two OSs Philips is starting to use. As a result, Philips wants to explore whether it is possible to deprecate the home-made existing OSAL and use the native C++11 OSAL or POSIX OSAL instead. The goal of this project is shown in **Figure 1.4**. To replace Philips' home-made OSAL with standard C++11/POSIX OSAL, the positioning applications must be modernized.

With the help of the native C++11 OSAL or POSIX OSAL, Philips only has to maintain the applications, while the maintenance of OSAL is no longer needed. Our challenge is: Is it possible to automatically modernize the source code so that it uses native C++11 OSAL or POSIX OSAL instead of home-made OSAL? If so, what should be done? In order to reach this goal, we raised the following research questions:

[2] A Real-time OS used in Philips' IGT systems.

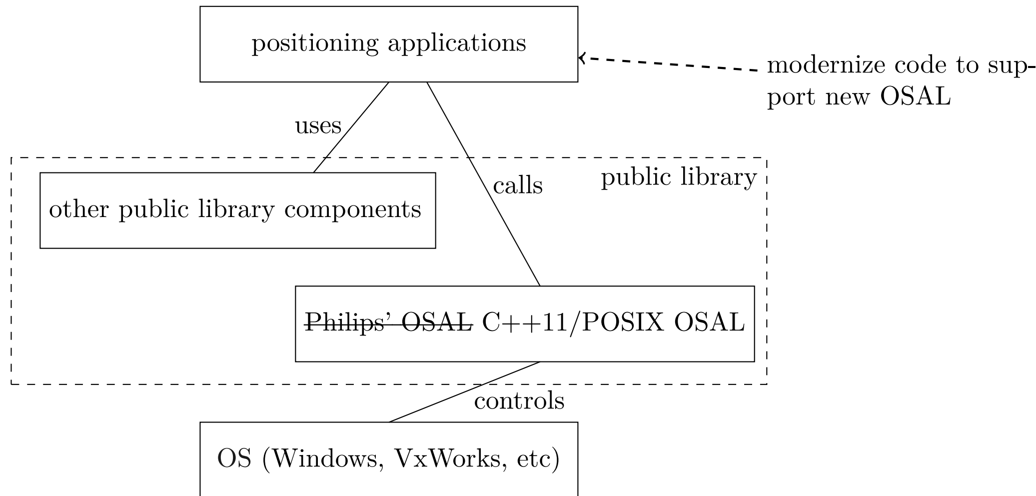

Figure 1.4: An abstract illustration of the project goal.

RQ1 What Application Programming Interface (API)s are provided in Philips' home-made OSAL?

RQ2 What are the functionality of Philips' OSAL APIs?

RQ3 What is the quantity and the distribution of Philips' OSAL APIs?

RQ4 How are Philips' OSAL APIs being used?

RQ5 What are the key aspects to make a transformation of OSAL APIs?

RQ6 What can be done to make the transformation as automated as possible?

1.2 Thesis Outline

The main content of each chapter is shown below:

- **Chapter 2**, **Background and Related Work.**
  This chapter describes the background, related work of this study, the existing OSAL in Philips' code base and an explanation of Rascal.
- **Chapter 3**, **Analysis of Function Calls.**
  This chapter contains an extensive analysis on the existing OSAL in Philips code base. The analysis categorizes all interfaces provided by the OSAL, counts the number of occurrences, summarizes the way to use the OSAL and finally investigates the use of preprocessor macros in the OSAL.
- **Chapter 4**, **Tool Design.**
  This chapter describes the detailed design and implementation of the transformation tool developed in this project.
- **Chapter 5**, **Application and Case Study.**
  In this chapter, we conduct several case studies where we apply the transformation tool to Philips' code base.
- **Chapter 6**, **Conclusions.**

Chapter 2

Background and Related Work

This chapter introduces the background and related work of this study, including the existing examples of software modernization and a description of the tooling used in this study.

2.1 Software Modernization

Software system modernization is a part of domain **software evolution**. In the past decades, many researchers and developers have elaborated on investigating different approaches to evolving the software systems and making them operate more efficient.

2.1.1 Early Usage

Due to the rapid change of the software systems in the past decades, new architecture, new hardware or new version of OS have been introduced continuously. Thus, it has become difficult to port the source code to a new architecture/hardware/OS version. The first use of term software modernization can be tracked back to as early as 1998 by NASA [10]. In NASA's early approach, a six-step approach for code modernization was introduced: clean up, serial optimization(modernization), parallel optimization, performance monitoring, making automated tooling and making machine specific optimization [10].

2.1.2 State of the Art

Apart from the early approach by NASA, other researches also introduced several approaches to evolving legacy software systems. In 1997, Weiderman et al. [31], [32] introduced a taxonomy of operational activities can be applied for legacy software systems evolution. Weiderman et al. divided the evolution activities into 5 steps: assessment, maintenance, transformation, replacement and the combination of the aforementioned four activities [32]. With the help of these five major steps, a legacy system at any stage can be evolved in a proper manner. Thus, if a legacy system is still running efficiently, assessment and maintenance can be applied; if a system is old and inefficient, then replacement can be applied to evolve the system.

Further research in 2000 by Comella-Dera et al. [5] redivided the five steps introduced by Weiderman et al. into three steps, the S1, S2 and S3 listed below is based on different stages of the legacy code:

S1 Maintenance
Maintenance is an iterative and incremental activity to apply small changes in the software systems. An example of maintenance is the rewrite implementation of one specific function. The activity will not involve the major structure of the system.

S2 Modernization
Modernization is renamed from the term **transformation** described by Weiderman et al. [32]. If maintenance is not sufficient for improving the performance and stability of a software system, then modernization can be a more efficient option. During modernization, a larger portion of the software system will be involved. Modernization often contains reconstruction, function rewriting and other operations involving multiple files [5].

S3 Replacement
Replacement is an extensive as well as intensive step in legacy software system evolution. When a legacy system is extremely difficult to be maintained/modernized, or the maintenance cost/modernization cost is too high, replacement is then a better option to evolve the system. Replacement employs a new element to compensate the missing functionality.

2.1.3 Exemplary of Modernization Techniques

In software modernization domain, there are different techniques to modernize a software system, and to fulfill different requirements, namely T1, T2 and T3 listed below:

T1 User Interface (UI) modernization
The UI of a software system requires frequent maintenances, and sometimes requires a modernization. In the past decades, the most important modernization has been transferring a legacy text-based screen to a graphical interface screen. [2].

T2 Database modernization
Database modernization is also widely applied, as the reflection of a database in real world may change from time to time [5], [17]. Three major sub-techniques can be used for database modernization, namely wrapping, statement rewriting and logic rewriting. Wrapping means to encapsulate the data into an interface for new data representation. Statement rewriting is to re-write the statement to access the database. Logic rewriting is to re-define the logic inside the database [17].

T3 Functional modernization
Functional modernization is also known as logic modernization [5]. This can be a transformation from legacy functionality (logic) into a more modern one according to the business requirements of a system. It can also be modernizing the implementation from a legacy version to a later version.

2.1.4 Code Modernization and Refactoring

When considering code modernization, **refactoring** cannot be neglected as it intersects with code modernization significantly. Refactoring is defined as: changing the software system (incl. the structure, the code construct or the calling sequence etc.) without modifying the behavior of it [9]. Considering the software modernization techniques described in **Section 2.1.3**, part of UI modernization and functional modernization can also be refactoring.

Refactoring tools are often integrated in modern Integrated Development Environment (IDE)s, such as *Eclipse*, *Microsoft Visual Studio*, *IntelliJ IDEA*. Commonly used refactoring tools include *rename*, *extract local variable or methods*, *inline methods* and *introduce parameter*. [18], [29], [16]. These refactoring tools are widely used in industry and open source communities [29], [16]. Apart from the refactoring on code, refactoring a UML model is often applied in the industry to provide an overview on how to modernize the software system [28].

In this project, none of the aforementioned techniques is applicable to address our problem described in **Section 1.1** since these techniques are used for single files rather than a whole system. Therefore, a new tool for transforming the legacy code in Philips' system has to be designed.

2.2 OSAL and Philips' TOS OSAL

In a modern computer system, components are organized in different layers including hardware, operating system, system applications, user applications and other components [26] as shown in **Figure 2.1**.

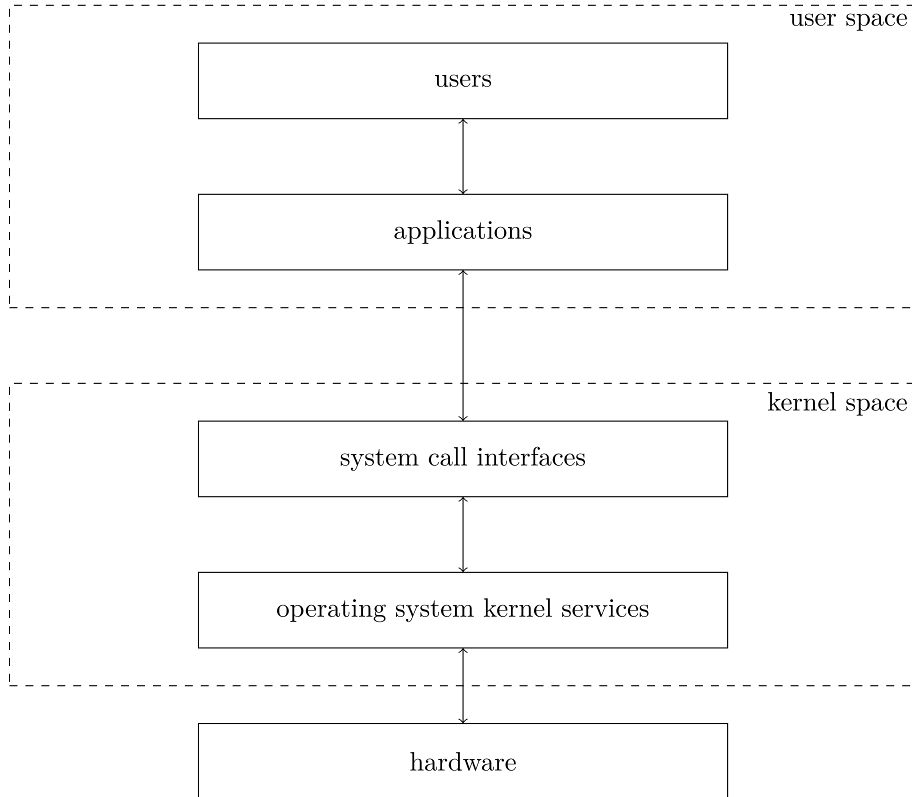

Figure 2.1: An abstract model of components in a general computer system [26].

In such a general model, there is a barrier between **user space** and **kernel space**. **user space** means the domain that gives access to users and applications, while **kernel spaces** means the domain that only gives access to the OS kernel. The applications in user space cannot access the OS kernel services in kernel space, and vice versa. One special component of the kernel space is the **System Call Interfaces**, which exposes a batch of interfaces to the user space so that users and applications can use the OS resources to control the OS and hardware indirectly. And the **operating system kernel services** often include memory manager, process scheduler (also known as task manager, file system and other necessary services. In this study we will mainly focus on the interfaces.

Multi-platform system with an Operating System Abstraction Layer (OSAL)

In practical cases, a large-scale computer system often contains several different types of hardware, uses different versions of OS to control them. Upon these OS and hardware types, the applications keep the same. This type of system is also called a multi-platform system, depicted in **Figure 2.2**.

The extended parts in a multi-platform system are the additional variants of system call interfaces, the additional types of operation systems, the additional types of hardware and an new component named Operating System Abstraction Layer (OSAL). The OSAL in a multi-platform

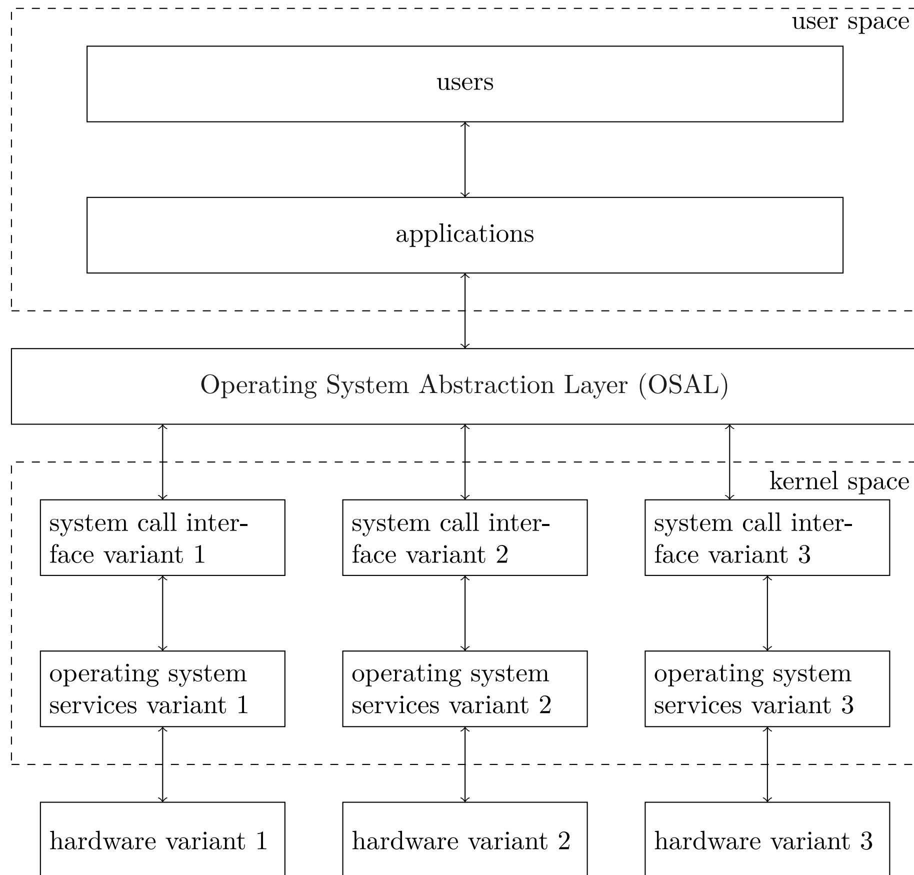

Figure 2.2: An abstract model of multi-platform system.

system replaces the task of system call interfaces in a general computer system and is exposed to the user space with only one variant. This concept follows the definition of an OSAL [19], [23]. With the help of this additional layer, the user space applications only need one implementation to support multiple variants of OSes and hardwares.

Philips' multi-platform system

In Philips' case, there is also an OSAL to bridge the gap between user space and kernel space. This OSAL was developed by Philips, named TOS. This TOS layer needs to be replaced by a modern C++11/POSIX OSAL due to the difficulties to maintain it. In the following text of this thesis, the term **OSAL** specifically means **Philips' TOS OSAL**

This model of Philips' OSAL, illustrated in **Figure 2.3**, is fairly similar to the aforementioned multi-platform model shown in **Figure 2.2**. In Philips' system, the users of positioning applications are the developers of these applications, and the client applications which use it. With the help of this OSAL, the developers can easily develop and test the applications on a Microsoft Windows development PC and then build them to the target systems running on VxWorks. All these operations can be done with only one variant of application implementation.

In the two different OS versions, each contains its own implementation to fit the corresponding hardware. The services are supported by Philips' OSAL including the following functionalities: **memory manager**, **task manager**, **network manager**, **clock (for real-time environment)** and **data-type manager**. Detailed description will be explained in **Chapter 3**.

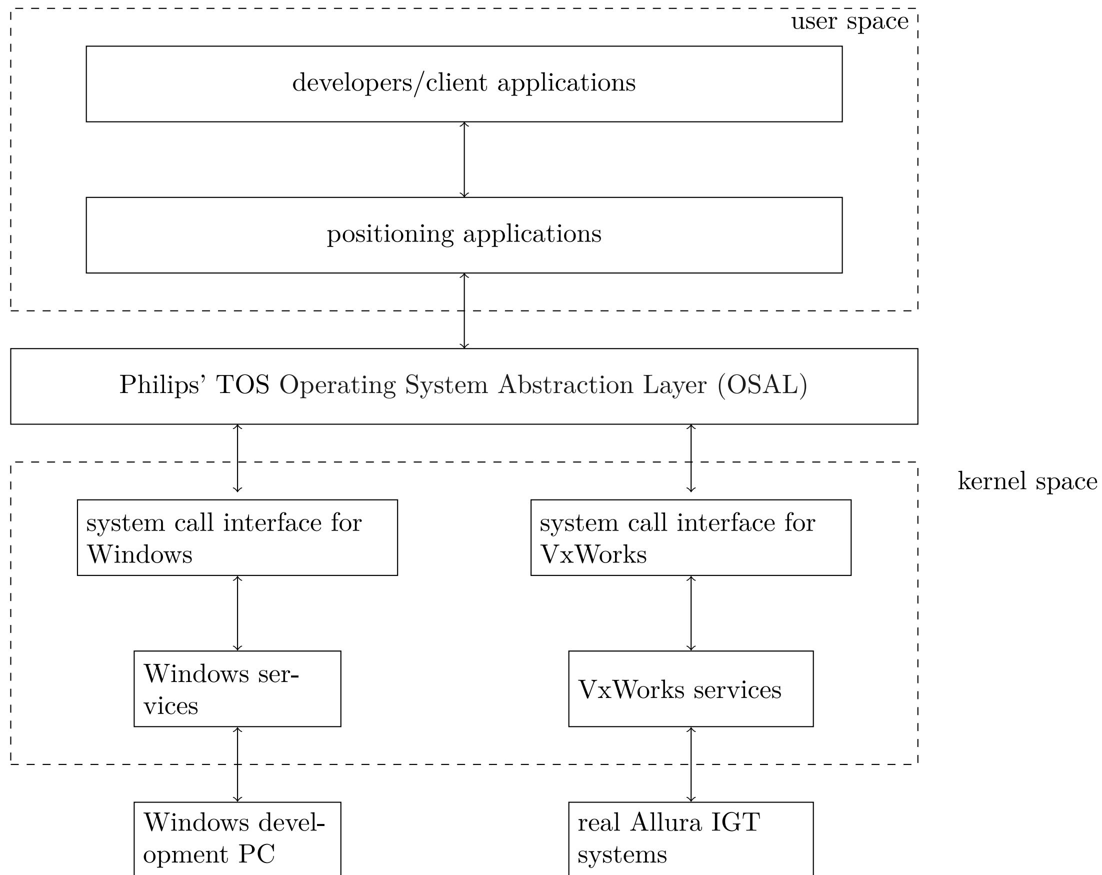

Figure 2.3: The abstract model of Philips' multi-platform system.

2.3 Rascal and its Extensions

Rascal[1] is a Domain-Specific Language (DSL) running on Java that provides an insight of source code from a high abstraction level for analysis and manipulation [11]. Rascal was first introduced in 2009 by Centrum Wiskunde & Informatica (CWI) Amsterdam[2], as a successor of syntax and semantic analysis tool named *ASF+SDF* meta-environment [30]. Currently, Rascal supports the analysis of most popular programming languages including Java, C/C++, PhP. From the users' perspective, the syntax of Rascal is very close to that of Java. In **Section 2.3.1**, some examples of Rascal code will be introduced.

Rascal is considered as a research-oriented language employed in University of Amsterdam and Eindhoven University of Technology to analyze, visualize and transform source code of legacy systems, and to construct DSLs. Apart from the education activities, Jezequel et al. [4] introduced a concern-oriented language development approach uses some concept from Rascal. Given these examples in research domains, Rascal is considered as a potential tool to address our research problems.

2.3.1 Example of Rascal Code

Rascal is a plug-in of Eclipse, so the development of Rascal can simply be conducted in Eclipse. **Figure 2.4** presents a typical layout to write Rascal code in Eclipse. The left part of the figure is

[1] `https://www.rascal-mpl.org/`
[2] `https://www.cwi.nl/research/groups/software-analysis-and-transformation`

the project navigator, lists all the files in opened projects. The upper right part is the text editing area and the lower right part is the Rascal console. Rascal script can not only be run from a file, but also directly in the console.

