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

An Automated Approach to Check Software Architecture Erosion

Master's Thesis

Ruichen Hu

Supervisors:
Prof. Dr. Michel Chaudron
Prof. Dr. Jurgen Vinju
Dr. Mathijs Schuts

Eindhoven, August 2023

Abstract

Architecture erosion (AE) arises when the implementation diverges from the prescribed architecture rules. It increases software complexity and undermines code maintainability. This thesis presents a method to automate the process of detecting AE in Philips positioning software. To achieve this, a domain-specific language (DSL) tool has been developed to perform conformance checking between the software's implementation and its intended architecture. This tool is called READ (**R**uichen **e**rosion of software **a**rchitecture **d**etection), consisting of a CMAKE parser, a model generator, a DSL parser and a rule checker implemented with the meta-programming language RASCAL.

The tool is specific to positioning software in that it provides tailored syntax to define the architectural rules specified for the software. The language's expressive nature ensures adaptability to accommodate future architectural rules. To evaluate the usability of the tool, multiple tests employing different methodologies were conducted. These tests prompted iterative improvements aiming at enhancing result accuracy and reducing response time. The conclusive evaluation proves that the tool provides precise results with respect to disallowed architecture layer dependency and invalid interface usage. Additionally, it demonstrates usability in identifying disallowed dynamic memory allocation in the implementation of the positioning software.

Acknowledgement

This thesis reflects my master's graduation project undertaken with the cooperation between the computer science department at Eindhoven University of Technology and the software group at Philips. I would like to express my sincere gratitude to all the people who contributed to the successful completion of my thesis.

First of all, I would like to thank my first supervisor, Professor Michel Chaudron, for introducing me to an interesting topic in software architecture analysis with meta-programming and holding weekly progress meetings to share ideas. Secondly, I am deeply thankful to my external supervisor, Dr. Mathijs Schuts, for providing me with an opportunity to research software at Philips and guiding me with practical ideas in tool design and implementation. Furthermore, I express my appreciation to Professor Jurgen Vinju for continuous support and guidance in Rascal programming and tool evaluation. Collaborating with him on pair programming and debugging has been an enriching experience.

Lastly, I acknowledge the support of my parents and friends, without whom my two-year study would not have been successful. Their financial and emotional encouragement have been instrumental in my academic journey.

Contents

Contents vii

List of Figures ix

List of Tables xi

List of Listings xiii

List of Algorithms xv

1 Introduction 1
1.1 Motivation . . . 1
1.2 Problem Description . . . 2
1.3 Research Questions . . . 3
1.4 Thesis Outline . . . 3

2 Background and Concepts 5
2.1 Software architecture and Architecture Erosion . . . 5
2.2 Philips Positioning Software . . . 6
2.3 Static Analysis . . . 7
2.4 Metaprogramming and Domain-Specific Languages . . . 8
2.4.1 RASCAL . . . 8
2.4.2 CLAIR . . . 9
2.4.3 TYPEPAL . . . 11

3 Related Work 13
3.1 Detection of Architecture Erosion . . . 13
3.2 Domain-Specific Languages for Architecture Analysis . . . 13
3.3 Code Analysis and Transformation with RASCAL . . . 15
3.4 Rule Formalization with DSL Syntax . . . 16
3.5 Conclusions . . . 16

4 Design and Implementation 17
4.1 Development Procedures . . . 17
4.2 Requirements . . . 18
4.3 Architectural Rules . . . 19
4.4 Workflow . . . 21
4.4.1 Model Generation . . . 23
4.4.2 Design of the Domain Specific Language . . . 28
4.4.3 Architecture Erosion Checking . . . 34
4.5 Deployment . . . 38
4.6 Conclusions . . . 39

5 Evaluation and Results **41**
5.1 Quality of the M3 Models . . . 42
5.1.1 Parsing of the CMake Files . . . 42
5.1.2 Evaluation of Include Path Extraction . . . 43
5.2 Implementation of Architectural Rules . . . 48
5.2.1 Disallowed Layer Dependency . . . 48
5.2.2 Non-public Interface Usage . . . 52
5.2.3 Dynamic Memory Allocation . . . 54
5.3 Response Time . . . 64
5.4 Anecdotal Evidence . . . 66
5.5 Conclusions . . . 67

6 Conclusions and Future Work **69**
6.1 Conclusions . . . 69
6.2 Future Work . . . 70

Bibliography **73**

Appendix **75**

A Description of M3 Model Fields **77**

B List of Architectural Rules **81**

C Structure of DSL Syntax **83**

List of Figures

1.1 Philips Azurion System 2

2.1 Architecture of the positioning software (adapted from [1]) 7
2.2 Comparision between CST and AST for the same expression 11