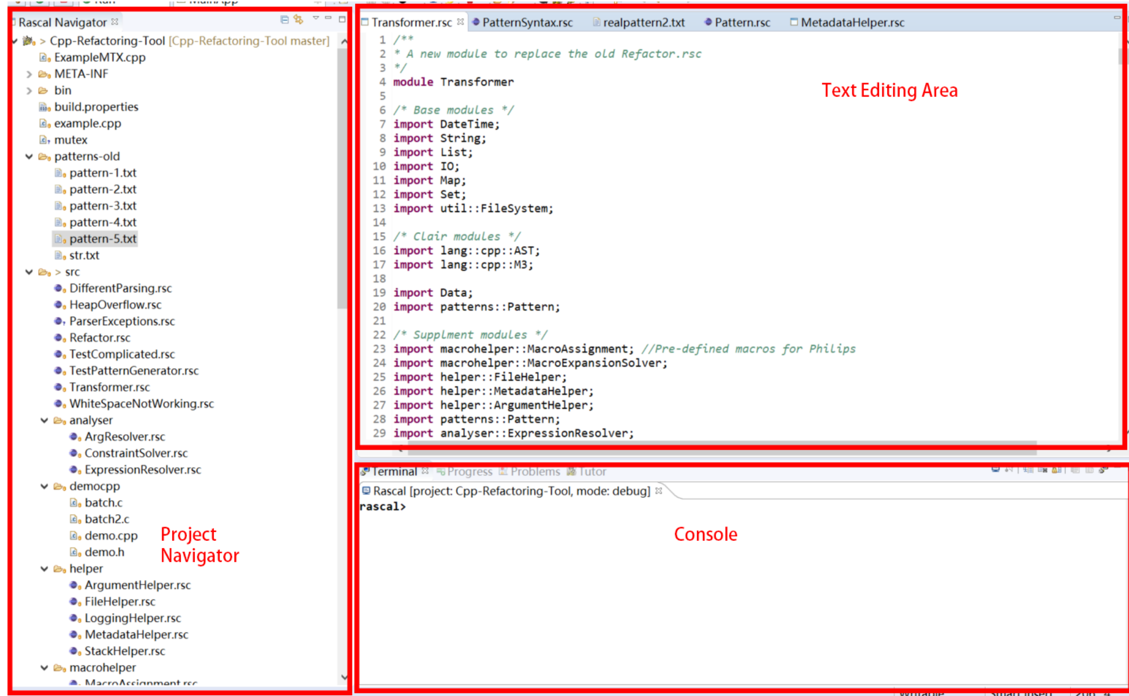

Figure 2.4: The Rascal development environment in Eclipse.

Though the syntax of Rascal is similar to that of Java, these two languages differ significantly in the declaration of data structures. In this study, the declaration of data structures is a very important aspect in the implementation phase.

Listing 2.1: A Rascal code example to declare a new data type.

```
1  data NewDataType(str optionalName = "")
2      = oneNumber(int num)
3      | twoNumbers(int numA, int numB)
4      | threeNumbers(int numA, int numB, int numC);
5
6  //Instantiate the data structure
7  NewDataType ndt = threeNumbers(1, 2, 3, optionalName = "new");
```

Listing 2.1 is a declaration of a data structure called `NewDataType`. The new data structure `NewDataType` contains one **optional attribute** named `optionalName` and the default value is an empty string. The optional attribute does not need to be present when declaring a data structure. This data structure has three variants, one named **`oneNumber`**, with one integer attribute; the second one named **`twoNumbers`**, with two integer attributes; and the third one named **`threeNumbers`**, with three integer attributes. These three variants can be considered as three inheritances of the data structure. **Line 7** declares a new **variable** `ndt` with the newly-declared data structure as its type. The variable takes the **`threeNumbers`** variant, sets **`numA`**, **`numB`** and **`numC`**

to 1, 2 and 3 respectively. This variable declaration also sets the optional variable `optionalName` to *new*.

2.3.2 ClaiR

C(++) language analysis in Rascal (ClaiR)[3] is a Rascal extension as well as an Eclipse plug-in, developed by Rodin Aarssen from CWI Amsterdam[4]. ClaiR can generate an Abstract Syntax Tree (AST) per file of a C/C++ source code file, named ClaiR AST. ClaiR AST is understandable by humans and is defined as Rascal's data structure. Hence, it is Rascal compatible. The generated AST comes from the parser of Eclipse CDT[5], the C/C++ development tooling developed for Eclipse. The core parsing steps in ClaiR are processed by the internal parser of Eclipse CDT, including parsing files and getting the inclusions (headers). The output of these core steps is a so-called CDT AST. However, the CDT AST is not that human-readable and cannot be directly analyzed in Rascal. Thus, the contribution of ClaiR is translating the CDT AST into a Rascal-compatible and human-readable version so that further analysis can be done by this tool.

2.3.3 Abstract Syntax Tree (AST) and ClaiR AST

In the general computer science domain, an AST is a tree representation of any source code. The tree representation is an abstract syntactic structure, so that it is language independent in most cases. The syntactic structure only describes the logic of the source code, ignoring the language-specific details. **Figure 2.5** indicates an standard AST of the statement in **Listing 2.2**.

Listing 2.2: An example code fragment to demonstrate an AST.

```
1 void foo(int a, int b) {
2     if (a > b) {
3         a = a + b;
4     } else {
5         a = a - b;
6     }
7 }
```

ClaiR AST

ClaiR AST is different from the standard AST in **Figure 2.5**, the ClaiR AST stores the tree representation in a textual way, using brackets to distinguish different tree levels. **Listing 2.4** shows a ClaiR AST. The AST example is a representation of a declaration statement in **Listing 2.3**.

Listing 2.3: An example statement to be parsed in ClaiR.

```
1     \texttt{int var = 1;}
```

Illustrated in **Listing 2.4**, the ClaiR AST is a `translationUnit` that contains all the statements of this file. The statements are stored in this translationUnit as a list. All other information is stored in an indented view to represent the tree structure. Furthermore, the ClaiR AST also stores a very important field for each of the nodes in the tree, named `src`, which is the exact position of

[3] `https://github.com/cwi-swat/clair`
[4] `https://www.cwi.nl/people/rodin-aarssen`
[5] `https://www.eclipse.org/cdt/`

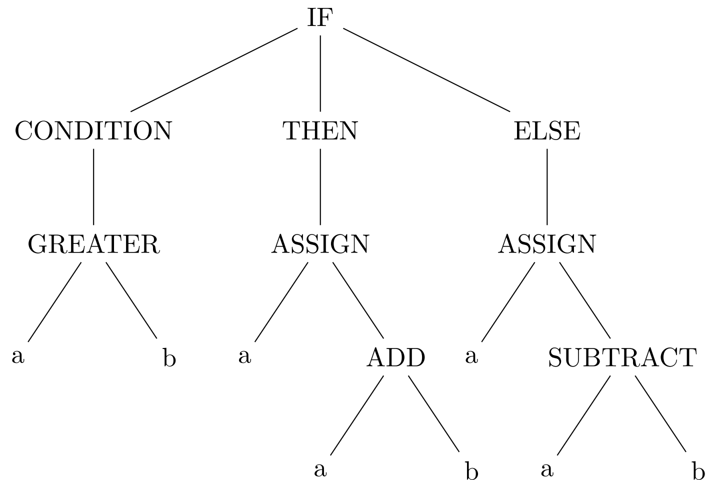

Figure 2.5: An example of a standard AST.

Listing 2.4: An example of ClaiR AST.

```
1  translationUnit(
2    [simpleDeclaration(
3        [],
4        declSpecifier(
5          [],
6          [],
7          integer(src=|tmp:///t.cpp|(0,3)),
8          src=|tmp:///t.cpp|(0,3)),
9        [declarator(
10           [],
11           [],
12           name(
13             "var",
14             src=|tmp:///t.cpp|(4,3)),
15           equalsInitializer(
16             integerConstant(
17               "0",
18               src=|tmp:///t.cpp|(10,1),
19               typ=basicType(
20                 [],
21                 int())),
22             src=|tmp:///t.cpp|(8,3)),
23           src=|tmp:///t.cpp|(4,7),
24           decl=|cpp+variable:///var|)],
25       src=|tmp:///t.cpp|(0,12))],
26   src=|tmp:///t.cpp|(0,12))
```

the corresponding node. The position is stored with type `loc`, which is a Rascal basic data type and can be used as a hyper link to point to the exact position in the file. The AST is considered as the **structure of a program**.

Chapter 3

Analysis of Philips' Operating System Abstraction Layer

This chapter contains the detailed results of Operating System Abstraction Layer (OSAL) analysis. These analyses include a categorization of all OSAL function calls, the distribution of each category, the use patterns of the function calls and the preprocessor macros in the OSAL.

3.1 Analysis Steps

To transform Philips' OSAL correctly, we list the OSAL APIs, then categorize them based on function name prefix. After that, we calculate the distribution of the OSAL APIs. At last, we discover the use cases of the APIs.

- **Examination of OSAL header files**
  We check original header files of the Philips' OSAL to retrieve all OSAL APIs and macros. OSAL APIs will be used for categorization and macros will be used to help override compiler configurations.

- **Categorization of OSAL APIs**
  We categorize the OSAL functions based on function name prefixes. Each prefix represents one functionality. For instance, if an OSAL function is used for *basic task management*, it will have prefix `TSK_`. Hence, a function which is used for task management in Philips' OSAL will be named as `TOS_TSK_foo`, where `foo` can be a specific name according to its exact behavior. Furthermore, we check the definitions (implementations) of each functionality to categorize it from a general OS point of view. For instance, if there are two aforementioned categories, one has functionality of a general task management and the other has functionality of a conditional task management, both of the categories are considered to have the **task management** functionality.

- **Calculation of the distribution of the OSAL functions**
  From the list of function calls, we calculate the distribution of function calls per category automatically.

- **Investigation of the OSAL functions**
  We examine the occurrences and refer back to the original source code file then check the usage of each OSAL function. The usage includes the calling sequences, patterns and other potential existing properties.

3.2 Categories of the Philips' OSAL APIs

Philips' OSAL APIs can be categorized into 19 functionalities, listed in **Table 3.1**. The table contains 4 columns: **Function prefix**, the prefixes are retrieved from OSAL header files; **Functionality**, the functionality of the corresponding prefix; **Category**, the category from OS perspective and **Description**, a brief description per category. The table is sorted by first column with the original sequence in the header file.

Table 3.1: Philips'OSAL category and description.

Function prefix	Functionality	Category	Description
`TOS_p_INT`	Interrupt	Task/event management	Interrupt is a signal to override current event/task. The event/task with interrupt property can be processed immediately.
`TOS_p_JOB`	OS job	Task/event management	Job object in OSAL is similar to task. In real situation these functions were only declared but never implemented or used.
`TOS_p_MBX`	OS mailbox	Task/event management	Mailbox is used as a communication media between tasks. In a multi-task scenario, messages can be sent by using the mailbox feature among all tasks.
`TOS_p_OBJ`	Philips's OSAL (TOS) object	Data type management	This type of functions are used to manage all TOS specific objects.
`TOS_p_REG`	Region-based memory management	Memory management	The region-based memory management provides a more flexible scope of a memory segment. The allocated memory segment can live longer than standard allocated memory.
`TOS_p_MTX`	Task mutex	Task/event management	Mutex means mutual exclusive objects. The mutex functions can set mutual exclusive flag to an object so that the object is locked and can only be modified by one thread.

`TOS_p_RTS`	Philips' OSAL (TOS) object type converter	Data type management	The TOS object converter converts other objects (incl. fundamental types and user-defined types) into TOS compatible objects.
`TOS_p_SEG`	Memory segment management	Memory manager	The memory segment management interface can have a directly control to a specific memory segment from an OS level.
`TOS_p_SEM`	Semaphore (non-counting)	Task/event management	Semaphore is used similar to mutex, between multiple tasks to store the sync informations. Non-counting semaphore represents only two states, also known as binary semaphore.
`TOS_p_CSEM`	Semaphore (counting)	Task/event management	Counting smaphore is identical to non-counting semaphore, only except for the representations. Counting semaphore represents multiple states.
`TOS_p_EVT`	OS event	Task/event management	The event functions handles the OS events.
`TOS_p_SCK`	Network layer	Network	This type of functions implement the TCP/IP socket from OSAL level.
`TOS_p_STK`	Stack implementation	Memory management	Provides the stack from an OS level, specifically used as memory stack.
`TOS_p_TSK`	Basic task management	Task/event management	Manages the basic task operations.
`TOS_p_MAIN`	OSAL self manager	Administrative	Provides a control panel of the OSAL layer.
`TOS_p_RTC`	Real time clock	Clock	Provides the essential APIs to control the real time clock in VxWorks, an real time OS
`TOS_p_LOG`	OSAL log system	Administrative	This interface provides the log system for this Philips' OSAL.
`TOS_p_STR`	Start job of Philips' OSAL	Administrative	The start job is specifically designed for VxWorks variant of the OSAL, helping start the OS from a cold state. However, this interface was never used.

`TOS_p_STP`	VxWorks booting	Administrative	The booting category was specifically designed for booting VxWorks as there are some additional requirements and settings for booting VxWorks compare to booting Windows.

Furthermore, we categorized these OSAL functions according to their behaviors from an OS point of view. These functions are categorized into the following:

C1 Task/event management
Handles everything that is related to task or event. Task or event in this study is also called OS jobs by Philips.

C2 Data type management
Manages the custom-defined data types in Philips' system. The operations are used to transform a non-Philips-OSAL data type to a Philips' OSAL compatible data type.

C3 Memory management
Handles everything that is memory related including creating, deleting and region based management. Here the term *memory* states the main memory of a computer system.

C4 Network
Provides an interface for the use of network layer in the OSAL.

C5 Administrative
The administrative functions for Philips' OSAL itself. It provides a control panel so that some settings can be applied to the OSAL. Also, it provides the booting procedure specifically designed for VxWorks and the build-in log system.

C6 Clock
Provides a real-time clock so that the applications that have a real-time requirement can work properly.

3.3 Distribution of Operating System Abstraction Layer Application Programming Interfaces

According to OSAL header, there are 108 single APIs provided by the OSAL, as listed in **Table 3.2**. The first column **Functionality** is identical to the same column in **Table 3.1**, the second column **#Interfaces** shows the number of APIs per prefix (functionality) and the last column **Description** explains how those APIs work.

Table 3.2: List of OSAL APIs

Functionality	#APIs	Description
OSAL log system	1	This category contains only one function to initialize the log used in servers.
Start job of Philips' OSAL	1	This category contains one function to start OSAL.

OSAL object type converter	2	The **OSAL object type converter** interfaces contain functions to either convert a C/C++ pointer to a OSAL token or convert a OSAL token to a C/C++ pointer.
OSAL self manager	2	The **OSAL self manager** provides an initialization function and a menu.
OS object	3	The **OS object** interfaces provide functions to lookup an object and catalog/uncataglog an object. The interfaces were only declared but never implemented or used.
Stack	3	The **stack** implementation in Philips' OSAL only contains interfaces to check the status of a stack. The checking functions include checking whether a stack is free, how many bits are left in a stack, the pointer to the stack and switching on/off free stack checking.
OS job	4	The **job** interfaces contain functions to create/delete a job, and get a current/root job. The interfaces were only declared in header, but were never implemented or used in real code.
Semaphore (non-counting)	4	**Non-counting semaphore** provides interfaces to create/delete a semaphore and send/receive message related to the semaphore.
Semaphore (counting)	4	**Counting semaphore** is similar to the non-counting semaphore, only the semaphore object is a multi-state semaphore.
OS event	4	The **OS event** interfaces contain functions to create/delete an OS event, set the flag of an event check whether an event is set.
Region-based memory management	4	The **region-base memory management** interfaces contain functions to create/delete a region and enter/exit a region.
VxWorks booting	7	This category only applys to VxWorks. It provides interfaces to boot VxWorks as well as check whether the OS is available and operable.
OS mailbox	8	The **mailbox** interfaces contain mailbox create, delete, send/receive, queue length query and name query functions.
Real time clock	8	**Real time clock** provides an entry to manage the real-time behavior from upper application. The interfaces contain functions to create/delete a real-time closk, get/set timestamp, get/set a timer and get elapsed time.
Memory segment management	8	**Memory segment management** provides interfaces to create/allocate, delete/delocate a memory segment and to check the remaining bits left in an existing memory segment.

Interrupt	11	The **interrupt** interfaces contain functions to create, delete, enable, disable, exit, interrupt level settings and waiting function.
Network layer	13	The **network layer** provides interfaces to create/delete a network socket, connect to a socket, bind, accept data, attach data, close socket, send and receive data. For each operation related to data, it also has a version for UDP transmission.
Basic task management	21	**Basic task management** provides the common used task managing functions including sleep, create, delete, suspend, resume. Also, it provides functions to check/set task name, check/set task priority, get thread id, measure the performance and two functions to manage the flags.

3.4 Distribution of Application Programming Interface (API) Usage

The distribution of each functionality is listed in **Table 3.3**. The table is considered as an extended version of **Table 3.2**. The first column **Functionality** is identical to the same column in both **Table 3.1** and **Table 3.2**, the second column **#Occurrence** is the sum of all invocations per functionality in positioning software code base.

As shown in **Table 3.3**, the most frequently used functionality is **Basic task management**, followed by **Memory segment management** and **Region-based memory management**, the fourth most used functionality is **OS mailbox**, which is also related to task management.

Furthermore, we calculate the distribution of each category from general OS point of view, listed in **Table 3.4**. Among all the occurrences of OSAL APIs, the most used category is **Task/event management** and is followed by **Memory management**. These two categories take more than 90% of all function calls.

Table 3.3: Distribution of OSAL function calls by functionality.

Functionality	#Occurrence
OS job	0
OSAL object	0
Start job of OSAL	0
OSAL object type converter	2
OSAL log system	7
OSAL self manager	12
Stack implementation	18
OS event	31
Real time clock	64
Interrupt	70
Network layer	111
Semaphore (counting)	115
VxWorks booting	272
Semaphore (non-counting)	394
Task mutex	397
OS mailbox	1087
Region-based memory management	1252
Memory segment management	1472
Basic task management	2651

Table 3.4: Distribution of OSAL function calls by general category.

Category	#Occurrence
Data type management	2
Clock	64
Network	111
Administrative	291
Memory management	2742
Task/event management	4745

3.5 Operating System Abstraction Layer (OSAL) APIs Use Cases

We analyze the use of OSAL APIs and generalize the model of their use cases, which can be divided into three cases:

C1 Atomic use
The term **atomic use** means that the API is not related to any other APIs or functions.

An example of an atomic API usage is shown in **Listing 3.1**.

Listing 3.1: A minimal example of atomic function use.

```
1       /* Here goes expressions */
2       int varA = 1;
3       foo(varA);
4       /* Here goes other expressions */
```

C2 Simple paired use
The simple paired use means to use the APIs in pairs. Each pair contains a **open pattern** and a **close pattern**. Both **open pattern** and **close pattern** are a group of statements. The definition of a group depends on the concrete usage in the source code. In the simple paired cases, the data in open/close patterns are not read/written by statements in between; the data in statements between the open/close patterns are not modified by both patterns. **Listing 3.2** shows a minimal example of simple paired scenario.

Listing 3.2: A minimal example of simple paired function use.

```
1       /* Here goes some expressions */
2       { //Here goes into a scope
3       /* Here goes some expressions */
4           int varA = 1;
5           if (varA == 1) {
6               varA = 2;
7           }
8           fooStartCall(varA);
9           /* ... */
10          /* Here goes critical section statements */
11          fooClose(varA);
12          /* Here goes expressions */
13      } //Here goes out of a scope
```

C3 Complex paired use with value passing
The complex paired use is an extension of simple paired use. The difference between the two cases is that in complex pair cases, the data in open/source pattern can be read/written by statements in between; the data in statements between the open/close patterns can be modified by either pattern. A minimal example of complex paired use with value passing is shown in **Listing 3.3**.

Furthermore, there are some OSAL API implementations use other OSAL APIs. Hence, we summarize these into two situations:

S1 Independent functions
Independent function means that **for every variant of the implementation**, no other OSAL APIs are used. **Listing 3.4** is a minimal example to show this case.

S2 Dependent functions
Dependent function means that **at least one implementation** of that API uses **at least one** other OSAL API. **Listing 3.5** is a minimal example to show the dependent function case.

Listing 3.3: A minimal example of complex paired function use with value passing.

```
1       /* Here goes some expressions */
2       { //Here goes into a scope
3           /* Here goes some expressions */
4           int varA = 1;
5           if (varA == 1) {
6               varA = 2;
7           }
8           type_t varB = fooStartCall(varA);
9           /* ... */
10          intermediateExp(varB);
11          /* Here goes critical section statements */
12          fooClose(varA);
13          /* Here goes expressions */
14      } //Here goes out of a scope
```

Listing 3.4: A minimal example of an independent function implementation.

```
1  //Declaration
2  void TOS_p_CATA_foo();
3
4  //Definition/implementation
5  void TOS_p_CATA_foo()
6  {
7      fooa();
8      foob();
9      sa;
10     /* ... */
11     sb;
12     /* Non of the above statement is from OSAL */
13 }
```

Listing 3.5: A minimal example of a dependent function.

```
1  //Declaration
2  void TOS_p_CATA_foo();
3  void TOS_p_CATB_dependent();
4
5  //Definition/implementation
6  void TOS_p_CATA_foo()
7  {
8      fooa();
9      foob();
10     TOS_p_CATB_dependent();
11     sa;
12     /* ... */
13     sb;
14     /* At least one of the above statement is from OSAL */
15 }
```

3.6 Preprocessor Statements

Preprocessor is an important feature in C and C++. The C programming language is incomplete without its C Preprocessor (Cpp) [7]. Ernst et al. [7] summarized the usage of Cpp in practical situations:

- **Definitions**
  Definitions are annotated with `#define` and `#undef`, meaning to define a macro, and to revoke the macro if possible. The use of definitions is `#define MACRONAME value` and `#undef MACRONAME`.

- **Inclusions**
  Inclusions are annotated with `#include`, usually divided into two types: system inclusion and normal inclusion. System inclusions are written as `#include <inc>` and normal inclusions are written as `#include "inc"`.

- **Conditional complications**
  Conditional complications includes different annotations: `#if, #else, #elif #endif` and `#ifdef, #ifndef`. The later are usually used together with definitions.

In practical situations, conditional complications are usually used as a *implement variable* [15]. With the help of this single variable, the compiler can choose the target platform easily. **Listing 3.6** shows a small example about the conditional complications.

Listing 3.6: An exmaple of conditional complication.

```
1  #if TARGET == PC
2      #define pc_environment 1
3      #define mac_environment 0
4      #define linux_environment 0
5  #elif TARGET == mac
6      #define pc_environment 0
7      #define mac_environment 1
8      #define linux_environment 0
9  #else
10     #define pc_environment 0
11     #define mac_environment 0
12     #define linux_environment 1
13 #endif
```

Suppose we want to compile executable for one of the three platforms: PC, Mac and Linux. In this case the compiler must choose the correct platform to generate the binary code. As a result, the conditional complications are used in this case. Assume `TARGET` is a variable to indicate the platform, and `PC`, `mac` and `linux` is set in the configuration file. So with the combination of this conditional complications and the setting, the target platform can be changed.

Usage of Preprocessor in this study

In this study, the usage of preprocessor macros has to be taken into consideration. Inclusions of header files, definitions of preprocessor variables and conditional complications are all used in Philips' code base.

The `#include` used in this study are the header of OSAL, which declares functions/macros/data types can be used by applications. No other actions are needed to be concerned on inclusions.

The definitions of OSAL define important constants. For instance, an *OK* status is 0, an *error* status is 2. The values of these definitions are not important to us, but the names are. During the investigation of the AST, we realize that a macro definition will be expanded to its value in the AST, which causes the lost of macro name in AST. We have to take this situation into account in the design phase, the details are described in **Section 4.3** and **Section 4.5**.

One of the important preprocessor statements is the conditional complication. As described in **Section 2.2**, Philips has several different platforms to deploy their software. The different platforms are running different OSs, which require different configurations to compile the executables. Therefore, there are several different conditional complications to distinguish these platforms. In Philips' code base, these conditional complications are called **compiler switching helper**. There are 6 different cases switched by the helper.

The cases are selected in a pre-defined work flow. In the preprocessing work flow of Philips' code, the **compiler switching helper**s are first `#undef`. After that, the preprocessor determines which platform should be chosen and `#define` the corresponding case. Due to the `#undef` step, setting the conditional complications manually is not possible. This mechanism can protect the compiler switching helper variables from unexpected changes, but it makes our analysis more difficult. As a result, an approach to address this problem has to be found. The details to solve this problem is described in **Section 4.4.5**.

3.7 Conclusion of the Analysis

The results of the analysis contain the category of OSAL APIs, the distribution, the usage and the preprocessor statements. We have found out that Philips' OSAL provides 108 APIs, which are divided into 19 categories by its function name prefix. They can also be divided into 6 groups of functionality from general OS perspective.

The use of Philips' OSAL APIs contains 7955 function calls. The most used category is basic task management with 2651 calls, followed by memory segment management with 1472 calls. The same distribution also applies to the functionality, with 4745 calls to task/event management followed by 2742 calls to general memory management. The use cases are summarized into three cases, atomic, simple paired and complex paired.

Chapter 4

Tool Design

This chapter describes the detailed design of the transformation tool. We apply an extended case study of the code usages. A intensive requirement analysis is conducted based on the case study of code usages. According to the requirement analysis, we design a model of the transformation tool and a DSL to describe the input of the transformation. Additionally, the implementation details are included in this chapter.

4.1 Design Approach

The tool design step is based on the results of case study, namely the inventory of OSAL interfaces described in **Chapter 3**. However, those results were not sufficient for building a tool as we also need a few examples to understand the patterns of a transformation. Therefore, the design starts with an extended case study on the three function call usages explained in **Section 3.5**. After gathering enough information of the examples, we start building the tool.

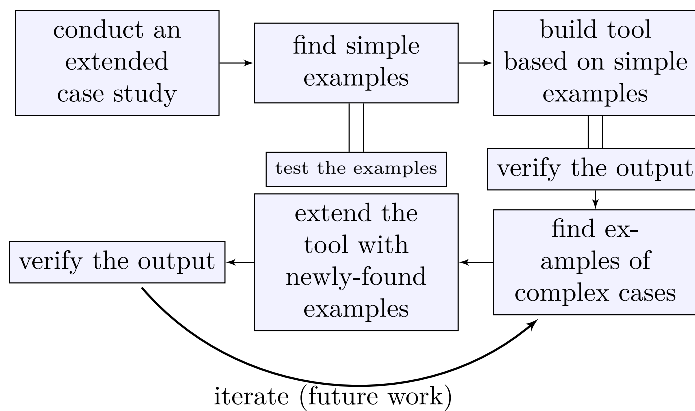

Figure 4.1: An illustration of the approach work flow.

The workflow as shown in **Figure 4.1** is divided into six blocks. The first three blocks in the top row indicate the three steps to build the structure of the tool, while the other three blocks in the bottom row show the steps to extend the tool. In the step of building the structure, we use a **criteria-based** selection strategy to find simple examples. We also test the examples that have been found, and derive the pattern from the examples. In the step of extending the tool, we use the examples found by Philips' engineers so that verifying output is unnecessary. The curve with label *iterate* indicates the future work to extend this tool.

Iterative and incremental strategy

As mentioned in the previous paragraph, the approach we use in this study fulfills the concept of iterative and incremental strategy. The iterative strategy aims at doing rework regularly to improve parts or elements of the system [3], while increment strategy is to break large scale tasks into several small tasks [3]. In the industry, it is frequent to use the combination of these two strategies [22], [14], [3].

The history of iterative and incremental strategy can be tracked to more than seven decades ago introduced by *Shewhart* [25] in quality control field. After which, this strategy appeared in software engineering field in the 1970s and grew rapidly in the 1990s [14].

In this thesis, we the concepts of iterative and incremental strategy rather than directly applying the strategy. As described before, the first step of this study is to build structure of the transformation tool, which is considered as **the first iteration**. Furthermore, the second iteration is to extend the tool based on this structure. More iterations can be applied as future work.

4.2 Study of Code Use Cases

Based on the use of function calls and the idea of iterative and incremental strategy, we aim at finding some examples of each use case to build the tool. Each example contains the usage of an OSAL API and its alternative. In this thesis, we applied the extended code study twice, which are used to find the examples of the use cases: atomic use case and simple paired use case.

For these two use cases, we use different approaches to find working examples: the first approach is **criteria-based** selection and the other one is **using previous investigation conducted by Philips**. The criteria-based selection is used to select an example for **atomic transformation** since we plan to find the example by ourselves. Due to the complexity and required knowledge, finding an example of simple paired cases is far beyond the scope of this thesis. As a result, we cannot find the examples without any help. Luckily, Philips' engineers have conducted several investigations to find the transformation examples for paired use cases. Therefore, we use the example of simple paired cases from Philips' engineers in our study.

Since the underlying structure of complex paired cases are more difficult than simple paired cases, the extended study of this case is out of the scope of this thesis. Also, based on the concept of iterative and incremental strategy, it is not recommend to directly jump into the most difficult scenario. Additionally, Philips' engineers did not find any example of this case either. In this thesis, the complex paired scenario will not be considered and it will be investigated in the future.

4.2.1 Study of Atomic Cases

The study of atomic cases is an extension of the function call analysis in **Chapter 3**. The intent of this study is to find a transformation example of this case. We define four criteria to select the example from all OSAL function call occurrences with this type, as shown in **Table 4.1**.

Table 4.1: The criteria to choose an example for atomic cases.

Criterion	Description	Rationale
Atomic	The example shall be an atomic function call according to the usage in **Section 3.5**.	Since we are choosing the example for the atomic use cases, there is no doubt to select an atomic function. This is considered as the key criterion of this selection.

Testable	There are existing test cases to test the functionality of the starting function call.	In this study, we use empirical evidences to ensure the functional correctness. As shown in **Figure 4.1**, we also investigate the possible alternative of the starting function call. Thus, a testable function call shall be chosen so that we can make sure the alternative is usable in the whole system.
Independent	The starting function call shall be used independently according to the interoperability categories in **Section 3.5**.	Since we rely on empirical evidences and testing, the first transformation should not break the testing environment. In our analysis, we have found that for some of the interface implementations that depend on other Philips' OSAL APIs, the test functions may check the variables created/modified by the related OSAL APIs. Thus, the transformation of these types of functions can break the test environment and even make the function not testable. Therefore we consider independent function calls only for starting.
Widely used	The starting function call should be called for **more than 400 times** in the code base analyzed.	The outcome of this study will finally be applied in the industry, the empirical evidence of the possibility to do the transformation must be convincing. Thus, if a function is only called 10 times, the success in transforming it is not convincing for us. We consider a function call **widely used** if its proportion of usage is **more than 5%**. According to **Table 3.3** and **Table 3.4**, for the total number of 8000 function calls, 5% is 400.

Apart from the four criteria listed in **Table 4.1**, we also consider the ease to find the alternative. Since the main goal of this study is to build the transformation model instead of finding alternatives, if it is too difficult to find an alternative of the example use, the procedure to find the alternative will be beyond the scope of this thesis. Hence, it should be easy to find an alternative for the example.

Result of the criteria-based selection

With the help of the aforementioned criteria, we manage to choose a proper example for atomic cases. We select the `sleep` function as the example of atomic cases to build the tool. Table

Table 4.2 shows detailed information of the selected sleep function.

Table 4.2: Detailed information about the starting point: sleep function.

Details about sleep function	
Item	**Description**
Original function call	`TOS_p_TSK_sleep(unsigned int)`
Target alternative (C++11)	`std::this_thread::sleep_for(chrono time)`
Criteria - atomic	Fulfilled. The use of Philips' sleep function is atomic, only a single call is needed.
Criteria - testable	Fulfilled. There exists test cases that cover the use of the function. As a result, the transformation can be verified.
Criteria - independent	Fulfilled. The implementation uses a standard delay function and no other Philips' OSAL APIs were used.
Criteria - widely used	Fulfilled. The number of occurrence of the sleep function is 1850. It is also considered the most widely used function in the code archive.

Manual transformation and verification

From the table above, it is clear that the selection of the `sleep` function fulfills all the criteria defined above. Apart from the function call transformation, there is also another data type transformation that should be conducted as the type of argument of `std::this_thread::sleep_for` is `std::chrono`. Hence we used the native type converter `std::chrono::milliseconds` to convert an `unsigned int` to a `chrono` type.

We applied a manual transformation of the **sleep** function on a file with several occurrences and then ran the tests. The testing result showed that this transformation did not break the original behavior of the sleep function. Therefore, we use this specific transformation as a starting point to build the tool.

4.2.2 Study of Paired Cases

After understanding the simple scenario and designing an automated transformation tool for that scenario, we focused on a more complicated case: the simple paired scenario as mentioned in **Section 3.5**. Since finding the alternative of the complicated cases is out of the scope of this project and this task is performed by other engineers at Philips, we directly used the result from the other engineers at Philips as an example to build the tool.

According to another investigation at Philips, the example to be taken for simple paired scenario is the functions to handle `mutex` for a task. **Table 4.3** shows all the details of the `mutex` function pair.

From the summary listed in **Table 4.3**, it is clear that the transformation of simple paired scenario is completely different from the aforementioned atomic scenario. **Listing 4.1** is an example of the two variants, the upper code fragment is Philips' version and the lower code fragment is the standard C++11 version.

Table 4.3: Details of simple paired scenario example.

Item	Description
Original functions	`TOS_p_MTX_create`, `TOS_p_MTX_enter` and `TOS_p_MTX_exit`
Functionality according to **Table 3.1**	Task mutex
OS category according to **Table 3.1**	Task/event management
# Occurrence of this functionality according to **Table 3.3**	397
Target alternative (C++11)	Declare a `std::mutex` variable and a `std::lock_guard` for that `std::mutex` variable.

Listing 4.1: A transformation example of the mutex functions.

```
1  //Here is the TOS version
2  {
3      static TOS_p_MTX_object var = 0;
4      if (var == 0)
5      {
6          var = TOS_p_MTX_create();
7      }
8      TOS_P_MTX_enter(var);
9      /* Here goes the code which is supposed in the mutex */
10     /* ... */
11     TOS_p_MTX_exit(var);
12
13     /* followed by other code */
14 }
15
16 //The following is the standard version
17 {
18     //Very important: the scope of the lock_guard is the enter
           point
19     {
20         std::mutex var;
21         std::lock_guard<std::mutex> lk(var);
22         /* Here goes the code in mutex */
23         /* ... */
24     } //No exit, but exit with end of scope bracket.
25
26     /* Here followed by other code */
27 }
```

A simple paired functions are used in pairs, start with a fixed calling sequence and end with a fixed calling sequence. Here we name the start sequence **open pattern** and the end sequence

close pattern. Apart from the patterns, another interesting observation is that for this specific case, the use of the standard `mutex` construct does not need a close pattern. The mutex lock will be ineffective after the bracket '}' at the end of the pair

4.3 Requirement Analysis

In software design, it is important to use requirements to guide and constraint the design approach [13]. The design of this transformation tool is also based on requirements that are derived from the results of previous analysis. In this section we describe the requirements derived from the problem statements in **Section 1.1** and the results of previous analysis in **Chapter 3** and **Section 4.2**. The requirement analysis intends to declare the essential requirements and discard or merge the unnecessary ones to reduce redundant and duplication in the design task. Furthermore, analyzing the requirements makes the requirements S.M.A.R.T. [6].

The requirements are listed in the following text. The description starts with the requirement itself and followed by some explanation.

REQ1 The tool must be able to transform atomic statements.
An atomic statement means the statement is used individually in the source code. The atomic transformation is considered as the first example to be used to design and build a tool. Therefore it must be supported and acts as a basis of the tool.

REQ2 The tool must be able to transform simple paired statements.
A pair of **simple paired statements** means two list of statements are in this code fragment. The code fragment starts with a particular list of statements and ends with another particular list of statements, with some other statements in between. The rationale is similar to **REQ1**, as we have an example to understand the idea of this type of transformation, it should also be supported in the tool.

REQ3 The tool must be able to insert new headers needed by the new code introduced by transformation.
Since the transformation is applied on C/C++ code and in this particular case the transformation targets are usually the latest C++11 construct, they require corresponding include files (also known as headers) to ensure static semantic correctness. Therefore it is essential to have the headers inserted together with the transformation.

REQ4 The tool must be able to pass the variables in a statement from the original code to transformed code.
Here passing variable means: if a variable in a statement is supposed to be reused after a transformation, it has to be kept unchanged during the transformation. An example of this requirement is: if the original code contains a statements such as `foo(var1);` and `var1` must be kept unchanged, then the transformed code shall also contain the same form of the variable for instance `foo_new(var1)`. Since in the code base we used for this study one type of function call can have different forms of arguments, including basic types, variables, macro expansions or variables with field accesss. If the same variable cannot be passed into the transformed code, the functionality of the code will be changed. Therefore it is necessary to pass variables if applicable.

REQ5 The tool must be able to solve expanded macros in the source code.
Macro definitions are used very often in C/C++ programming. Sometimes a variable used in source code comes from a macro definition with a meaningful name. As stated in **REQ4**, the design should let the tool be able to pass those variables and the variables may be a macro expansion. Additionally, as mentioned in **Section 3.6**, there are some OSAL constants defined as macros. However, the AST of a code file is based on the information

after preprocessing. Hence, the macro definitions are expanded into its true value in an AST. Thus, if the macro has an informative name, it will get lost in an AST. From the transformation and functionality perspective, this will not break the transformation and the functionality. However, the meaningful names will lose in the source code file and reduce the readability of the source code. Therefore there shall be a feature to retrieve the original macros of the expanded ones.

REQ6 The tool must be able to do the transformation for all conditional complication flags.
Preprocessor macro is a C/C++ exclusive feature to let the compiler process a piece of code fragment before real compiling. Conditional complication is one of the preprocessor types. Practically, conditional complication is usually used to let the compiler to select different platforms so that it can generate proper executables. According to the analysis in **Section 3.6**, there are several different platforms to be chosen by the conditional complication. Since the flags are set internally by the compiler, we need to find a solution to let the transformation tool override the settings from the compiler and visit all possible conditions.

REQ7 The tool must be able to modify the source code file directly from the file system.
The last step of code transformation is to generate transformed code and output it into original source code files instead of printing it in the console only. Hence, a proper file writer is also planned for design.

REQ8 For each transformation, there can only have one input to define the transformation details.

Since the goal of this study is to investigate whether it is possible to perform code transformation automatically, and if it is possible how to achieve it. We shall try to make the transformation as automated as possible. We concluded that more inputs lead to less automation. Therefore, we decide to reduce the number of input entry to only one and encapsulate all transformation data in that entry. With the help of this one-time input, the transformation only needs one manual invocation and the rest will be performed automatically.

REQ9 The location to start a transformation must be defined when invoking it.
The transformation will not be applied on all files on the computer, hence there should be a location or path to reduce the scope of the files. Hard-coding the start directory is not sophisticated in designing, therefore the top level directory that contains all files to be transferred will be defined in the input data of the transformation.

REQ10 The applicable file types shall be defined in the input data.
Similar to **REQ9**, another aspect to reduce the transformation scope is the file types, usually defined by file extensions. For standard C/C++ files, the source code files can be `.c` or `.cpp`. However in practical situations like Philips' case, other file extensions like `.i0c` are also used. Therefore the applicable file extensions shall also be defined manually.

REQ11 The additional header files shall be defined in the input data.
As mentioned previously the new C++ constructs may require new header files to work properly. Therefore the transformation shall have the information on which inclusion shall be inserted.

REQ12 The tool should support extensions for new transformation types.
Since we only include two out of three cases in this study, the design of the tool should support further extensions, in this study the further extension means to support the complex paired case.

4.3.1 Additional Description About the Requirements

Based on the analysis of the 12 requirements listed in **Section 4.3**, for each of the requirements listed we have made a design choice to meet the requirement. The requirements can easily be

designed and implemented in the model as the core logics (or basic logics) of the tool, different approaches of design and implementation only differ the efficiency but can function similar. The detailed design and algorithms to implement functional requirements will be described in **Section 4.4.1** and **Section 4.4.5**.

One input entry

As mentioned in **REQ8**, we aim at reducing the number of input entry to one in the design process. There are different ways to reach this goal including building a graphic interface to retrieve the input data, retrieving an external pattern file with all necessary transformation details or creating a Rascal module to store all the transformation details and then changing this module. Since we use Rascal for transformation in this study, building a graphical interface for input is the first to discard as Rascal is a console based language and does not provide a powerful interactive graphical user interface. Changing a Rascal module directly is the second discarded option since it is better to keep the original Rascal modules unchanged from software engineering perspective. Furthermore, changing Rascal module requires the user of this transformation tool to learn Rascal.

As a result, we chose to let the tool retrieve transformation data from an external pattern file and then process the transformation automatically. Retrieving information from an external file and operate the code is actually the concept of a DSL. Hence, we let the transformation tool use a DSL as input. The DSL script describes the detail of transformations and is the input used by the transformation tool. The detailed design and examples of this DSL will be described in **Section 4.4.3**.

Retrieving additional information

Except for the transformation detail, we also need additional information including **start place**, **applicable file type** and **additional inclusions** as mentioned **REQ9**, **REQ10** and **REQ11**. Since we chose the DSL approach to let the DSL script be the only input entry of all transformation information, these additional information will also be written in the DSL script. The details of retrieving this information will be described in **Section 4.4.1** and **Section 4.4.2**.

4.4 Design Results

This section describes the detailed results of the design according to the choices made based on the requirements. The design started with describing the models of this tool, followed by defining data structures and the DSL. Then we describes the work flow of the transformation engine and the algorithms used in each of the step.

4.4.1 Architecture Design

The architecture of the tool is used to describe the overall structure of each component in the tool. We will use two types of models to describe the overall structure: the high-level view of the tool and the data flow model.