3.1 Data-flow diagram of EASY paradigm for the design of DSL tooling with RASCAL (adapted from [23]) 15
3.2 A common pattern for architectural rule language design (adapted from [34]) 16

4.1 A UML activity diagram illustrating steps to develop the DSL tool 18
4.2 Visualization of the desired layer dependency 20
4.3 Data-flow diagram of the DSL tool 21
4.4 A concrete example of architectural rule checking 22
4.5 UML package diagram of READ 24
4.6 A illustration of part CMAKE tree in Philips' codebase 25
4.7 The simplified flowchart for CMAKE parsing process constructed in accordance with ISO 5807 [13] 26
4.8 Syntax structure of the three blocks 30
4.9 Definitions of depend- and use-statement in DSL syntax 31
4.10 Definitions of template block in DSL syntax 33
4.11 Folder structure of the code base 36
4.12 Execution steps of the tool on Jenkins 38

5.1 Number of not found headers of each folder of source files categorized by four causing factors 46
5.2 Number of not found headers of each folder after the improvements categorized by four causing factors 48
5.3 Violations of disallowed dependency between layers or modules 52
5.4 Violations of non-public interface usage between layers or modules 53
5.5 Workflow of checking dynamic memory allocation 54
5.6 An example of the violation results for RULE 3 (Left: results in RASCAL; Right: source code in C/C++) 55
5.7 Design of the top-down mutation test 58
5.8 Workflow of the bottom-up test 62

List of Tables

2.1 A synopsis for the core part of positioning software . . . 6
2.2 Relations of the M3 model in CLAIR . . . 9

3.1 Description of six state-of-the-art DSLs . . . 14

4.1 Dependency relationships between the layers . . . 19

5.1 Performace of the CMAKE parser during software evolution . . . 43
5.2 The worst case results of the identified violations after the removal of an include path . 44
5.3 Comparison of source files with not found headers to the total number of source files categorized by folder . . . 46
5.4 Number of layer dependency violations in each layer or module before and after improvements . . . 49
5.6 Confusion matrix for the first implementation of RULE 1 . . . 51
5.8 Number of non-public interface violations in each layer or module after improvements . 53
5.10 Confusion matrix for the first implementation of RULE 2 . . . 53
5.12 Number of violations of dynamic memory allocation collected in each implementation . 55
5.14 Confusion matrix for the first implementation of RULE 3 . . . 57
5.16 Results of the top-down mutation testing . . . 59
5.18 Number of distinct problems before and after the improvement of the list of standard libraries . . . 60
5.20 Results of the bottom-up reachability test . . . 62
5.22 Comparison of the time required to generate the models for 100 sampled source files with different sources of include paths . . . 65
5.23 Comparison of the time required to compose the models between partial and complete model fields . . . 66
5.24 Comparison of the time required to check RULE 3 with and without the AST cache . . . 66

List of Listings

2.4.1 A RASCAL snippet to define the abstract syntax [22] 8
4.4.1 An example of CMakeLists.txt in the codebase 25
4.4.2 An example of READ instance 29
4.4.3 Syntax of template invocation, use- and depend-statement 32
4.4.4 Implementation of no-use-before-define rule with TYPEPAL 34
5.2.1 DSL instance implementing the checking on function `sleep` 64

List of Algorithms

1 M3 model generation 27
2 Checking of the disallowed layer dependencies 35
3 Checking of the uses of non-public interface 36
4 Checking of the dynamic memory allocation in the execution phase of the application . . 38
5 Model generation with the improved list of standard libraries 60

Chapter 1

Introduction

1.1 Motivation

Software is a system that provides functionalities to fulfill the requirements of its stakeholders [6]. Instead of directly inspecting the source code, people can gain insights into the software system by referring to its architecture. Software architecture manifests the organization of a system, as well as the relationships among the system components and the environment in which the system operates [17]. It reveals an abstract view of how the requirements are mapped to the components of the resulting software [6].

Architecture design requires the determination of design decisions, which can be documented as architecture constraints or architectural rules. These rules guide the developers to develop and maintain the software. Next to the source code, explicit architectural guidelines provide an independent and comprehensible overview of the existing code and the constraints on future adaptations of the code. It is vital to ensure the software implementation conforms to a well-defined architecture, as this conformance contributes to high software quality and reliable software performance. However, in practice, the implemented software often deviates from the designed architecture because these rules are violated during development. In case of our assignment, multiple architectural rules are designed for the software under investigation. Nevertheless, these rules are not checked by a compiler in the integrated development environment (IDE), nor by a continuous integration toolkit. As a result, it is possible that violations are added during software development and maintainence without being noticed. These violations increase the maintenance difficulty and reduce the software's coherence and clarity, further aggravating the architecture problems [32]. Ultimately, the system may fail to provide the expected functionalities. This phenomenon is commonly referred to as architecture erosion (AE). It is a metaphor describing software architecture degradation due to the increased divergence between the architecture design and code implementation during software evolution. The problematic parts, i.e., the artifacts violating architectural rules, erode over time and deviate from the original design decisions.

Recognizing that AE can eventually make maintenance and evolution of the software prohibitively dangerous and/or expensive and hinder the implementation of the new requirements, it is essential to take measures to manage the erosion. Erosion management consists of two aspects: detection and elimination. Detection involves identifying the source code which violates architectural rules, while elimination refers to removing and reimplementing the detected problematic code. In the context of this assignment, our focus is on detecting AE, with the aim of minimizing erosions in maintained source code.

This thesis contributes by an early detection mechanism, which can be executed by the CI periodicity to identify the newly introduced violations since the last checking and deliver the results to Philips' engineers.

1.2 Problem Description

This master's assignment is associated with the software group of the mechatronic department at Philips, a technology company specializing in healthcare. Philips provides integrated solutions for diagnostic imaging. As a significant part of these solutions, the Image Guided Therapy (IGT) system is developed to facilitate image-guided interventions, allowing disease diagnosis and treatment to be performed with minimized invasion [27]. Figure 1.1 illustrates a product line of the IGT system known as Azurion. Azurion serves to automatically align the X-ray beam with the patient and generate images for treatment. The mechatronic department is responsible for developing the positioning software for Azurion, which regulates the functioning of all the movable components within the system. This software serves as the codebase for our assignment.

Figure 1.1: Philips Azurion System

With respect to the software design, it is organized in a multi-layered architecture with several modules providing utility and test functionality. Based on this design decision, multiple architectural rules have been proposed to regulate the behaviors of the source code components within and between the layers. After decades of development by about 50 software engineers, the software has grown to a considerable scale with more than one million lines of code. As new code segments have been incrementally implemented along with legacy code, managing dependencies between different components has become increasingly challenging. Specifically, the prescribed architectural rules are violated due to the disallowed inter-layer dependencies and the skipping of the intermediate layer. This erosion of the architecture undermines software quality and hampers the practicability of software maintenance. Consequently, software architects are seeking a solution to detect the code snippets where AE happens and guide the developers in implementing functionalities to adhere to the rules. This expectation formulates the primary functional requirement of a tool, which is capable of accurately identifying rule violations in the source code. Additionally, the tool should support the definition of architectural rules such that it can be adapted to different checking requirements.

In summary, this assignment aims to design and implement a tool that automatically checks the conformance between the architectural rules and implementations for Philips' positioning software and identifies the architectural violations in an accurate and efficient manner.

1.3 Research Questions

With respect to the theoretical and practical parts of the study, we propose four research questions (RQs). By answering these questions, we can know how to tackle AE in an industry-scale software system with meta-programming technologies. Following are the definition and the intuition of RQs:

- **RQ1: Which method is suitable to check AE in Philips' codebase?**
  The detection of AE has been intensively investigated. We need to select a method specific to Philips' codebase, as some architectural rules could be present specifically in this codebase. This requires a literature study on the related work to filter the feasible solution.
- **RQ2: How are the architectural rules formalized in the tool?**
  The architectural rules are described in natural language. To encode these rules into code, a mechanism is necessary to represent the elements in rules with code artifacts while maintaining the semantics accurately.
- **RQ3: How is architecture erosion detected by the tool?**
  The aim of the tool is to detect AE with respect to the given architectural rule. Thus, the design of a clear flow from the input source file to the desired output is required. This question is closely related to the design and implementation of the tool.
- **RQ4: How is the quality of the developed tool evaluated?**
  For each rule that is checked by the tool, it is expected to collect a list of violation facts. It is necessary to test the correctness of the detected violations such that the usability of the tool can be guaranteed. Therefore, methods to evaluate the tool's performance need to be proposed.

These research questions are investigated in the following sections: Section 3.1 compares different approaches for AE detection. Section 4.4.2 describes how architectural rules are implemented. Section 4.4 provides a step-by-step description of how a rule is checked. In Chapter 5, different measures are applied to evaluate the performance of the tool. The answers to all the research questions are summarized in Chapter 6.