High-level view of the tool

The high-level view of the tool takes the transformation tool as a whole system, illustrates the components from an external perspective. In this transformation tool, the subject to be analyzed is C/C++ code, which is at a programming language level. From programming language perspective, a three-layer model is often used to describe it. This model contains the following three layers: syntax, static semantics and dynamic semantics. In generic cases, **syntax** means the structure of a statement of a language; **static semantics** means the meaning of a syntactically correct statement of a language; while **dynamic semantic** means the actual behavior when executing a

statement. In this study, **syntax** means the structure of the code; **static semantics** are related to the type declarations of functions/variables that are used in the code, which are put in the header files; while **dynamic semantics** means the runtime behavior of the code. The model and examples are illustrated in **Figure 4.2**.

		in a transformation	correct example
Layer 2	dynamic semantics	runtime behavior	`sleep(x);` pauses a thread for `x` milliseconds.
Layer 1	static semantics	declarations of functions/variables	`int a;` `a = 1;` (a is with type int)
Layer 0	syntax	code structures	`foo(v);` `if (v == 1){...}` (function call followed by if)

Figure 4.2: The three-layer model of this study.

In the three-layer model, layer 0 and layer 1 can be analyzed **without** executing the program, while the analysis of layer 2 requires to execute the program. In a correct transformation, the modification in layer 0 and layer 1 should ensure the same behavior of layer 2. Since the transformation tool cannot execute the code, we focus on the transformation of layer 0 and layer 1, while the behavior equivalence of layer 2 shall be verified by code testing system. Thus, the high-level view of the transformation tool is illustrated in **Figure 4.3**

In the high-level view, the transformation tool is divided into four parts: original file, DSL script, transformation engine and transformed file. In both original file and transformed file there are three composites: relevant code fragments, headers and irrelevant code. In the DSL script describes headers to be inserted and the actual transformation rule. The transformation engine checks both the structure and the static semantic based on the data input from DSL script.

The transformation rule is used to handle **syntactic transformation**, which defines source and target of a transformation in terms of structures. This affects **layer 0** as described in three-layer model. This rule is input to the transformation engine together with the original file. The relevant code fragments, if exists, are matched with source of the transformation rule. Then the matched relevant code fragments are transferred to a new version that fulfills the structure of target in the transformation rule. Source and target in a transformation rule should have the same runtime behavior while being defined.

The *headers to insert* part in DSL script is used to handle **static semantic transformation**. To keep static semantics correct, newly inserted code fragments might need additional declarations, which requires new headers to be inserted. This step also reflects **REQ11** described before. During a transformation, the headers that will be inserted are input into the header checker in the transformation engine to check whether it is necessary to insert. If so, the new header will be inserted into the transformed file. This step ensures the correctness from **layer 1**. Besides the relevant code fragments and the headers, all other text in the original file is considered as **irrelevant code**. These part of text is input into the transformation engine during a transformation but remains unchanged.

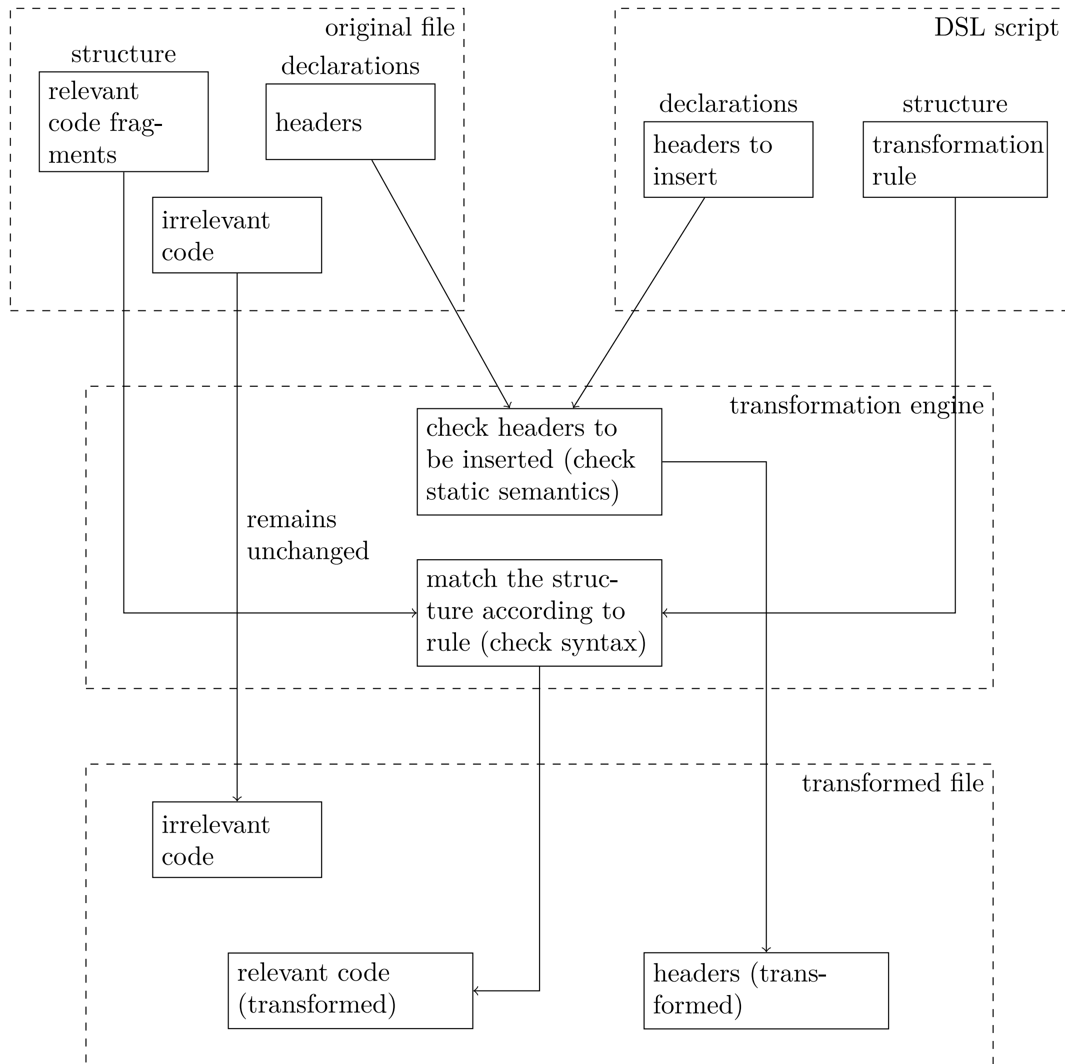

Figure 4.3: An high-level view of the transformation tool.

Furthermore, the transformed files should be executed in a testing environment to check whether the transformation breaks runtime behavior or not, which is on layer 2. Since this step is not performed in the transformation tool, it is not presented in the high-level view of **Figure 4.3**. In principle, the runtime behavior should be preserved when defining a transformation rule. Therefore, if the transformed code cannot past the test, the transformation rule might be incorrect and it should be modified.

Model of data flow

The data flow model is a diagram to indicate the data flow in the system. It describes the content of data between different components. The data flow in **Figure 4.4** illustrates that from the user side the data is the DSL script created by the users. The DSL script is input into the DSL parser and transformation generator. Transformation rules and headers to insert in **Figure 4.3** are generated by the DSL parser and transformation generator, which will be used by the transformation engine. In this design, ClaiR is used to generate the transformation rules. Then the **Transformation** in **Figure 4.4** is the output, which contains the transformation rules and the new headers.

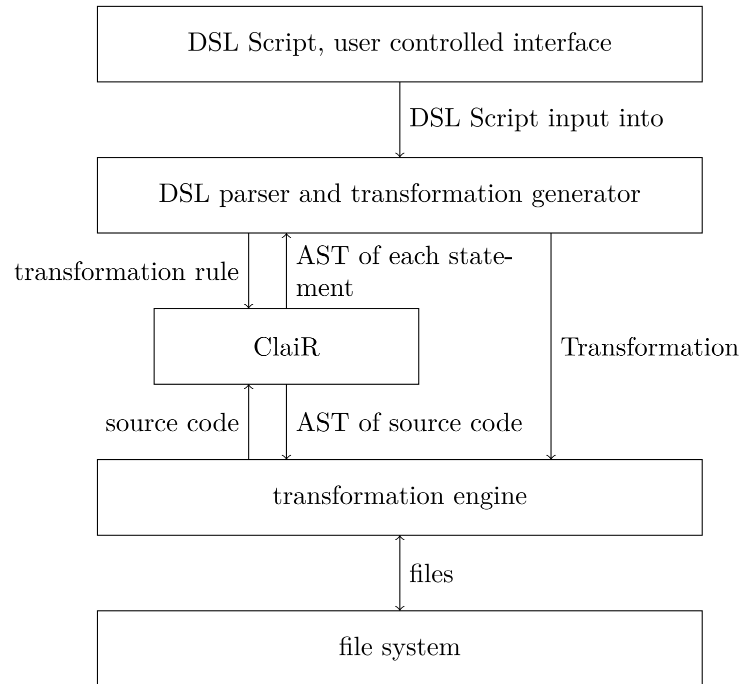

Figure 4.4: Data flow model of the transformation tool.

The transformation engine also communicates with ClaiR, sends the source code into ClaiR and gets the AST of it. The AST here produced by **ClaiR** is the structure of the complete source code file. Additionally, ClaiR also generates some static semantic information of the input source code file including the relevant headers. The **files** between **transformation engine** and **file systems** contains both the original files and the transformed files.

4.4.2 Data Structure

The term **data structure** in this study means the data type used in the implementation of the tool. The data structure design comes from the models defined in **Figure 4.3** and the data flow model in **Figure 4.4**. We focuses on the data structure used in **transformation engine** since the data used in other components are either come from an external software like ClaiR or just plain text. **Figure 4.5** illustrates the class diagram of the data structure in the form of Unified Modeling Language (UML).

The data structure contains 12 classes: `TransformationList`, `Transformation`, `AtomicTransformation`, `SimplePairedTransformation`, `AtomicSource`, `AtomicTarget`, `PairedSource`, `PairedTarget`, `OpenPattern`, `ClosePattern`, `ListOfStatements` and `Statement`.

The class `TransformationList` is a list of transformations, which reflects the data **Transformation** in **Figure 4.5**, as the output data of **DSL parser and transformation generator**. Each `Transformation` contains the transformation rule in terms of code structure (layer 0). In order to store additional information of a transformation, a `Transformation` has 3 attributes: `name`, the name of this particular transformation with type string; `appliesTo`, the applicable file types with type list of string and `addHeaders`, the additional header files need to be inserted in this transformation with type list of string.

Each `Transformation` class has two variants as sub classes, it can either be an `AtomicTransformation` or be a `SimplePairedTransformation`, which reflects the two cases that must be supported in **REQ1** and **REQ2**.

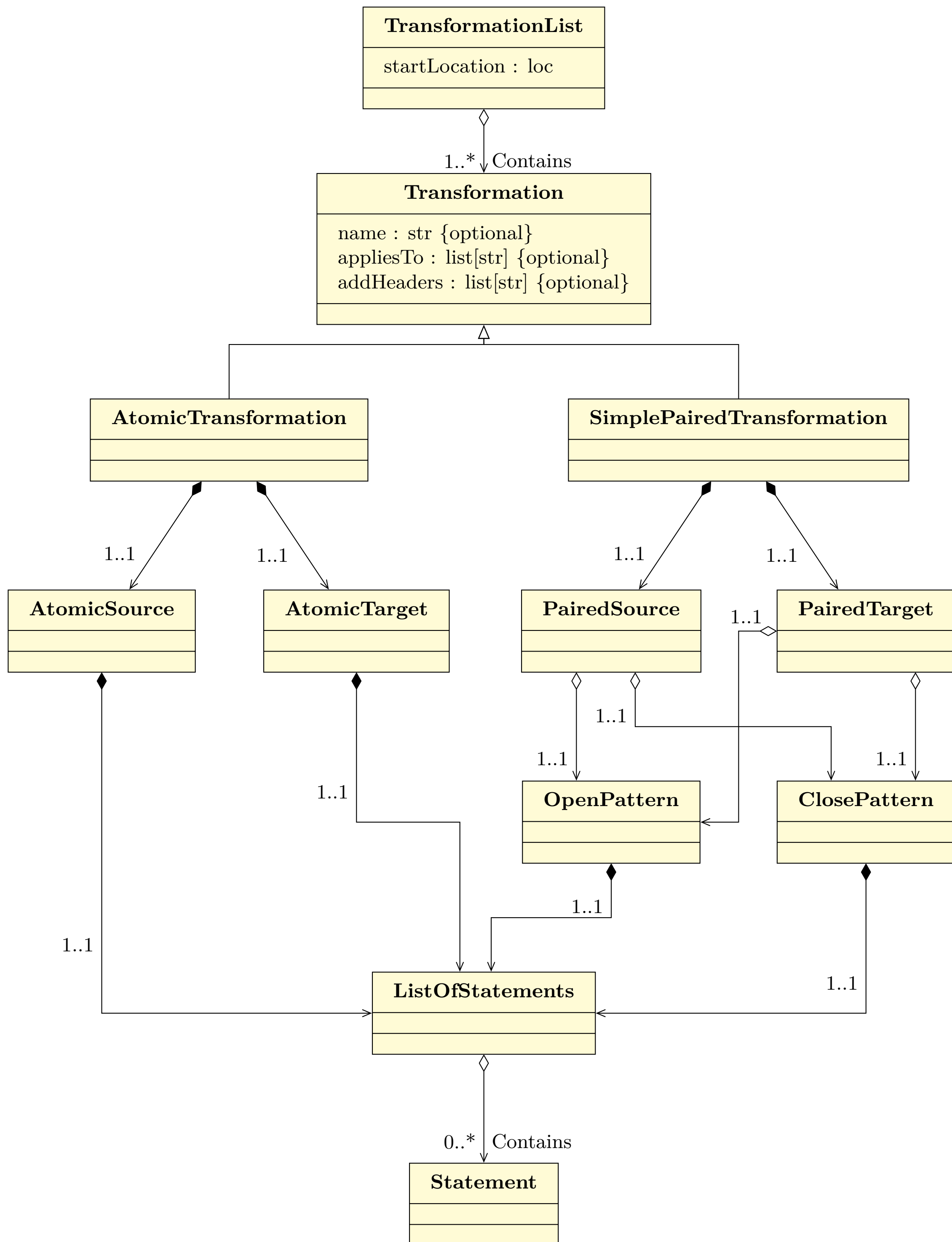

Figure 4.5: The illustration of the data structure of the transformation tool.

An `AtomicTransformation` contains two other classes, namely `AtomicSource`, the source to be found in the source code and `AtomicTarget`, the target of the transformation. Each `AtomicSource` and `AtomicTarget` contains a `ListOfStatements`, which is the structure of source/target.

A `SimplePairedTransformation` also contains two other classes, namely `PairedSource` and `PairedTarget` in **Figure 4.5**. Theses two classes are different from the aforementioned `AtomicSource` and `AtomicTarget` as both of them contain an `OpenPattern` and a `ClosePattern`. The `OpenPattern` and `ClosePattern` are also defined as classes in **Figure 4.5**, reflecting the **open pattern** and **close pattern** described in **Section 4.2.2**. Each of the `OpenPattern` class and `ClosePattern` class contains a `ListOfStatements`.

The `ListOfStatements` class in **Figure 4.5** models a list of statements in the data structure. Here the class **Statement** represents a ClaiR statement, which comes from the ClaiR AST. The `ListOfStatements` contains several `Statement`, the bottom class in **Figure 4.5**.

Listing 4.2 is the code we used to define the data structure in **Rascal**. For each *optional* attribute, it has a default value since the optional attribute in Rascal for a data type can be left empty when instantiating it. In this case the default name of a transformation is unspecified and both `appliesTo` and `addHeaders` are left empty. We also add the complex paired scenario here but leave it empty so it can easily be extended in the future if applicable.

Listing 4.2: The Rascal code to define the top-level data structure.

```
1  data Transformation(
2      str transformationName = "Unspecified name.",
3      list[str] appliesTo = [],
4      list[str] addHeaders = [])
5      = \transformationList(
6          list[Transformation] transformations,
7          loc startLocation)
8      | \atomicTransformation(
9          list[Statement] sourceStatements, list[Statement]
               targetStatements)
10     | \simplePairedTransformation(
11         tuple[list[Statement] openPattern, list[Statement]
               closePattern] patternTupleSource,
12         tuple[list[Statement] openPatternNew, list[Statement]
               closePatternNew] patternTupleTarget)
13     | \complexPairedTransformation(); //h!: complex paired
```

4.4.3 C++ Modernization Language - the DSL

We name the DSL as C++ Modernization Language (CML). The DSL is used to describe the transformation from the user point of view. The user defines all necessary information of the transformation in the DSL script, including syntax transformation and static semantic transformation. Then, the DSL parser of the tool can generate a list of transformation, shown as **Transformation** in **Figure 4.4** and `TransformationList` in **Figure 4.5**, based on the given information provided in the DSL script. Therefore, the design of the DSL contains and reflects the classes listed in **Figure 4.5**. **Figure 4.6** shows a model of the DSL. We name the script of this DSL a *transformation pattern*.

We define a single DSL script as a `TransformationPattern` in **Figure 4.6**, reflects the `TransformationList` in **Figure 4.5**. Each `TransformationPattern` contains exact one `StartLocation` class. The `StartLocation` class reflects the `startLocation` attribute of `TransformationList` class in **Figure 4.5**. A `TransformationPattern` class also contains several `SinglePattern` classes, reflecting the `Transformation` class in **Figure 4.5**.

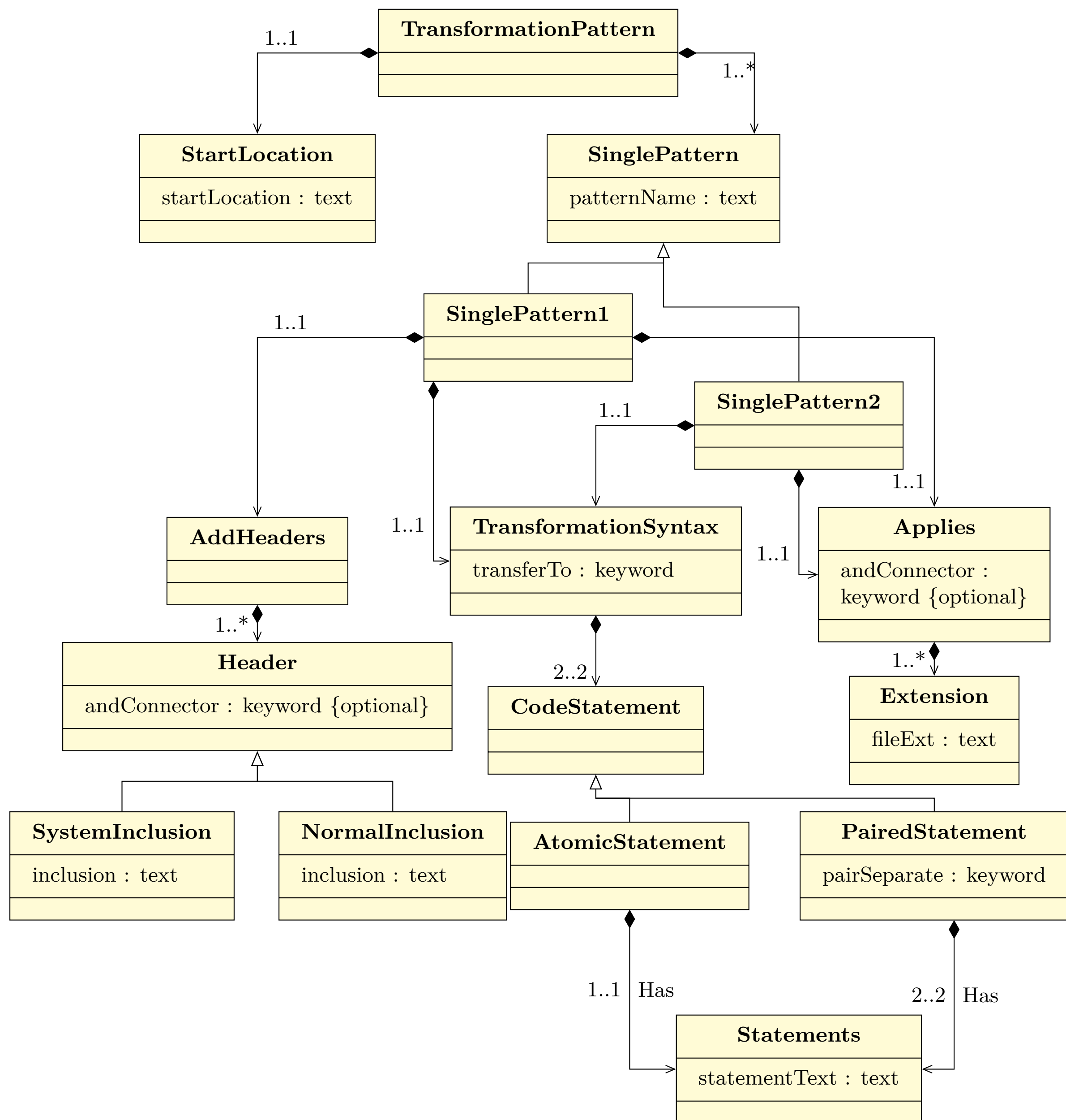

Figure 4.6: The model of the DSL.