1.4 Thesis Outline

The content of the thesis is structured as follows:

- **Chapter 2, Background and Concepts**
  This chapter introduces the basic concepts related to AE in the content of the assignment. Besides, it explains the architecture of Philips' codebase and the technology used for developing the tool.
- **Chapter 3, Related Work**
  This chapter contains a literature study on the state-of-the-art methods for AE detection and the recent usage of RASCAL for code analysis. This chapter explains the reasoning behind the method selection of the tool.
- **Chapter 4, Design and Implementation**
  This chapter interprets the workflow of the tool and the checking procedures for each architectural rule.
- **Chapter 5, Evaluation and Results**
  This chapter evaluates the quality of the tool from the aspect of usability and discusses the causes of inaccuracy. Subsequently, it shows how the implementation is improved iteratively and derivates the final results.
- **Chapter 6, Conclusions and Future Work**
  This chapter presents the contribution of the thesis with answers for RQs. In the end, it gives some prospects for future work.

Chapter 2

Background and Concepts

This chapter introduces the background of the assignment in the context of Philips' codebase and interprets the concepts related to AE as well as the theoretical foundation of the technologies used to develop the tool.

2.1 Software architecture and Architecture Erosion

Software architecture delineates the interactions between software components and their interface with the external environment in which the software operates [17]. During the design stage of the software, the architectural rules are proposed to specify the desired software architecture, which serves as a blueprint for software implementation. Accordingly, the software is expected to be developed and maintained in adherence to the desired architecture. Nonetheless, as the software evolves, the actual implementation may violate the prescribed architectural rules, leading to a degradation in the maintainability and quality of the software.

The cause of such software degradation can be attributed to two factors [36]. One is architectural drift, which happens when an architectural element not belonging to the planned architecture is introduced into the software. As the introduced element is not documented, it may be used in developing other modules unintentionally, leading to unexpected behaviors of the architectural components [4].

Another factor is architecture erosion, which is the research object in this assignment. The earliest usage of the term architecture erosion can be traced back to [32]. The authors interpret architecture erosion as a violation of architecture. Moreover, in [5], the definition of AE is expanded to introduce the architecture decision that violates the prescriptive architectural rules. Some other researchers define the AE phenomenon from another aspect rather than architectural violation: AE is the deterioration of the software quality during software evolution [10]. The quality metrics of a software system reflect the erosion in this definition.

In addition to architecture erosion, other terms are proposed to denote a similar phenomenon. In [37], design erosion represents the eroded software designs that lead to the redesign and rebuild of a new architecture. Some researchers use architectural decay to describe the architectural degeneration caused by uncontrolled architectural changes [33]. Although the definitions vary slightly compared to architecture erosion, the latest systematic mapping study reveals that AE, architecture deterioration and architecture decay correspond to the same deterioration phenomenon in software architecture [25]. Besides, we find that design erosion and architecture erosion are interchangeable based on the problems they present.

In the context of this thesis, architecture erosion refers to the violation of the intended architecture caused by the complex components of the implemented architecture. This definition has the same meaning as what is defined in [25].

2.2 Philips Positioning Software

As mentioned in Section 1.2, the codebase under investigation is the positioning software of the Azurion system. This software coordinates and controls the movements of Azurion's system components. One responsibility of the software is to adjust the position of the C-arm such that the radioactive beams precisely align with the patient's anatomy. Additionally, the software provides the interface to end users to control the positions of C-arm and the patient table with joysticks [1].

The software is implemented in C and C++. A brief synopsis for the number of source files is demonstrated in Table 2.1. This table only summarizes the core part of the codebase, excluding the dependencies. With respect to the design, the positioning software follows a layered architecture with five layers and two modules. Following are the descriptions of all the layers and modules [1]:

Table 2.1: A synopsis for the core part of positioning software

	C/C++ Header	C Source	C++ Source	CMake
#(Files)	5383	32	5263	599

- Interface layer (`A`): It is an interface between the clinical users and the internal system of Azurion. It provides access to positioning functionalities to clinical users and translates the users' requests into a sequence of application movements.
- Layer (`B`): Business logic.
- Layer (`C`): Business logic.
- Hardware layer (`D`): This layer translates the movements of axes to the movements across the three-dimensional coordinates of x, y and z. It also provides an interface to communicate and control the hardware
- Utility layer (`E`): This layer offers utilities for tasks including memory management, interprocess communication, etc., and wrappers for operating system primitives.
- Field Service module (`F`): This module contains all field service tools to assist the tasks like position adjustment.
- Test module (`G`): This is the test module for the implementations in all the other layers and modules.