A `SinglePattern` class has one attribute named `patternName`. This attribute is plain text and reflects the attribute `name` of `Transformation` in the data structure. Each `SinglePattern` class in **Figure 4.6** has two variants: named `SinglePattern1` and `SinglePattern2`, shown as classes. The only difference between `SinglePattern1` and `SinglePattern2` is that `SinglePattern1` contains a class `AddHeaders` and `SinglePattern2` does not. The `AddHeaders` reflects the attribute `addHeaders` of `Transformation` in the data structure. The other classes, `TransformationSyntax` and `Applies` remain the same. Each `SinglePattern1` or `SinglePattern2` contains exact one `TransformationSyntax` and one `Applies`. The `TransformationSyntax` reflects an `AtomicTransformation` or a `simplePairedTransformation` in the data structure and the `Applies` class reflects the attribute `appliesTo` of `Transformation` in the data structure.

Even though each single pattern contains only one Applies class, it is possible to apply the pattern to several different types of files. The Applies class in **Figure 4.6** composes several classes named Extension with an attribute fileExt, the fileExt is a piece of plain text representing a file extension in the DSL script. For different file extensions, a keyword 'and' is used to distinguish them. The same logic applies to AddHeaders class as well. The real definition of header files is not defined in AddHeaders class but in Header class. Since the headers are used in C/C++ code, we distinguish between a system inclusion, like #include<iostream> and a normal inclusion, like #include "demo.h".

A TransformationSyntax defines the rule of a transformation, the **source** and **target** attributes of an AtomicTransformation or a SimplePairedTransformation. Since we need both a source and a target, each TransformationSyntax class in **Figure 4.6** contains two CodeStatement classes, reflecting a transformation rule from AtomicSource to AtomicTarget or from PairedSource to PairedTarget in the data structure. The two CodeStatement are connected with a keyword transferTo, denoted as '-->'. The CodeStatement differs by the type of the transformation as the syntax of an atomic transformation is different from a simple paired transformation.

The Statements class in **Figure 4.6** reflects the class ListOfStatements in the data structure (**Figure 4.5**). The attribute statementText of Statements reflects the class Statement in the data structure. The statementText is a piece of real C/C++ code. It can either be one statement or be a larger piece of code fragment. During DSL parsing and transformation generation, this piece of text will be input into ClaiR and get a list of statement, as shown in **Figure 4.4**.

An AtomicStatement contains only one Statements class because the atomic transformation is not in pair. A PairedStatement contains two Statements classes: the first one indicates an open pattern of this pair and the second indicates the close pattern of the pair. Each Statements class reflects either OpenPattern or ClosePattern in the data structure. We define a keyword pairSeparate, an attribute in class PairedStatement of **Figure 4.6**, annotated as '...' in the DSL script. In both cases, the plain text statementText accepts an empty string.

Listing 4.3 is an example of the DSL introduced in this section.

This script does the following things:

- Define a start location at X:/FolderY.
- Define an atomic transformation as shown in **Line 7** to **Line 9**, applies to .cpp files, with additional headers <thread>, <chrono> and "demo.h".
- Define a paired transformation as shown from **Line 16** to **Line 28** applies to both .c and .cpp files.

There are some additional aspects in this DSL script:

- The quote mark in **Line 7** and **Line 9** of **Listing 4.3** but not in the paired transformation. This is due to a limitation of ClaiR. The statement in **Line 7** or **Line 9** is a **single statement** without any other declarations. In this case the internal compiler of ClaiR does not know the context of this statement, which means it does not know the detail of __var__ and __var2__ in this transformation. As a result, the compiler will identify these two statements as 2 **declaration statement** instead of **function call**, which is not correct. Thus, we use the quote mark to avoid this issue as in this case the compiler will identify "__var1__" and "__var2__" as a **string**, a basic type of C/C++ and this statement can be correctly parsed as a **function call**.

Listing 4.3: An example of a DSL script.

```
1  *** Comments look like this ***
2  Start patterns :=
3  Start from:= X:/FolderY;
4
5  *** atomic ***
6  Pattern #1 :=
7  [foo1("__var1__", "__var2__")]
8  -->
9  [foo2("__var2__", "__var1__")];
10 Add header := <thread> and <chrono> and "demo.h";
11 Applies to := cpp;
12 End pattern;
13
14 *** paired ***
15 Pattern #2 :=
16 [
17     static int __var1__ = 1;
18     foo_enter(__var1__);
19     ...
20     foo_exit(__var__1);
21 ]
22 -->
23 [
24     static int __var1__ = 2;
25     foo2_enter(__var1__);
26     ...
27     _NULL_;
28 ]
29 Applies to := c and cpp;
30 End pattern;
31 End.
```

For the paired transformation, since `__var1__` is declared in **Line 17**, the compiler knows the context of the code, it can be correctly parsed so we do not need the quote marks.

- A `_NULL_` symbol is also used here in a simple paired scenario to indicate that the target of this transformation does not have a close pattern.

4.4.4 Work Flow

The work flow of a transformation is designed based on the high-level view and data flow model. A complete transformation starts with the user defined DSL script, and ends with file modifications. **Figure 4.7** illustrates the complete work flow from writing a DSL script to the modification of a file.

The start point is the only input entry as described before, the DSL script. After the DSL script has been input into the system, a DSL parser parses the script and feeds it into a transformation generator to generate the `TransformationList`. This step reflects the **DSL parser and transformation generator** in the data flow (**Figure 4.4**). The following steps are iteration based, namely iterate over all transformations in the generated list. Here the transformation in each iteration is a single transformation.

For each transformation, we distinguish them by its type, namely atomic or simple paired. Different types use different algorithms to detect, match and transform. The algorithms used in each process of the work flow will be described in detail in **Section 4.4.5**.

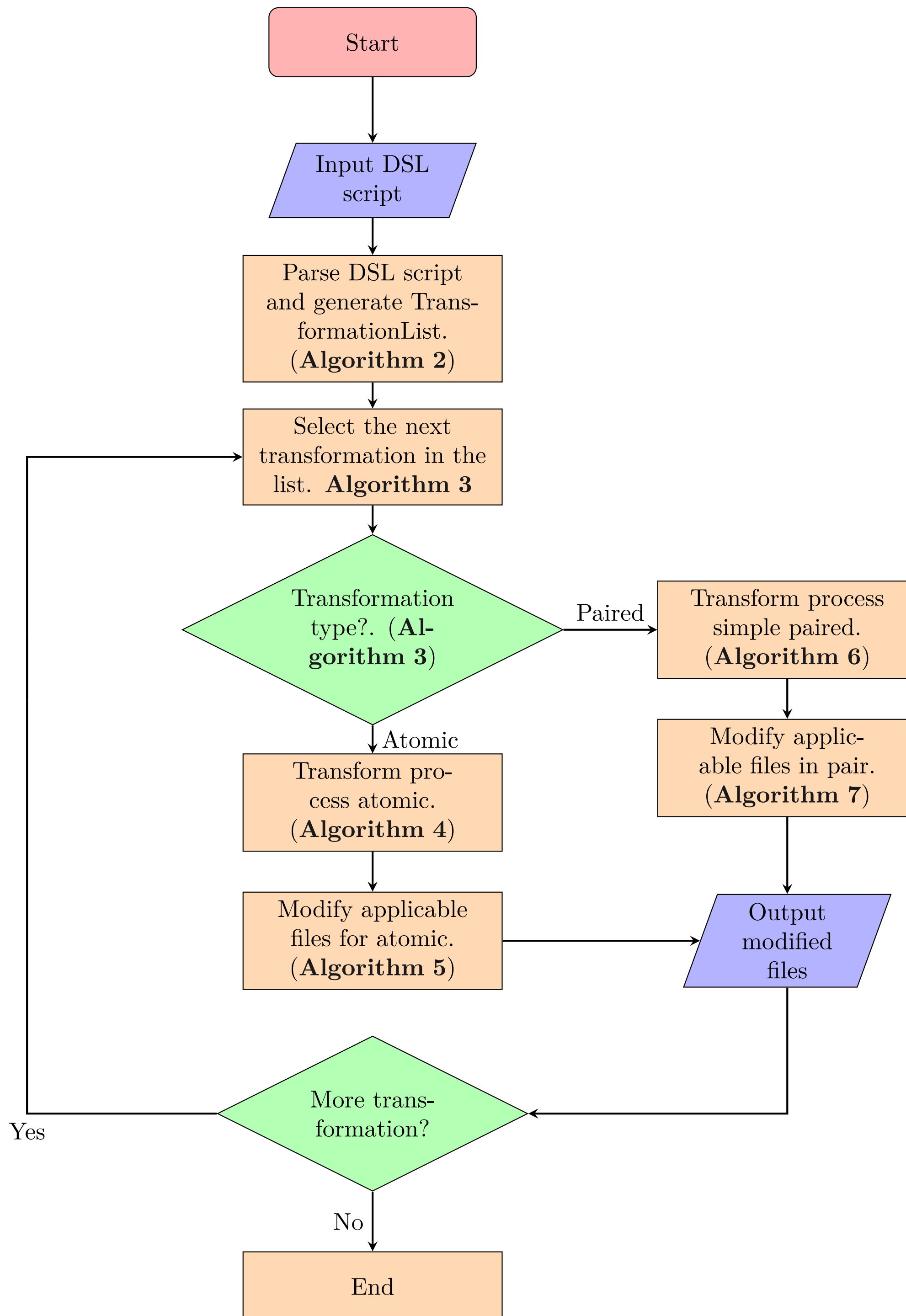

Figure 4.7: The workflow design of the transformation tool.

The process **ParseDSL script and generate TransformationList** is described in **Algorithm 2**. The **Select the next transformation in the list** and the condition block is described in **Algorithm 3**. The **Transform process atomic** and **Modifyapplicable files for atomic** are described in **Algorithm 4** and **Algorithm 5**. The **Transform process simple paired** and **Modify applicable files in pair** are described in **Algorithm 6** and **Algorithm 7**.

4.4.5 Algorithms

The algorithms describe in this section are the detailed operations for a transformation tool. Seven algorithms are described in this section:

- **Algorithm 1**, top level transform. `transform(l)`
  This algorithm describes the operation at the top level, which is the method to be invoked by users.
- **Algorithm 2**, parse a DSL and generate transformation. `generateTransformation(l)`
  This algorithm describes the operations to parse a DSL script from a given location, and then generate transformation details based on this DSL script.
- **Algorithm 3**, transform a list. `transformList(listOfTransformation)`
  This algorithm is used to apply transformations from a list.
- **Algorithm 4**, apply an atomic transformation. `transformBasedOnType(t, startLocation)`
  This algorithm describes the necessary steps to apply an atomic transformation.
- **Algorithm 5**, modify a file for an atomic transformation. `modifyFileAtomicVersion(source, target, file, metadata)`
  This algorithms describes the details about the syntactical transformation of an atomic transformation in a file.
- **Algorithm 6**, apply a simple paired transformation. `transformBasedOnType(t, startLocation)`
  This algorithm describes the necessary steps to apply a simple paired transformation.
- **Algorithm 7**, modify a file for a simple paired transformation. `modifyFileSimplePairedVersion(pairSource, pairTarget, file, metadata)`
  This algorithms describes how to apply the syntactical transformation on a file of a simple paired transformation.

Top level operation

According to the requirements, there should be only one operation for each transformation at the top level. In this case top level transform algorithm (**Algorithm 1**) represents this operation. The algorithm has only one input, which is the location of the DSL script describing the details of this transformation. This algorithm first creates an empty log file for logging the transformation and then call method `transformList` (**Algorithm 3**) to perform a transformation generated by method `generateTransformation` (**Algorithm 2**).

Algorithm 1: Top level transform. `transform(l)`

Data: {`l`}, the location of the DSL script to be input.

Result: The modified files with code transformed based on the transformation rules written in the input DSL script.

```
1 createLogFile();
2 transformList(generateTransformation(l));
```

Parse DSL and generate transformation

This algorithm (**Algorithm 2**) is used to parse a DSL script written by users, and then generate a transformation based on the details in it. The algorithm has one input, which is the location of the DSL script; and one output, which is a list contains all single transformations defined in the DSL script. The first step of this algorithm is to parse the DSL script into a parse tree. After that,

every node in that parse tree shall be visited and processed, where each node represents a single transformation. Every single transformation is generated according to its type, namely atomic or simple paired. The generated transformations are added to the list.

Algorithm 2: Parse a DSL and generate transformation. `generateTransformation(l)`

Input: {`l`}, the location of the DSL script to be input.

Output: A {`listOfTransformation`} contains all transformation rules described in the input DSL script.

```
1 parseTree ⟵ parse(transformationPattern, l);
  // parse is a native Rascal method to generate a parse tree.
2 listOfTransformation ⟵ empty_list;
3 foreach Node n ∈ parseTree ∧ !visited do
4     if n.transformation is atomic then
5         addToList(listOfTransformation,
           atomicTransformation(transformationDetail(n.transformation)));
6     else
7         addToList(listOfTransformation,
           simplePairedTransformation(transformationDetail(n.transformation)));
8 return listOfTransformation;
```

Transform a list of transformation

This algorithm (**Algorithm 3**) is also called in the top-level algorithm. The reason to have this algorithm is that the output of DSL parser and transformation generator is a list of different single transformations. Hence, an algorithm is needed to iterate over every single transformation (the loop in **Figure 4.7**). As a result, the input of this algorithm is a list with all transformations generated by **Algorithm 2**. Furthermore, from the list, the start location can also be extracted. In each iteration of this algorithm, a method is invoked based on the type of that single transformation.

Algorithm 3: Transform a list. `transformList(listOfTransformation)`

Data: {`listOfTransformation`}, a list contains all transformation rules described in the DSL script.
{`startLocation`}, the start location of the transformation list, is extracted from `listOfTransformation`.

Result: The modified files start from `startLocation` with code transformed according to the given Transformation.

```
1 foreach singleTransformation ∈ transformations do
2     transformBasedOnType(singleTransformation, startLocation) ;
```

Transform an atomic transformation

An atomic transformation is presented in two algorithms: the first one (**Algorithm 4**) describes each step of that transformation and the second one (**Algorithm 5**) describes how to apply the **syntactical transformation** on a file. If the input transformation `t` is an atomic transformation, then the transformation engine will invoke this algorithm. The input data of this algorithm is the

transformation detail and the start location of the transformation. Additionally, another variable named `additionalMacros` is also used to let the transformation tool access the code under all conditional complications. This macros are pre-defined in the transformation tool, as an internal variable.

An atomic transformation is divided into three steps: generate a list of files according to the applicable file extensions, start from the start locations; then for each file, apply the syntactical transformation with method `modifyFileAtomicVersion` (**Algorithm 5**) for all additional macros and at last apply static semantical transformation by adding additional headers to that file.

Algorithm 4: Transformation algorithm for an atomic transformation. `transformBasedOnType(t, startLocation)`

Data: {`t`}, an `atomicTransformation` contains elements:
{`sourceStatement`}, the statement in the source code file that will be modified;
{`targetStatement`}, the statement that will be transformed to in the source code file;
{`appliesTo`}, the appliciable file extensions;
{`addHeaders`}, the additional header files to be inserted in this transformation.
{`startLocation`}, the start location of this transformation.
{`additionalMacros`}, the additional preprocessor macros according to the platforms.

Result: The modified files start from `startLocation` with atomic transformations have been performed on the files with extensions described in `appliesTo`.

```
  // Generate a list of all files corresponding to the file extension.
1 listOfFilesToCheck ⟵ empty_list;
2 foreach e ∈ t.appliesTo do
3   addToList(listOfFilesToCheck, getFileList(startLocation, e)) ;
4 foreach f ∈ listOfFilesToCheck do
    // Go over all additional macro definitions.
5   foreach macro ∈ additionalMacros do
      // Modify file.
6     modifyFileAtomicVersion(t.sourceStatement, t.targetStatement, f, parseCpp(f,
       macro));
7   addHeader(t.addHeaders, f);
```

To modify a file for an atomic transformation, it is necessary to know the **structure of source**, **structure of target**, the file location and full AST of the file. In **Algorithm 5**, these four elements are the input of the algorithm. Where in our case, the AST is stored in the variable named `metadata` together with other information of the file including the headers included and the expanded macros. This algorithm first reads the file into a list of bytes, then matches the structure of source with the full AST of the file, trying to find the statements **have the same structure** as the source defined in the DSL script, then stores them into a list. This list contains the statements that will be modified. After that, the algorithm iterates over each statement in that list, and transforms the corresponding statement from syntactic level. The syntactic transformation starts with creating a map for **meta variables**, helping transfer the value which is represented by a meta variable directly into the new code. The following step comments out the old statement in file, followed by a step to insert the target statement. The insertion of a target statement is handled by a code generator that generates the code from AST to real C++ code. In this study, we use Rascal to implement the algorithms. Some exemplar implementations of the matching step can be found in **Appendix C**.

Algorithm 5: Modify files for an atomic transformation. `modifyFileAtomicVersion(source, target, file, metadata)`

Data: {`source`}, the AST represents the structure of the statement that has to be changed in the source code.
{`target`}, the AST represents the structure of the statement that will be transformed to.
{`file`}, the location of the file which is currently working with.
{`metadata`}, the data generated by ClaiR of the file which is currently working with containing the AST of the file and other information like the headers, macro expansions.

Result: The modified file at location `file`.

```
1  fileByte ⟵ readFileBytes(file);
   // Match statements that will be changed.
2  matchedStatements ⟵ matchStatement(source, metadata);
3  foreach i ∈ matchedStatements do
4      metaVarMap ⟵ createMetaVarMap(source, i, metadata);
5      commentOut(i);
6      if size(target) > 0 then
7          if isEmpty(metaVarMap) then
8              fileByte ⟵ addString(fileByte, statementToStr(target), i.location.offset);
9          else
10             fileByte ⟵ addString(fileByte, statementToStr(target), metaVarMap,
                i.location.offset);
11 writeFileBytes(file, fileByte);
```

Transform a simple paired transformation

To perform a simple paired transformation, two algorithms are needed, which is similar to an atomic transformation. The first algorithm (**Algorithm 6**) describes the steps, while the second one (**Algorithm 7**) describes the syntactical transformation to files. The internal variable `additionalMacros` is also used in a simple paired transformation to help the transformation tool accessing all possible conditional complications. A simple paired transformation also contains three steps: generating a list of files according to the applicable file extensions, starting from the start locations; then for each file, applying the syntactical transformation with method `modifyFileSimplePairedVersion` (**Algorithm 7**) for all additional macros and finally applying static semantical transformation by adding additional headers to that file.

Applying a syntactical transformation (modifying file) for a simple paired transformation is different from that for an atomic transformation. The input data of this algorithm (**Algorithm 7**) also contains four elements: **structure of source pair**, **structure of target pair**, the file location and full AST of the file. In this case the both source and target store open pattern and close pattern separately. The algorithm first reads the file, then uses the structure of source pair to match corresponding pairs in the original file ad stores the matched statements in a list. Additionally, the irrelevant statements other than the pair itself are removed from the list, so that all the statements in that list are relevant. Furthermore, this list also stores additional information like the insert position for a new open pattern/close pattern. After that, the algorithm iterates over all statements matched in the list, creates a map for meta variables, comments out the statement. During each iteration, if the location of the statement in this iteration is the position to insert the new open/close pattern, then the code generator will generate new C++ code according to corresponding target pair. Some exemplar implementations to match a pair of statements and to simplify the matched lists can be found in **Appendix C**.

Algorithm 6: Transform a simple paired transformation. `transformBasedOnType(t, startLocation)`

Data: {`t`}, a `simplePairedTransformation` contains elements:
{`pairSource`}, the pair of statements in the source code file that will be modified;
{`pairTarget`}, the statement that will be transformed to in the source code file;
{`appliesTo`}, the appliciable file extensions;
{`addHeaders`}, the additional header files to be inserted in this transformation.
{`startLocation`}, the start location of this transformation.
{`additionalMacros`}, the additional preprocessor macros according to the platforms.