Figure 2.1 demonstrates the grouping of the layers and modules of the architecture. It is clear that all the layers compose the main building block of the architecture. Here, `A`, `B`, `C` and `D` are presented with a hierarchy that the layer at the bottom directly controls the hardware while the layer at the top is exposed to the user interface. The exception is layer `E` that lays vertically from top to bottom because this layer is accessible for all the other layers. As for the other two modules, they are grouped separately.

Each layer or module has a prescribed restriction on which other layers or modules it can depend on. For example, layer `E` provides utilities for all the other layers and modules. Therefore, all the other architecture components can depend on `E`, while `E` is not allowed to request service from any other layers and modules. The restrictions are proposed by the architects based on the functionality of each layer and may change along with software evolution.

With regard to architecture implementation, each layer and module mentioned above is mapped to one or multiple folders in the codebase. The folders contain headers and source files written in C and C++. The architect defines the mapping with a YMAL file by specifying which directories belong to which layer or module. Additionally, the files generated during software building are placed in `Build` folder. A `Core` folder is constructed to store the source files whose layer has not been determined yet. The `Core` and `Build` folders are not part of the desired architecture. However, the subfolders within

them are used for the mapping between the architecture and the implementation. Therefore, the YAML file also contains the directories from these two folders. The files in `Core` folder will be migrated to the layer or module corresponding to their functionality in the future.

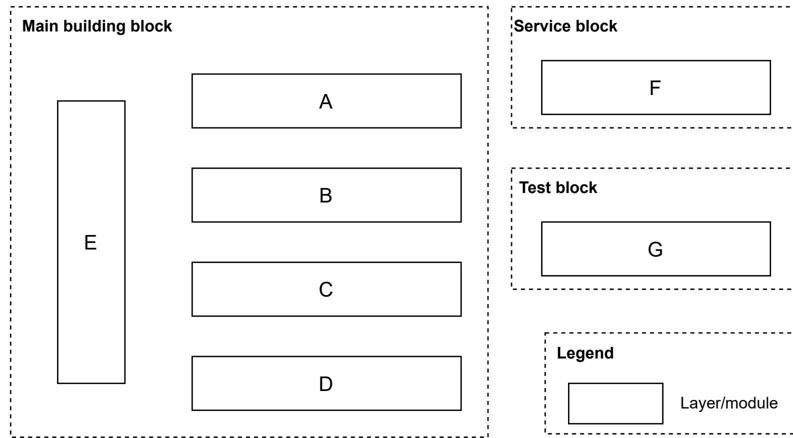


Figure 2.1: Architecture of the positioning software (adapted from [1])

2.3 Static Analysis

Static analysis is a method to analyze the behavior of a program without its actual execution [11]. It is a widely used approach to detect defects and predict possible software behaviors. The main advantage of static analysis is that it automates the processes of checking the problematic code segments in the program. Here, the problem denotes the implementations are syntactically correct but violate the prescribed design rules or contain unexpected runtime behaviors [11]. One example in the context of Philips' codebase is the use of specific disallowed functions and deadlocks. Compared to other manual ways to evaluate the implementations, like code review, static analysis is an automated way that covers the entire scope of the software. This process reduces the time to review the code and also considers corner cases that humans may neglect.

Depending on the concrete design and implementation of static analysis, the analysis results may contain problems. An accurate static analysis neither reports code falsely nor misses a true violation. A static analysis without false negatives typically has some false positives. This is called an over-appromation [29]. A false positive is the case regarded by the static analysis tool as incorrect implementation but is actually correct, while a false negative arises when the presence of a fault is not detected by static analysis. The validation of whether the collected results are real defects involves manual work, which increases developers' workload. On the other hand, a static analysis without false positives typically does not report several false negatives. That is an under-approximation. The common causes for the false negatives are the flaws in the design and the implementation of the tool. Therefore, to ensure a static analysis tool contributes to the development and maintenance of software, it is important to improve the precision of the tool and develop a scheme to evaluate its quality. The strategy we used is to implement the tool with as minimal false negatives as possible. Then, the implementation is improved to remove false positives. The reason for this strategy is we need first to ensure the usability of the tool in identifying the rule violations. As mentioned in Chapter 1, the tool must be usable to detect the violations. If the obtained results are all false negatives, the tool is not capable of capturing the divergence between the actual implementation and intended architecture at all. Therefore, we prioritized the minimization of false negatives in the implementation of the tool.

2.4 Metaprogramming and Domain-Specific Languages

Metaprogramming refers to the realization of programs that regulate the behaviors and functionalities of other programs [23]. In other words, it is a process of code transformation or generation that uses a language, called metaprogramming language, to modify or generate another language, called object language [26]. The input of a metaprogram can be a code snippet written in any programming language, a list of log information stored in plain text, or any other formats representing the data. The metaprogramming language provides a framework to use abstract and concrete syntax to handle input data, including matching the patterns presented in the input and applying the transformation within the desired scope. That means once the syntax is defined, all the patterns matched by the syntax can be extracted from the source input and further processed with the desired operations. Therefore, metaprogramming is an efficient way to inspect and maintain programs.