Result: The modified files start from `startLocation` with simple paired transformation performed on the files with extensions desribed in `appliesTo`.

```
   // Generate a list of all files corresponding to the file extension.
1  listOfFilesToCheck ⟵ empty_list;
2  foreach e ∈ t.appliesTo do
3  │  addToList(listOfFilesToCheck, getFileList(startLocation, e)) ;
4  foreach f ∈ listOfFilesToCheck do
5  │  foreach macro ∈ additionalMacros do
6  │  │  modifyFileSimplePairedVersion(t.pairSource, t.pairTarget, f, parseCpp(f, macro));
7  │  addHeader(addHeaders, f);
```

Algorithm 7: Modify files for a simple paired transformation. `modifyFileSimplePairedVersion(pairSource, pairTarget, file, metadata)`

Data: {`pairSource`}, the pair that has to be modified in the source code.
{`pairTarget`}, the pair that will be transformed to.
{`file`}, the location of the file which is currently working with.
{`metadata`}, the data generated by ClaiR of the file which is currently working with containing the AST of the file and other information like the headers, macro expansions.

Result: The modified file at location `file`.

```
1  fileByte ⟵ readFileBytes(file);
2  toBeModified ⟵ generateStatementList(pairSource, metadata);
3  metaVarMap ⟵ empty_map;
4  foreach s ∈ toBeModified do
5      addToMap(metaVarMap, s.openPatternInsertPlace, createMap(pairSource,
        s.statementToChange, metadata));
6      fileByte ⟵ commentOut(s.statementToChange);
7      if s.statementToChange.location = s.openPatternInsertPlace then
8          if isEmpty(metaVarMap.getKey(s.openPatternInsertPlace)) then
9              foreach i ∈ pairTarget.openPatternNew do
10                 fileByte ⟵ addString(fileByte, statementToStr(i),
                    s.statementToChange.location.offset);
11         else
12             fileByte ⟵ addString(fileByte, statementToStr(i,
                metaVarMap.getKey(s.openPatternInsertPlace)),
                s.statementToChange.location.offset);
13     if s.statementToChange.location = s.closePatternInsertPlace then
14         if isEmpty(metaVarMap.getKey(s.openPatternInsertPlace)) then
15             foreach i ∈ pairTarget.closePatternNew do
16                 fileByte ⟵ addString(fileByte, statementToStr(pairTarget.i),
                    s.statementToChange.location.offset);
17         else
18             fileByte ⟵ addString(fileByte, statementToStr(i,
                metaVarMap.getKey(s.openPatternInsertPlace)),
                s.statementToChange.location.offset);
19 writeFileBytes(file, fileByte);
```

4.5 Limitations and Future Work

There are several limitations have been found during the design. These limitations concerning the supported transformation type and the format of transformation rules.

4.5.1 Limitation of Supported Transformation Type

At this moment, only two types of transformation, namely atomic transformation and simple paired transformation, are supported by the transformation tool. The complex paired transformations are still not supported. For a complex paired scenario, or even complex grouped scenario, the obstacle is to track data flow between statements. Referring back to **Listing 3.3**, since a statement between an open pattern and a close pattern uses the value produced by the open pattern, the transformation of the open/close pattern should also check all values produced by the open/close pattern to ensure static semantic correctness. However, in ClaiR there is no such a feature to track the data flow between statements. Furthermore, if the type of value produced by open/close pattern is changed during a transformation, then the statement using that value might need a completely new implementation to support the newly-introduced type, which is hardly to be finished automatically now.

4.5.2 Limitations of the Format of Transformation Rules

Another limitation is the format of transforamtion rules. In the current design, each single transformation uses a *source-target* format to define the original code and target code of a transformation. For each transformation the *source-target* is only a **one-to-one** mapping. Thus, for a **one-to-many**, **many-to-one** or **many-to-many** transformation, it must be spitted into multiple **one-to-one** transformations. This is not a limitation for Philips' code that is used in this study, but it will be one for generic cases.

4.5.3 Future Work

The future work from design point of view can focus on extending the models. One approach is to focus on investigating a data flow tracker of the C/C++ program. This tracker can use the idea of **Object Flow Diagram** produced by Rascal for Java analysis. An object flow diagram tracks all object flow (data flow) in a Java program. This object flow can be an object flow from a class to a method or a method to a field. With the help of this kind of data flow tracker, it is possible to conduct further research on transforming a complex pair scenario. The other approach is to focus on investigating the format of transformation rules. By extending the model of transformation rules, it is promising to support a many-to-many transformation.

4.6 Conclusion of Tool Design

This chapter explains the design process of this DSL based code transformation tool in detail, from the requirement analysis to a concrete choice of algorithms. Followed by the design results, including a detail explanation of each algorithm and its implementation.

We have designed a DSL based transformation tool for the code transformation. The tool can now handle two types of transformation: the one-to-one atomic transformation for single statement; and the one-to-one paired transformation for a pair of statements. To apply the transformation, the user only needs to write one single DSL script that includes all transformation rules. Furthermore, the DSL script includes the applicable file extensions and additional headers.

Referring back to our two research questions in this chapter: **What are the key aspects to make a transformation of OSAL APIs?** and **What can be done to make the transformation as automated as possible?**. This chapter has make the answer clearly: the key aspects of a

transformation contains: **a transformation rule**, including a source and a target, applicable files and additional header; **a start location**, limiting the scope of files to be transformed and **type of the transformation**. The answer to our second research question is reducing the number of input entries. In this study, we chose to make a DSL as the only entry of the transformation.

Chapter 5

Application and Case Study

To verify the output, the implementation of the design result should be applied to some examples. This chapter presents details about the application of the transformation tool, in the form of case study. We apply the transformation tool on Philips' code base, to get evidence that the transformation tool can be directly used in real industry scenario.

5.1 Apply an Atomic Transformation to Philips' Code Base

We apply the atomic transformation on two different directories: one is named `gsc` and the other is `GeoCVEmb` directory. We choose `gsc` because previous investigations about `sleep` construct were done on file in this directory. It is informative to apply it on the same directory automatically as a "reproduction". The reason to choose `GeoCVEmb` is that this directory is the complete code base of Philips' positioning software. Running a transformation on the complete code base can help to measure the performance of it.

We use two metrics to proof the correctness of an atomic transformation. The first metric is the compile result, the other is test result of Philips' module test mechanism. A passed compile result indicates the transformation does not break the syntax as well as static semantics of the code. A passed test result indicates the equivalence of the functionality behavior. Thanks to Philips' modular testing mechanism, we can test the transformed code automatically to verify the functionality. Furthermore, there is a tool to automatically detect a changed file and run the required tests, named Test Impact Analyzer (TIA). With the help of TIA, we do not run all tests in Philips' modular testing mechanism.

Transformation of the sleep function on `gsc`

Listing 5.1 is the script to apply the transformation on directory `gsc`. The transformation rule is defined the same as we described in **Table 4.2**. The argument name is also covered with quote marks because they are not declared in this particular content.

The transformation tool examined 94 files, and 25 files have been changed with 257 statements modified, indicating the possibility of applying the transformation on a number of files. The running time of this particular transformation is around **30 minutes**.

In **Figure 5.1** the header part of the transformation was successful, showing the correct insertion of the two headers. The text with red color on the left side of the figure shows the newly inserted headers, which is the same as we defined in the DSL.

Listing 5.1: The DSL script to transform the Sleep function on Philips' code base.

```
1  *** trial ***
2  Start patterns :=
3
4  Start from := Y:/PosCore/GeoCVEmb/gsc/src;
5
6  *** previous atomic ***
7  Pattern #2 :=
8  [TOS_p_TSK_sleep("__var__");] --> [std::this_thread::sleep_for(std::chrono::
       milliseconds("__var__"));];
9  Add header := <chrono> and <thread>;
10 Applies to := cpp;
11 End pattern;
12
13 End.
```

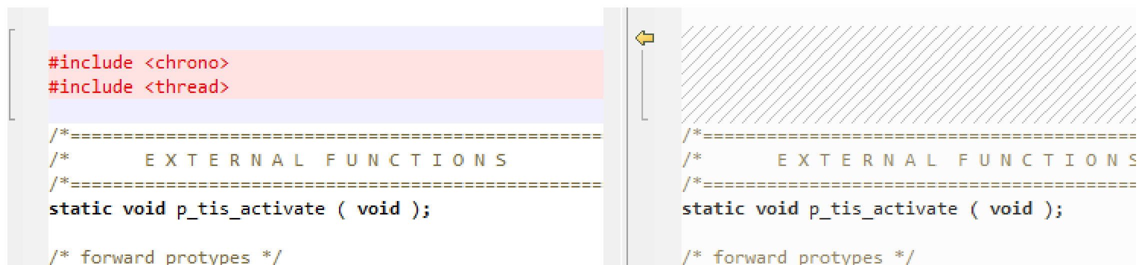

Figure 5.1: A file comparison for adding inclusion.

Figure 5.2 presents the function transformation. It should be noted that in **Figure 5.2**, the passed variable is a pointer access, which is also supported by the transformation tool.

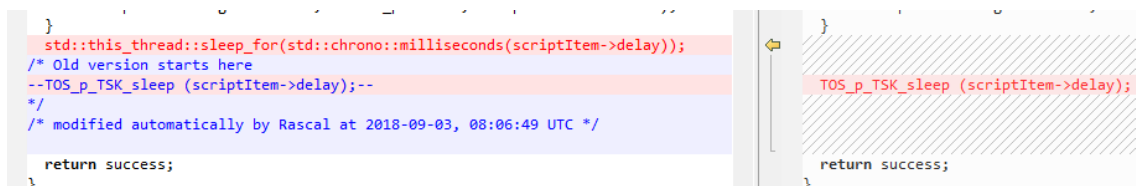

Figure 5.2: A file comparison for passing a pointer access.

Figure 5.3 is the test result from TIA running. TIA considers to run one test module named `fsco_pcm` for these changes and the test result is successful.

```
Running: NUnitSTXExecutor.STXExecutor.STX_OBJ_TN_fsco_pcm
NUnitSTXExecutor.STXExecutor.STX_OBJ_TN_fsco_pcm succeeded.
All tests finished successfully

Y:\PosCore\Build>
```

Figure 5.3: The test result for transforming the sleep function.

Transformation of the sleep function on complete Philips' code base

To apply the transformation of `sleep` function on a complete code base, we only change the `Start location` field in DSL script. **Listing 5.2** is the DSL script of the transformation we applied to the complete Philips' code base. The transformed code can be built correctly and pass Philips' modular test using TIA as the previous case. In this case, the transformation tool examined 808 files and applied 731 modifications to statements. The running time of this transformation is around **4 hours**.

Listing 5.2: The DSL script to transform the Sleep function on Philips' code base.

```
1  *** trial ***
2  Start patterns :=
3
4  Start from := Y:/PosCore/GeoCVEmb/gsc/src;
5
6  *** previous atomic ***
7  Pattern #2 :=
8  [TOS_p_TSK_sleep("__var__");] --> [std::this_thread::sleep_for(std::chrono::
       milliseconds("__var__"));];
9  Add header := <chrono> and <thread>;
10 Applies to := cpp;
11 End pattern;
12
13 End.
```

5.2 Apply a Simple Paired Transformation to Philips' Code Base

We apply the simple paired transformation on `gsc` directory since Philips' engineers conducted their investigations about `mutex` construct on the same directory. We produce two different transformations: one is exactly the same as the `mutex` construct investigated by Philips' engineers, and the other is a split version to match more occurrences in the code.

We use one metric to verify the output: the compile result. A passed compile result (successful build) indicates that the transformation is successful. Philips' modular testing mechanism is not applicable there for the following two reasons: according to TIA no suitable tests can be applied to the files with `mutex` constructs in `gsc` directory; according to Philips' engineers the transformation they have investigated about `mutex` construct still contains some limitations that cannot be tested.

Transformation of mutex construct in one step

Listing 5.3 is the script to apply a transformation of `mutex` construct from Philips' engineers' investigations. Note that the _NULL_ at the end of the close pattern means there is no need to insert a new close pattern for this case.

Figure 5.4 shows the file comparison of the open pattern. It can be concluded that the open pattern of transformation was completed successfully since the standard `mutex` construct was inserted correctly and Philips' version was commented out. For this new open pattern a '{' was also added to limit the scope.

Listing 5.3: The DSL script to transform the mutex construct on Philips' code base.

```
1   *** trial ***
2   Start patterns :=
3
4   Start from := Y:/PosCore/GeoCVEmb/gsc/src;
5
6   *** mutex ***
7   Pattern #3 :=
8   [
9       static TOS_t_MTX_object __var2__ = 0;
10      if(__var2__ == 0) {
11          __var2__ = TOS_p_MTX_create();
12      }
13      TOS_p_MTX_enter(__var2__);
14
15      ...
16
17      TOS_p_MTX_exit(__var2__);
18  ]
19  -->
20  [
21      std::mutex __var2__;
22      std::lock_guard<std::mutex> lock(__var2__);
23
24      ...
25
26      _NULL_;
27  ];
28  Add header := <mutex>;
29  Applies to := cpp;
30  End pattern;
31
32  End.
```

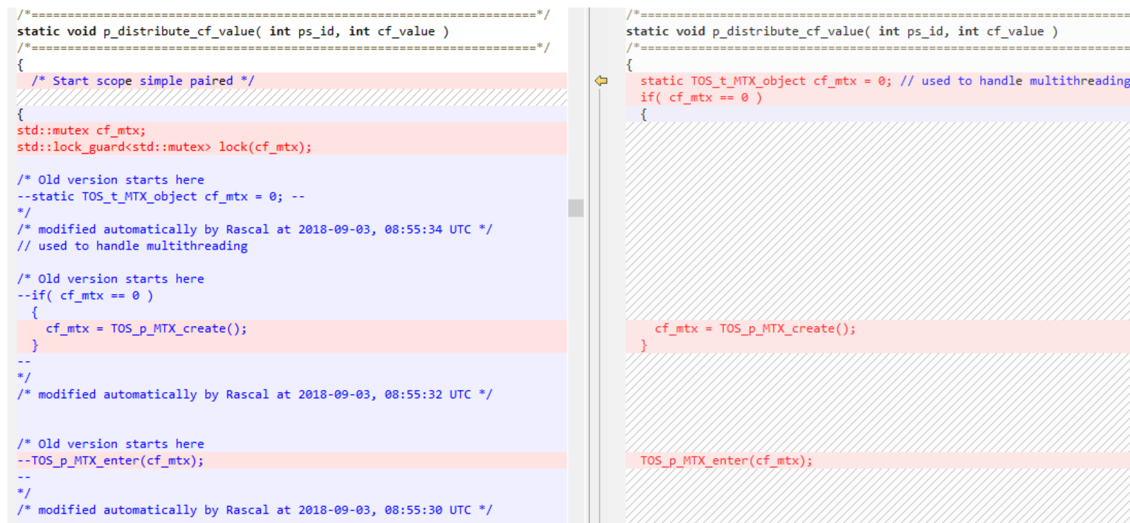

Figure 5.4: File comparison of the open pattern.

Figure 5.5 presents the file comparison of the close pattern. Since standard `mutex` construct does not require a close pattern, the new close pattern is a '}' symbol, which also reflects the `_NULL_` in the DSL script.

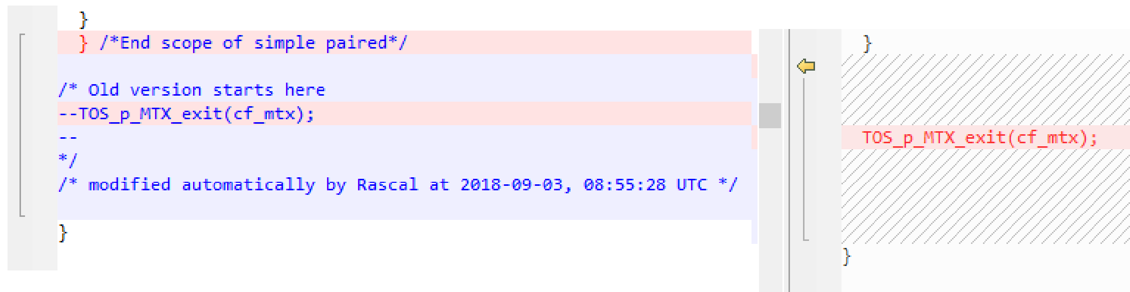

Figure 5.5: File comparison of the close pattern.

The number of files examined by the transformation tool is 94, with one statement modified. The total running time of this transformation is roughly **27 minutes**.

Transformation of mutex construct in several steps

The transformation of `mutex` construct can also be applied in several steps by spiting the transformation investigated by Philips' engineers. The reason to split the transformation is that in the real code base, not all `mutex` occurrences have a *declaration statement* and an *if statement* in the open pattern like the one in the example of **Listing 5.3**. Some of them have only one *function call* as open pattern and the *declaration statement* is located somewhere else in the file. Thus, splitting the transformation into several steps can help transform this situation. **Listing 5.4** is the DSL script to transform `mutex` construct in several steps.

This particular transformation is a combination of three steps, as three single transformations named `Pattern #4`, `Pattern #5` and `Pattern #6` in **Listing 5.4**. The first step transforms a `mutex` construct starts with an *if statement* and followed by a pair of *function calls*. The second step transforms a `mutex` construct with only a pair of *function calls*. The last step removes the original declarations from Philips' OSAL.

In this particular transformation, the transformation tool examined 94 files and applied **22 modifications** to statements. Since the DSL script contains three single transformations, the total number of files examined was 280. Each file was examined three times. The running time of this transformation is around **1.4 hours**. The transformed code can be built successfully.

Listing 5.4: The DSL script to transform the mutex construct on Philips' code base in several steps.

```
1  Start patterns :=
2
3  Start from := Y:/PosCore/GeoCVEmb/gsc/src;
4
5  *** mutex with only if and pair ***
6  Pattern #4 :=
7  [
8      if ("__var3__" == 0) {
9          "__var3__" = TOS_p_MTX_create();
10     }
11     TOS_p_MTX_enter("__var3__");
12     ...
13     TOS_p_MTX_exit("__var3__");
14 ]
15 -->
16 [
17     std::mutex __var3__;
18     std::lock_guard<std::mutex> lock(__var3__);
19     ...
20     _NULL_;
21 ];
22 Add header := <mutex>;
23 Applies to := cpp;
24 End pattern;
25
26 *** mutex with only pair ***
27 Pattern #5 :=
28 [
29     TOS_p_MTX_enter("__var4__");
30     ...
31     TOS_p_MTX_exit("__var4__");
32 ]
33 -->
34 [
35     std::mutex __var4__;
36     std::lock_guard<std::mutex> lock(__var4__);
37     ...
38     _NULL_;
39 ];
40 Add header := <mutex>;
41 Applies to := cpp;
42 End pattern;
43
44 *** remove TOSMTX declarations ***
45 Pattern #6 :=
46 [static TOS_t_MTX_object __var5__ = 0;] --> [];
47 Applies to := cpp;
48 End pattern;
49 End.
```

5.3 Discussion

The discussion section about case study includes two aspects: the correctness of the results, and the performance of the transformation tool.

5.3.1 Discussion of the Correctness

From the cases we have applied to Philips' code base, the results are positive from a correctness perspective.

In both applications of atomic transformation, the transformed code can be built successfully and can pass Philips' modular tests. Hence, for `sleep` construct it is fully transformable with the tool designed in this thesis. The transformation of a **single statement**, namely the atomic transformation, is always the same from the transformation engine's point of view. The transformation engine only matches the semantic of transformation rules described in the DSL script. This successful result of `sleep` construct indicates the correctness of the transformation engine. Hence, we can conclude that if transformation rules are defined correctly in DSL scripts, the transformation engine can ensure the correctness of that transformation.

Same logic applies to simple paired transformations. The transformation engine has been illustrated to handle the semantics correctly. Another difference between different simple paired transformations is the quantity and sequence of statements in a pair. The results also indicates that the transformation engine has the ability to match and transform statements with different amount and orders, which ensures the correctness of other paired transformations.

Furthermore, for transformations other than the two cases studied in this thesis, the correctness of the transformation is also verifiable. The verification of syntax level and static semantics level can be done by compilers, and the verification of dynamic semantics level can be done with testing mechanisms.

5.3.2 Discussion of the Performance

The metric we use to measure the performance is **running time** of a transformation. In the applied transformations, the running times are different. We list the result of four transformations in **Table 5.1**, with aspects including transformation type, number of examined files, number of modified statements and the running time. A **transformation type** is either atomic transformation or simple paired transformation. **Number of examined files** shows the number of examined files by the transformation in total of a transformation. If one transformation contains several single transformations, then the number of examined files is the sum of files examined in each single transformation. **Number of modified files** is the number of statements that have been modified among all files in a transformation. **Running time** is the exact running time of a transformation, starting from the invocation of the transform method.

Table 5.1: A summary of running time in different transformation.

Transformation type	#Examined files	#Modified statements	Running time (hours)
Atomic transformation	94	257	0.5
Atomic transformation	808	731	4
Simple paired transformation	94	1	0.45
Simple paired transformation	282	22	1.4

Comparing the first row and the second row with the same transformation type in, but different number of examined files in **Table 5.1**. In the second row, 8.5 times more examined files resulting in nine times more running time than that in the first row even though the number of modified statements is only three times more than that in the first row. This result shows a linear relation between the number of examined files and the running time. Roughly, the transformation tool can examine 200 files per hour.

The comparison between the first row and the third row shows that there is no significant difference in running time between an atomic transformation and a simple paired transformation (94 files with 0.5 and 0.45 hour). Furthermore, it also indicates that different number of modified statements does not affect the running time.

The third row and the fourth row in **Table 5.1** differ in number of examined files and number of modified statements. The number of modified statements is the total number of modifications have been made to all files. The result of this comparison is similar to the result of comparison between the first row and the second row. The running time of the fourth row is three times as that of the third row since the number of files examined in the fourth row is roughly three times as that of the third row. Even though the number of modified statements in the fourth row is 22 times as the number in the third row, it does not affect the running time.

It can be found from the comparison between rows that the dominate part affecting the transformation performance is the **number of examined files** in a software system. For each transformation, larger number of files to be examined resulting in longer running time. The other aspects, namely *number of statements modified* and *transformation type* do not affect the transformation performance. The overall speed of a transformation in a large-scale software system is approximately **200 files per hour**.

The results above hint that the bottleneck of the performance is in operations related to file system. In the transformation tool, there are two operations related to file system: the parsing step provided by ClaiR and the output step after modifying statements. Since for the same number of files, different number of modified statements does not affect the running time, we can conclude that the output step does not affect the performance. Hence, ClaiR restricts the running time of a transformation.

5.4 Conclusion of Case Study

In the case study phase, we have applied two types of transformation: the atomic transformation and the simple paired transformation to Philips' code base. The atomic transformation that has been applied is the `sleep` construct and the simple paired transformation applied is the `mutex` construct. For each type, we have applied two variants. All the four applications have presented positive results, namely the transformed code can be built successfully. We can conclude that the transformation tool can handle an arbitrary transformation fulfill the properties of these two transformation types.

Furthermore, we have tested the performance of a transformation, which is the running time. The investigation is that the dominate aspect that can affect the transformation performance is the number of examined files of a transformation. From our experiments, the transformation tool can run at a process rate of 200 files per hour. The bottleneck of this transformation tool is ClaiR, the external component that is used to generate AST of a file.

Chapter 6

Conclusions and Suggestions

In this thesis, we have conducted an extensive research on legacy code of Philips' code base and the possibilities to modernize the legacy code automatically. The ultimate goal of this study is to investigate whether it is possible to modernize the legacy code in large-scale software systems in an automated way. The specific case discussed in this thesis is the Operating System Abstraction Layer (OSAL) in Philips' code base. In order to reach this ultimate goal, we started with an extensive examination on the positioning code in Philips' code base, categorized the types and functionalities of the OSAL functions, calculated the number of occurrences, the distribution of the occurrences and summarized the ways to use those OSAL functions. Then we chose several occurrences as examples, from the simplest one to a complex one, designed and implemented a Domain-Specific Language (DSL) based transformation tool from the examples. We applied the automated transformation to Philips' code base and get several positive results.

6.1 Contributions of this Thesis

The main contributions of this thesis are the answers to the research questions:

RQ1 What APIs are provided in Philips' home-made OSAL?
To answer this research question, we manually checked the header file of Philips' OSAL library. Each function declaration in the header file contains a prefix in the function name. The prefix indicates the functionality of that function, from the concrete usage perspective. The detailed list of the prefixes is in **Section 3.2**.

RQ2 What are the functionality of Philips' OSAL APIs?
We investigated the functionality by examining the implementation of each function prefix and categorizing the functions from a general Operating System point of view. From the prefix, we have found 19 different functionalities provided by the existing OSAL. And from a general Operating System perspective, these 19 different functionalities can be divided into 5 categories: data type management, real-time clock, network management, memory management and task/event management. Furthermore, another category named administrative is also included in the OSAL, which is used to manage the other functions provided by the OSAL. The detail list and description of the functionality is in **Section 3.2**.

RQ3 What is the quantity and the distribution of Philips' OSAL APIs?
We first counted the number of function declarations from the header file manually, there are in total 108 function declarations in the header file. In other words, there are 108 interfaces provided by this OSAL. Afterwards, we applied automated examination with the help of Rascal. The automated examination iterates over all C and C++ source code files

in Philips' code base and found all function calls use the interfaces provided by the existing OSAL. There are 7955 function calls use the interfaces provided by the existing OSAL, with task/event management used the most, followed by memory management. The detailed list of the quantity and distribution is in **Section 3.3** and **Section 3.4**.

RQ4 How are Philips' OSAL APIs being used?
We checked the use of the OSAL functions manually in the code base. We can conclude 3 ways of using the OSAL functions: atomic use, the function is used only in a single statement. Simple paired use, the function is used in pair, starts with a list of statements end ends with a list of statements, with other statements in between. In simple paired use case, the statements in between are not affected by the return value provided by the OSAL functions. Complex paired use, the function is used in pair, starts with a list of statements end ends with a list of statements, with other statements in between. The statements in between may retrieve values returned by the OSAL functions. The detailed description and examples of the 3 use cases are explained in **Section 3.5**.

RQ5 What are the key aspects to make a transformation of OSAL APIs?
We answered this research question by an intensive requirement analysis. The requirements are derived from the problem statement, the ultimate goal of this study, and the analysis of the existing code base. We have concluded 8 key aspects of a transformation that must be taken into consideration: **type of transformation**, **transformation detail**, **new inclusions**, **applicable type of files**, **start location of the transformation**, **related variables in the transformation**, **related preprocessor macros** and **file modification**. All the 8 key aspects are spread in 13 requirements. The detail of these requirements are described in **Section 4.3**.

RQ6 What can be done to make the the transformation as automated as possible?
The solution of this research question came from a proper model-based design. We designed a complete transformation tool with the input side to the users, the output side to the file system and a transformation engine in between. The approach we used is to reduce the number of input entries of the transformation. Eventually, we proposed a Domain-Specific Language (DSL) to handle the automated transformation. The DSL is considered as the only entry in the input side of the transformation tool provided to the user to write the key aspects of the transformation. The rest of the jobs are performed by the transformation engine designed in a model-based way. The detail of the design is explained in **Chapter 4**.

6.2 Suggestions to Philips

Apart from the answers to the research questions, one other important contribution of this study is a working prototype of the transformation tool. We suggest Philips to use this tool to **partially** solve Philips' problem of modernizing the legacy code. It is possible to apply a **proper defined** transformation to the whole code base of Philips. The proper defined transformation should be either an atomic transformation or a simple paired transformation. Unfortunately, at this moment due to the limitations the complex paired transformation is not supported yet and so that we cannot get rid of the legacy OSAL completely with this tool. Furthermore, the C++11 constructs cannot be applied to .c files. But with the help of the transformation tool a large amount of the legacy OSAL can be transformed into the new version and with the future extension the tool will support more cases.

Bibliography

[1] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA, 2000. ACM.

[2] David F. Carr. Web-enabling legacy data when resources are tight. *Internet World*, 1998.

[3] A. Cockburn. Using both incremental and iterative development. *Cross Talk. The Journal of Defense Software Engineering*, May 2008.

[4] B. Combemale, J. Kienzle, G. Mussbacher, O. Barais, E. Bousse, W. Cazzola, P. Collet, T. Degueule, R. Heinrich, J. Jézéquel, M. Leduc, T. Mayerhofer, S. Mosser, M. Schöttle, M. Strittmatter, and A. Wortmann. Concern-oriented language development (cold): Fostering reuse in language engineering. *Computer Languages, Systems and Structures*, 54:139 – 155, 2018.

[5] S. Comella-Dorda, K. Wallnau, Robert C. Seacord, and J. Robert. A survey of legacy system modernizationapproaches. Technical report, Carnegie-Mellon univ pittsburgh pa Software engineering inst, 2000.

[6] George T. Doran. ThereâĂŹsa smart way to write managementâĂŹs goals and objectives. *Management review*, 70(11):35–36, 1981.

[7] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of c preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, Dec 2002.

[8] M. Feathers. *Working Effectively with Legacy Code.* Prentice Hall, 1 edition, 2004.

[9] M. Flower, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1 edition, 1999.

[10] Michelle R. Hribar, M. Frumkin, H. Jin, A. Waheed, J. Yan, and S. Saini. Legacy code modernization. Technical report, NASA, Jan 1998.

[11] P. Klint, T.v.d. Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177, September 2009.

[12] A. G. Koru, D. Zhang, K. E. Emam, and H. Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering*, 35(2):293–304, March 2009.

[13] A. van Lamsweerde. *Requirements Engineering.* Wiley, 1 edition, 2009.

[14] C. Larman and V. R. Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, June 2003.

[15] J. Liebig, C. Kästner, and S. Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 191–202, New York, NY, USA, 2011. ACM.

[16] H. Liu, Y. Gao, and Z. Niu. An initial study on refactoring tactics. In *2012 IEEE 36th Annual Computer Software and Applications Conference*, pages 213–218, July 2012.

[17] T. Mens, A. Serebrenik, and A. Cleve, editors. *Evolving Software Systems*. Springer-Verlag Berlin Heidelberg, 1 edition, 2014.

[18] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, Jan 2012.

[19] R. S. Oliver, I. Shcherbakov, and G. Fohler. An operating system abstraction layer for portable applications in wireless sensor networks. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 742–748, New York, NY, USA, 2010. ACM.

[20] M. Palmieri, I. Singh, and I. Cicchetti. Comparison of cross-platform mobile development tools. In *2012 16th International Conference on Intelligence in Next Generation Networks*, pages 179–186, Oct 2012.

[21] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. Wiley Publishing, 1st edition, 1996.

[22] Winston W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, pages 328–338. IEEE Computer Society Press, 1987.

[23] A. Schoofs, M. Aoun, P. van der Stok, J. Catalano, R. S. Oliver, and G. Fohler. A framework for time-controlled and portable wsn applications. In N. Komninos, editor, *Sensor Applications, Experimentation, and Logistics*, pages 126–144, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[24] M.T.M. Schuts. *Industrial Experiences in Applying Domain Specific Languages for System Evolution*. PhD thesis, Radboud University Nijmegen, September 2017.

[25] W. A. Shewart. *Statistical Method from the Viewpoint of Quality Control*. Dover Publications Inc., 1939.

[26] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts Essentials*. John Wiley & Sons Ltd., 1 edition, 2010.

[27] S. A. Slaughter, D. E. Harter, and M. S. Krishnan. Evaluating the cost of software quality. *Commun. ACM*, 41(8):67–73, August 1998.

[28] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring uml models. In Martin Gogolla and Cris Kobryn, editors, *âĽłUMLâĽń 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 134–148, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[29] T. Tourwé and T. Mens. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30:126–139, 01 2004.

[30] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment: A component-based language development environment. In Reinhard Wilhelm, editor, *Compiler Construction*, pages 365–370, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[31] N. Weiderman, K. Bergey, D. Smith, and S. Tilley. Approaches to legacy system evolution. Technical report, Carnegie-Mellon univ pittsburgh pa Software engineering inst, Dec 1997.

[32] N. Weiderman, L. Northrop, D. Smith, S. Tilley, and K. Wallnau. Implications of distributed object technology for reengineering. Technical report, Carnegie-Mellon univ pittsburgh pa Software engineering inst, 1997.

[33] S. Yoo and Ahmed A. Jerraya. *Introduction to Hardware Abstraction Layers for SoC*, pages 179–186. Springer US, Boston, MA, 2003.

Appendix A

Rascal Script Used for Searching Functions

This appendix contains all the detailed information about the Rascal script we used for function call analysis described in **Chapter 3**. The aims of using the automated Rascal script are:

- Automated iterating over all files in Philips' code archive.
- Getting ClaiR representatives (ClaiR AST) per file.
- Matching desired function calls.
- Calculating the total number of function call occurrences.

The code fragment in **Listing A.1** was used to reach our goal.

The basic task in this code fragment is searching and outputting. The first part of this method is to create a log file in order to store all function call occurrences. The file name is specifically determined by the only parameter of this method, also indicates the **prefix** of the function call to be searched. The second part of this method simply iterates over all the file in the list `files`, which stores all the file locations with `.c` and `.cpp` extension in the positioning software. The positioning software is in the `GeoCVEmb` folder as listed in **Line 2**.

In the iterations, the method first makes query via ClaiR to get a ClaiR AST for further queries. The AST has a datatype named `Declaration`, shown in **Line 17**. After getting a `Declaration`, a pattern matching is applied as shown in **Line 19** to get the exact function calls according to the input prefix. The matched function calls are stored in a list.

The final step is outputting. All the matched function calls are output to a log file, indicating the name of the function call and its exact location.

Listing A.1: A Rascal code fragment for searching existing functions.

```
1  /* Root location */
2  public loc l = |file:///Y:/PosCore/GeoCVEmb|;
3
4  /* Count method */
5  public void count(str functionCallPattern) {
6      map[loc, str] listOfFunctions = ();
7      list[loc] files = getLoc(l);
8      loc logFile = |project://DataExtractor/analysisresult/<
          functionCallPattern>.txt|;
9      if (! exists(logFile)) {
10         writeFile(logFile);
11     }
12
13     for (f <- files) {
14         println("File <f>");
15         for (m <- additionMacros) {
16             println("Macro <m> \r\n");
17             Declaration a = parseCpp(f,
18             additionalMacros = m);
19             list[Expression] funcs = getFunctionCall(a,
                  functionCallPattern);
20             for (e <- funcs) {
21                 if (! (e.src in listOfFunctions)) {
22                     listOfFunctions += (e.src : e.functionName.
                          name.\value);
23                     str s = "Function call <e.functionName.name
                          .\value> at <e.src> \r\n";
24                     appendToFile(logFile, s);
25                 }
26             }
27         }
28     }
29 }
```

Appendix B

User Manual of the Transformation Tool

This appendix gives a detailed guide on how to use the DSL designed in this thesis to apply a code transformation. The manual contains all necessary information, including installation and configurations. The manual will be written in a hands-on format, describe a transformation in a step-by-step approach.

B.1 Installation

This transformation tool is a Rascal project and Rascal is running on Eclipse. Therefore, a installation of Rascal and Eclipse is necessary. In order to install Eclipse, a proper version of Java Development Kit (JDK) is also needed. The installation flow starts with installing a JDK.

Installation of JDK

The proper version of JDK used in this study is JDK version 8. Go to Oracle's download page of JDK[1] to get a copy of JDK for the development machine.

Installation of Eclipse

The Eclipse version we used for implementing this tool is **Eclipse Oxygen.2**. However, the later version of Eclipse also works with this tool. The required packages of Eclipse are the Java development package and C/C++ development package. We recommend to install Java version first and then install C/C++ package.

Go to *Eclipse Packages* website `https://www.eclipse.org/downloads/packages/` and find the **Eclipse IDE for Java Developers** as shown in **Figure B.1**.

Make sure the installed version is for Java developers, otherwise there might be some unexpected issues in the further installation. It is also possible to use Eclipse package manager for the installation, but we do recommend to install the package directly with the installation executable.

After installing *Eclipse for Java Developers*, it is also required to install the C/C++ supporting package, which is named **Eclipse C/C++ Development Tooling (CDT)**. The installation of CDT is **NOT** from the website but from the internal software installation system.

[1] `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`

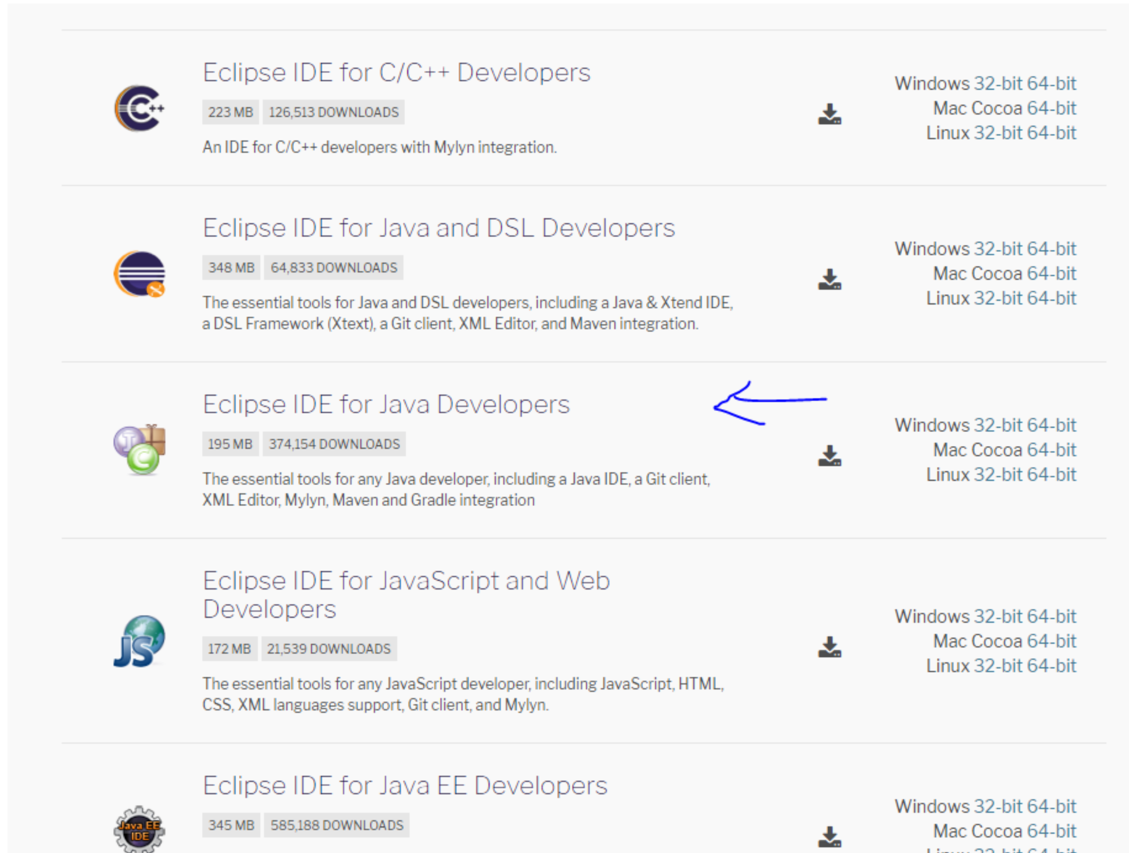

Figure B.1: Get Eclipse for Java Developers.

Open Eclipse just installed, go to **Help** − > **Install New Software** and in the **Work with** field click **Add**. Then in the pop-up window enter **CDT** in **Name** and `http://download.eclipse.org/tools/cdt/releases/9.2` in **Location**. **Figure B.2** is the correct information to be installed. Make sure the location ends with **9.2**, which is version 9.2 of CDT. A later version may not be compatible with this tool. Then select all options and click **Next** till finish.

Installation of Rascal

When the installation of Eclipse is finished, use the same steps to install Rascal. Go to **Help** − > **Install New Software** and add a new repository. In the **Name** filed enter Rascal and in the **Location** field enter `https://update.rascal-mpl.org/unstable`. Make sure the selected repository is unstable version as not all the features used in this transformation tool is merged to the stable version. Then select Rascal and finish the installation. After restarting Eclipse, select the perspective to Rascal.

Installation of ClaiR

The official installation of ClaiR is also via Eclipse software installation system. However for the transformation tool designed in this study, we need a modified version for it. The modification version is on Github, there are 2 places to get it: through the original repository at `https://github.com/cwi-swat/clair` or through a forked repository at `https://github.com/t-liu93/clair`.