One important usage of metaprogramming is to design domain-specific languages (DSLs). A DSL is a language applicable to a specific domain. Compared to general-purpose languages, which are usually designed for a wide range of use scenarios, DSLs merely focus on one domain determined by the requirements and specified by the language designers. The designers tailor the syntax and grammar of the DSL such that the users can utilize them to implement functionalities that may be too cumbersome or impossible to implement with general-purpose languages. DSL is the main method we used to formulate the architectural rule of Philips' positioning software and couple it with a rule checker to identify architectural violations. A detailed description of the use of DSL in our tool is in Chapter 4.

2.4.1 RASCAL

RASCAL[1] is a metaprogramming language for source code analysis and transformation [22]. It provides a flexible mechanism for the users to design languages, conduct static analysis and generate code. It is a modular language like Java that enables the desired modules to be imported and extended. Like other high-level programming languages, it offers primitives to manipulate data structures like set, list and map. Besides, it supports functionalities, such as pattern matching, comprehension, switching and visiting, to facilitate the programming process [22].

One application scenario of RASCAL is to design and implement DSLs. RASCAL provides context-free grammars (CFGs) to define the DSL syntax. The design of the syntax depends on the requirements of the language, which determines the layout and the language components of the DSL instance. Then, the instance is parsed by a built in parser provided by RASCAL to build a concrete syntax tree (CST). CST is a tree-like representation of the instance, which is constructed by parsing the language components with CFG and regular expression. Based on CST, further operations can be applied to complete specific tasks such as code analysis and code generation.

RASCAL also provides the mechanism to map the concrete syntax to abstract syntax, which is a more abstract and generic view of the source code. Listing 2.4.1 is an example of abstract syntax defined in RASCAL. This example illustrates how a statement in a certain programming language can be mapped by abstract syntax to form an abstract syntax tree (AST). Depending on whether it is an `assignment` or `if` or `for` or `switch` statement, different components are extracted from the source code as parameters to form the abstract statement.

Listing 2.4.1 A RASCAL snippet to define the abstract syntax [22]

```
1 data statement =
2         assignmentStat(Id name, Exp exp)
3       | ifStat(Exp cond, list[Stat] thenBody, list[Stat] elseBody)
4       | forStat(Exp cond, list[Exp] body)
5       | switchStat(Exp exp, list[Stat] body)
```

[1] https://www.rascal-mpl.org

After the instance is transformed into an AST, the subsequent operations are performed on the constructors and fields of the data type defined in the abstract syntax. Depending on the intention of the DSL tool, these operations could be checking processes that validate whether rules and constraints are fulfilled. It could also be a transformation process that translates the input language into another desired language.

2.4.2 ClaiR

ClaiR[2] [2] [3] is a Rascal extension that analyzes C and C++ source code based on Eclipse CDT[3] [35]. As the standard library of Rascal does not support parsing C and C++ code, CDT is integrated into ClaiR to parse and translate the code to an AST. AST is an internal and compact representation of the source code. To construct an AST, the source code is first tokenized to remove information like whitespace. Then, a parser analyzes the tokens to determine the syntactic relationship and represents them and their relationships in an AST. In this way, each node of the AST is a token that contains only the required information for the compiler to work. However, the ASTs generated by CDT cannot be directly read by the developer. Thus, one responsibility of ClaiR is to transform CDT ASTs to ClaiR ASTs to make the tree readable in text and to further process with Rascal code [27].

Another task of ClaiR is extracting a so-called M3 model from the ASTs. The M3 model is a Rascal data type that consists of a set of binary relations derived from the AST. In other words, each relation is a higher-level abstraction of an intended part in AST. The composition of these relations extracted from many different C++ files makes M3 model a database that reflects the static semantics of the source code [7]. The following Table 2.2 lists multiple key relations in M3 models. The completed list of relations in ClaiR is in Appendix A. Note that the specifier `loc` in the 'Data Structure' column refers to `location`, which is a Rascal data type representing either a physical part of a file or a logical "qualified" name of a declared entity in the source code [7].