It is recommend to get the modified ClaiR from the forked repository as it is tested and proved

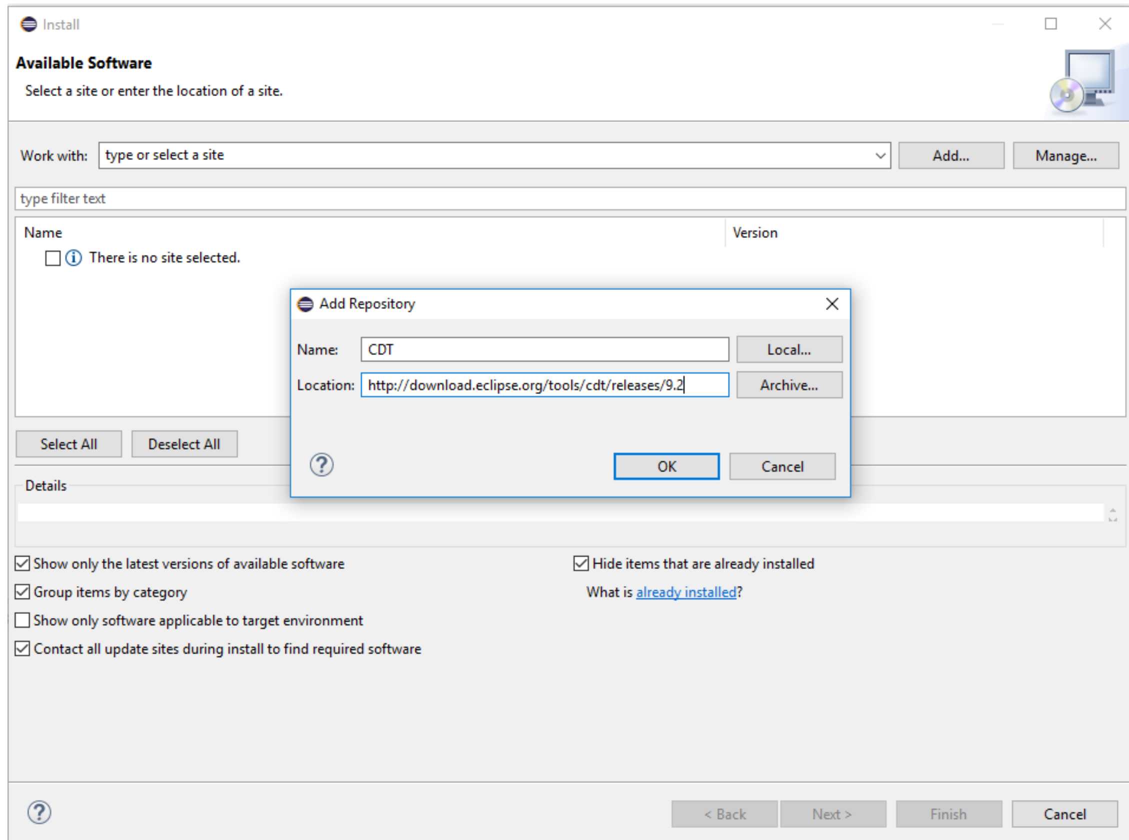

Figure B.2: A screen of installing Eclipse CDT.

working with this transformation tool. For the forked repository, go to the release page of the repository at `https://github.com/t-liu93/clair/releases` and download the release with tag **DemoVersion**, then import the whole project into Eclipse workspace.

After importing the project, click on any of the file or directory in this project, and press **Press to Start a Rascal console** button in the workbench toolbar. A new Rascal console should be created in the terminal area of Rascal, with the name `clair` and model `debug`. Then input
`import lang::cpp::AST;`
in the console and an `ok` is supposed to be printed in the console. After that type:
`import lang::cpp::ASTgen;`
`generate();`
in the console statement by statement. If it shows ok then the installation of ClaiR is finished. **Figure B.3** shows a successful installation in Rascal console.

Installation of the transformation tool

The transformation tool can be found from Github as well. On the release page of the repository `https://github.com/t-liu93/Code-Modernization-Tool/releases`[2], find and download the latest zip file.

When the whole project is downloaded, unzip it and simply input it into Eclipse workspace by clicking **File** − > **Open Projects from File System** and then select the unzipped directory

[2]Currently not available and please contact Mathijs Schuts at Philips `Mathijs.Schuts@philips.com` or the author of this thesis for the project file.

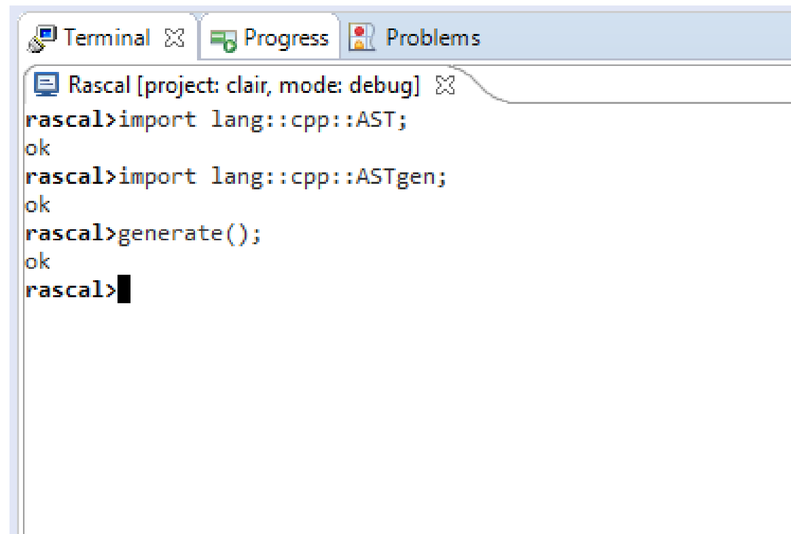

Figure B.3: A screenshot of a new Rascal console of ClaiR.

then click Finish.

Then **right-click** on the newly imported project and select **Properties**, navigate to **Java Build Path** then **Projects**, click **Add** and then check project **clair** imported in the previous step and click **OK** then **Apply and Close**. **Figure B.4** and **Figure B.5** indicates the correct build path setting.

After setting the build path, click on any file or directory in project `Code-Modernization-Tool` and open a new Rascal console on it. In the newly opened Rascal console type:
`import Transformer;`
If an ok is printed in the console as shown in **Figure B.6**, then the installation of the transformation tool is finished.

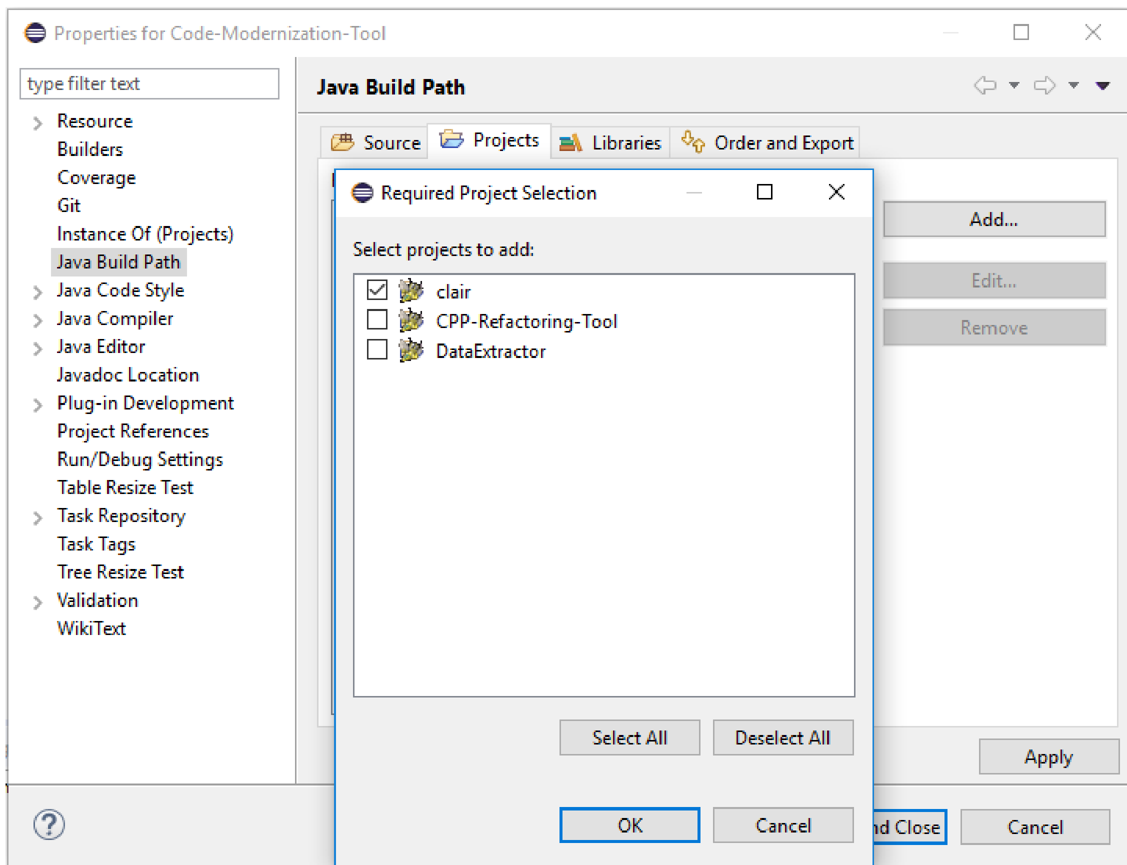

Figure B.4: Check ClaiR as reference project.

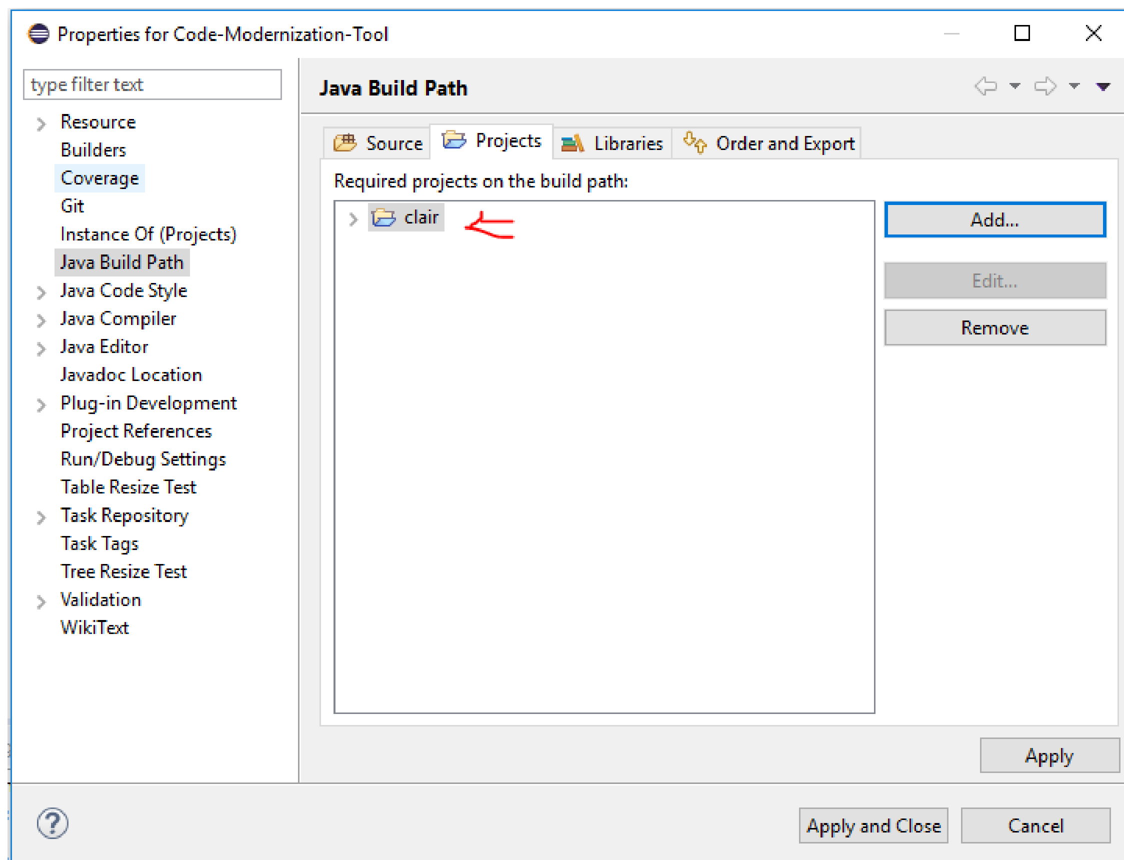

Figure B.5: The Java build path.

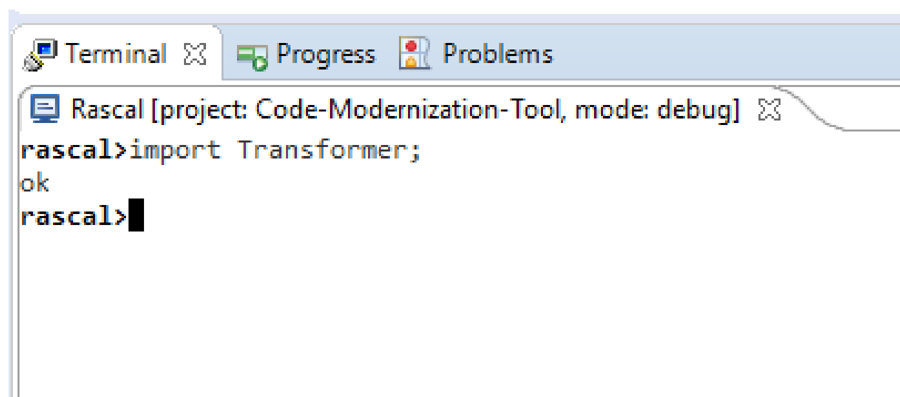

Figure B.6: Console example of a successful installation of transformation tool.

B.2 Write the DSL

The structure of a DSL script contains 4 parts: **start symbol**, **start location**, **transformations** and **end symbol**, as shown in **Figure B.7**.

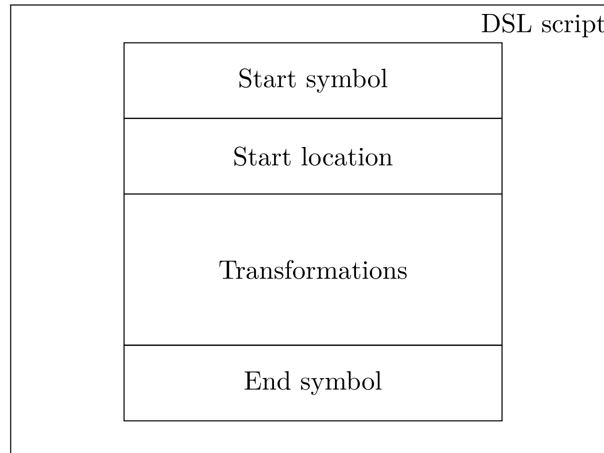

Figure B.7: The structure of a DSL script.

The **start symbol** is a **case sensitive keyword** denoted as `Start patterns`, followed by a **connector** denoted as `:=`.

The **start location** contains 3 elements sequentially: a **case sensitive keyword** `Start with`, a **connector** `:=` and a Uniform Resource Identifier (URL) that indicates the start location. A start location **must** ends with a semi column `;`.

The **transformations** contains several elements that describe the exact transformation details. These details is described in the later part of this section.

The **end symbol** is a **case sensitive keyword** denoted as **End.**. Note that this keyword ends with a **dot** `.`.

Details about transformations

The **transformations** block in **Figure B.7** contains the detail of multiple transformations. The detail of each transformation is independent from the others. It consists of 4 compulsory elements and 1 optional element. The compulsory elements including **transformation start symbol**, **transformation rule**, **applicable file extensions** and **transformation end symbol**, while the optional element is **additional header files**, as shown in **Figure B.8**.

The **transformation start symbol** consists of 3 tokens separated by white spaces. The tokens are input sequentially include a **case sensitive keyword** `Pattern`, a **transformation id** starts with a **hash symbol #** and a connector `:=`.

The **transformation rule** contains a **source** and a **target**. Both of them are covered in a pair of **square brackets** `[]`. The **source** and **target** are connected with a **transform to** symbol denoted as `-->`. Hence a transformation rule is described in form:
`[srouce] --> [target];`. It must ends with a semicolon.
For an atomic transformation, both **source** and **target** are written in C/C++ code format.

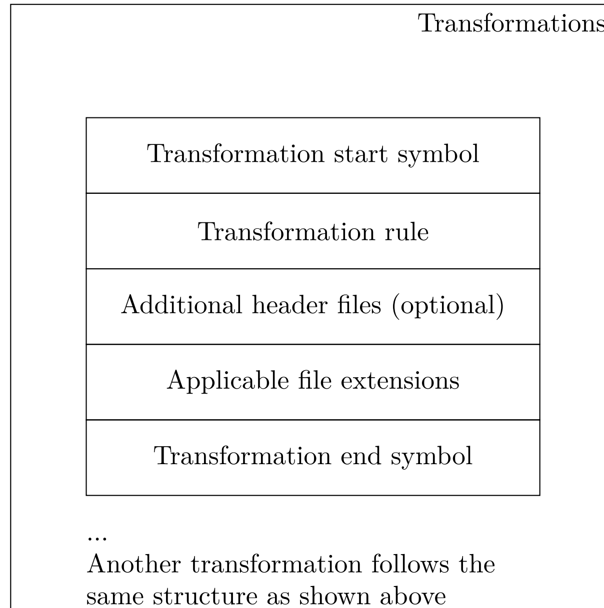

Figure B.8: The structure of transformations.

On the other hand, for a simple paired transformation, the `source` and `target` contains a **pair separator** `...` to distinguish an open pattern and a close pattern.

For each transformation, the transformation rule is followed by additional headers and applicable file extensions. Additional headers start with **`Add headers :=`** and applicable file starts with **`Applies to :=`**. Additional header statement is optional in this DSL.

B.3 Apply a Transformation

Applying a transformation contains 2 steps: writing a DSL script and applying the script in Rascal console.

The DSL script can be written in any text editors. Follow the manual described in **Appendix B.2**. When finish writing, save the DSL script in any place on local machine, with file extension `.txt`.

Select any file of the project **Code-Modernization-Tool** in Eclipse, and then open a new Rascal console with this project. In the input area of Rascal console, type the following commands:

```
l = |<locOfTheScript>|;
import Transformer;
transform(l);
```

Then the transformation tool will apply the transformation automatically according to the details defined in the DSL script. Note the `<locOfTheScript>` is the location of the DSL script defined in the previous paragraph.

Appendix C

Implementation Examples in Rascal

This appendix contains some code fragments we have implemented in this study. These implementations are about matching statements and generating code. The implementation is done with Rascal.

C.1 Exemplar Implementation to Match a Single Statement

In our implementation, we use pattern matching system provided by Rascal to match a statement. This implementation uses overloading feature provided by Rascal, so that different types of statements can be matched with one method name. The method used to match a single statement has two inputs: the first one is the structure of the statement that is going to be matched and the other is the AST of a source code file potentially containing the statement. All matched statements are stored in a list, which is the return value of this method.

In the example shown in **Listing C.1**, we give two methods to match a *function call* within a *expression statement*. The first method matches a *expression statement*, while the second one matches a *function call* inside that expression statement. We set one additional constraint in the pattern matching statement, which ensures the matched function call has the desired name. In our study, two functions are identical in structure if they have the same name.

Furthermore, we define a `default` method of this particular `matchStatement`. This default method accepts two arbitrary `value` variables, which is a generic data type in Rascal, as input, and then throws an exception to indicate the user that the input value is not supported. According to the design of Rascal, this default method will be invoked **if and only if** all other variants of this method have been tried, and all previous attempts have failed. In practices, if this exception has been thrown, the user only needs to add a new variant of this method.

Listing C.1: The implementation of matching a single statement.

```
1  //Match an expressionStatement
2  public list[Statement] matchStatement(
3      expressionStatement(_, e),
4      Declaration a) {
5
6      return matchStatement(e, a);
7  }
8
9  //Match a functionCall, assume it is always under an
       expressionStatement
10 public list[Statement] matchStatement(
11     functionCall(n, arg),
12     Declaration a) {
13
14     return [expressionStatement([], f, src=f.src) | /f:
           functionCall(idExpression(name(fn)), _) := a,
15          fn==n.name.\value];
16 }
17
18 //Default with an exception
19 default list[Statement] matchStatement(value v, value a) {
20     throw "Unknown statement when matching statement <v> in AST
           <a>. ";
21 }
```

C.2 Exemplar Implementation to Match a Pair of Statements

To match a pair of statements, we use the pattern matching system in Rascal as well, together with the overloading feature. Hence, all methods to match a pair of statement have an identical name. A pair matching statement has three inputs: the structure of the first statement in a pair (first statement of open pattern), the structure of the last statement in a pair (last statement of close pattern) and the AST of the file to be checked. The return value of this method is a **list of *list of statements***. Each inner list (nested list) is the matched statements that contain the pair. The outer list is used to store several matched list if there are several pairs exist in code.

In the example shown in **Listing C.2**, we match a pair with two structures: starting with a simple declaration and ending with a function call. Similar to single statement, we also use additional constraints to make sure the matched statements are the desired ones. The default method is also used here.

Listing C.2: The implementation of matching a pair of statements.

```
1  //Variant 1: declaration statement ... functionCall
2  public list[list[Statement]] match(
3      Declaration ast,
4      declarationStatement(_, decl),
5      expressionStatement(_, exp)) {
6
7      return [s | /s: [
8          *_,
9          declarationStatement(
10         _,
11         simpleDeclaration(
12             _,
13             namedTypeSpecifier(
14                 _,
15                 name(dn)),
16                 _)),
17         *after,
18         expressionStatement(
19             _,
20             functionCall(
21             idExpression(name(fn)),
22             _)),
23         *_] := ast,
24             dn == decl.declSpecifier.name.\value &&
25             fn == exp.functionName.name.\value];
26 }
27
28 default list[list[Statement]] match(
29     value ast,
30     value begin,
31     value end) {
32     throw "Unsupported statement pairs <begin> and <end>";
33 }
```