Table 2.2: Relations of the M3 model in ClaiR

Field	Data Structure	Description
containment	rel[loc from, loc to]	Mapping from the logical name of the container, such as class, function, method, to the logical name of the artifact that is contained, such as method, fields and parameter.
declarations	rel[loc name, loc src]	Mapping from the logical name of the artifact to the location where the artifact is declared.
uses	rel[loc src, loc name]	Mapping from the physical location where the artifact is used to the logical name of the artifact. Note that "artifact" refers to any declared C and C++ artifact like a compilation unit, a package, a class, a method, a function, a parameter, a field or a local variable, etc..

[2] https://github.com/usethesource/clair
[3] https://projects.eclipse.org/projects/tools.cdt

callGraph	rel[loc caller, loc callee]	Mapping from the location where the function is called (caller) to the location where the function is provided (callee). Note that the term "function" in the C++ M3 model refers to C++ functions, methods and constructors.
methodOverrides	rel[loc base, loc override]	Mapping from the logical name of the function defined in the base class to the logical name of the function in the derived class which overrides the function in the base class.
functionDefinitions	rel[loc name, loc src]	Mapping from the logical name of the function to the location where it is declared
requires	rel[loc includer, loc includee]	Mapping from the location of the source file (includer) to the location of the header file (includee) included in the source file
includeResolution	rel[loc directive, loc resolved]	Mapping from the logical name of the header to the physical location where the header locates
includeDirectives	rel[loc directive, loc occurrence]	Mapping from the logical name of the header to the physical location where it is included

As shown above, each relation is constructed for a specific relationship between the artifact, varying from class, and function, to macro and variable. We can also find the definition of some relations overlapping with each other. For example, `callGraph` and `methodinvocation` has the same description, containing information about calls between the functions. This is because the relations in the M3 models are built in two ways. Specifically, `methodinvocation` is an initial syntactical collection of the raw data in AST while `callGraph` is a secondary result based on interpreting the earlier syntactical information including `methodInvocation` and `methodOverrides`.

Compared to AST, an advantage of the relation representation is that it is supported by RASCAL primitives. That means the operations on relation and set, such as subtraction, composition, transitive closure, etc., also apply to the M3 model [7]. In this way, developers can write different queries to fetch facts from the model and perform further operations with RASCAL build-in functions. This reduces the time complexity for querying facts compared to traversing the AST. Additionally, the M3 model is also advantageous with respect to storage space usage. Because it is a more compact representation of the source code than AST, it requires less space to store the information. This feature increases the usability of the M3 model, especially when the codebase contains millions of lines of code. On the other hand, the M3 model also has some limitations. As mentioned beforehand, it is a further abstracted view of the source code. The abstraction level is in the direction of architectural models, which contain the source code components that interact with each other but not specifically inside method bodies on the statement and expression level. Nevertheless, if the target under investigation requires information related to the method body, then the M3 model may not be useable. For example, we can neither know the order of the statements and expressions in the source file nor the content of the looping and conditional statements. If these fields are the targets of static analysis, then AST would be a more

informative data source.

2.4.3 TypePal

TypePal[4] is a Rascal library that analyzes the type and the role of the name declared or used in the language. It also enables language users to perform well-formedness checking. In this assignment, as we developed REAL with a DSL to formulate the architectural rule, TypePal is included in the DSL to better regulate the formulation of the DSL instance. The concrete use case will be introduced in Section 4.4.2.

TypePal utilizes a CST to check the language instance. In the CST, the internal nodes of the tree correspond to the non-terminal, while the leaf nodes contain the information of the terminals. In comparison with an AST, a CST has the same semantics as AST but with more information about the instance, as shown in Figure 2.2. In Rascal, the CST also contains textual information such as layout, separators and brackets, while in AST this information is removed. As the checking in TypePal is based on CFG, and languages may have designs for a constrained layout like indentation, CST instead of AST is used as the checking target for TypePal.

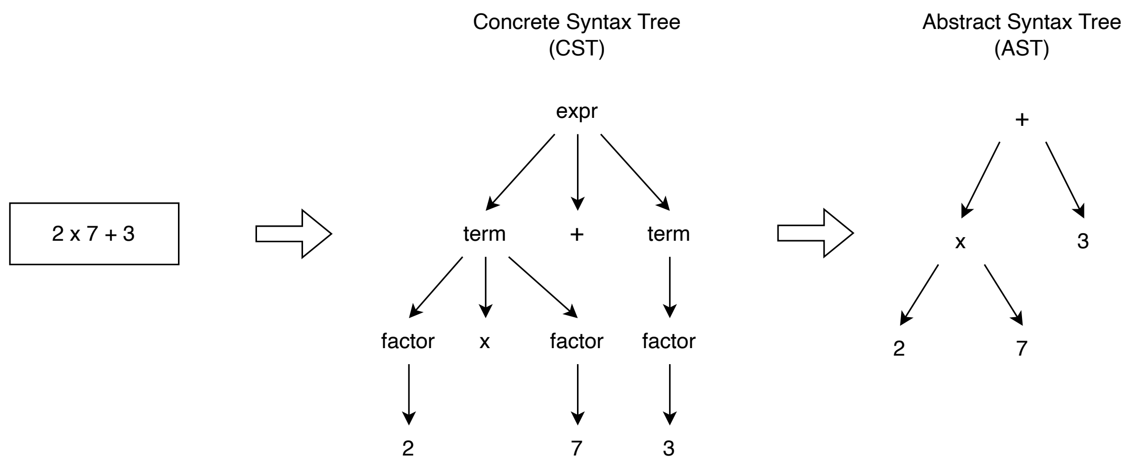


Figure 2.2: Comparision between CST and AST for the same expression

The functionalities of TypePal are realized by a collect-solve mechanism. Specifically, TypePal provides `collect` function to define the allowed action of a syntax component, such as define and use. The effectiveness of these actions can be constrained by giving a scope. Within the scope, users can specify the accessibility and behaviors of the names. In this way, concrete rules like no-use-before-define and no-duplicated-declaration can be implemented. The constraints defined in `collect` function are executed only after a solver is called. This solver solves the constraints to compute a role, a qualified name and optionally a static type for every entity in the CST.

[4]https://github.com/usethesource/typepal

Chapter 3

Related Work

This chapter introduces the study of the previous work related to detecting AE and using DSLs to investigate software architecture. Furthermore, the cases of using RASCAL for source code analysis are summarized. With the related work, we interpret why a metaprogramming solution implemented by RASCAL is selected for the tasks proposed by Philips.

3.1 Detection of Architecture Erosion

As the main task of this assignment is to collect the eroding code segments from the codebase, we decided to study the mainstream methods developed for detecting AE. We expect this theoretical study will provide valuable insights into AE management and inspiration for designing our tool.

The majority of the detection methods for AE are based on architecture conformance checking. More specifically, it verifies the mismatching between the intended and actual architecture abstracted from the source code implementation. One intensively used technique in this field is DSLs for expressing design rules. Haizer and Zdun developed a semi-automated method with a DSL to specify the architectural rules and compare the rules with the abstracted representation of the source code extracted from the Unified Modeling Language (UML) [19]. Similarly, Gurgel et al. proposed a DSL tool named TamDera to define the architectural rules and detect architectural violations. Another widely used method is the reflexion model introduced by Murphy et al. [30]. This method compares two models —, a source model abstracting the information in source code and a high-level model representing the design decisions —, to determine the gaps between design and implementation. Apart from performing checks directly on the architecture, plugin dependency [16] and exception control flow [12] are also used by researchers as the target of architecture-level consistency checking.

In addition to architecture conformance checking, AE can be detected by tracing the relationships between the artifacts during software evolution. Hassaine et al. present a method called ADvISE, which generates a class diagram for each release of the software through reverse engineering. The diagrams of the adjacent releases are then compared with the designed metrics to explore architectural drift and erosion [20]. Besides, defects such as code smells are used by some researchers as indicators for AE. Bertan determines that architectural violations correlate strongly with some smells, including the god aspect and long method. Based on this fact, several metrics are proposed to indicate the degree of AE by quantifying the smells [8].

3.2 Domain-Specific Languages for Architecture Analysis

As Section 3.1 mentions DSLs are a technique widely used for architecture conformance checking. To further investigate the usability of DSL in checking the consistency between rules and architecture artifacts, we study six state-of-the-art DSLs. Table 3.1 gives an overview of these DSLs.

From the table, we can perceive that all these DSLs can explicitly describe the constraints on the implementation architecture that could be violated. If we explore deeper for each tool, we can find that although the implementations of the six DSLs are all based on conformance checking, the details differ depending on the requirements. The most common way is to declare the architectural rules with the designed DSL and compare the source code with these rules to check AE. However, other types of input data are also applicable to DSL tools. For example, the object for checking in `Architecture abstraction` DSL is in the format of a UML class diagram [19]. Besides, the implementations of `DepCoL` and `ArCatch` are not based on architectural rules. Instead, the rules are derived from the plugin dependencies and exception handling respectively. For example, in [12], a software with a Model-view-controller (MVC) architectural pattern was inspected. Rules are defined to regulate the specific exceptions that can only be raised by `Controller` and only handled by `View`. To summarize, these tools prove that the DSL methodology is a common solution to analyze erosion on the level of software architecture, especially to detect AE. The language provides the flexibility to be tailored for specific analysis targets by accepting various formats of source data as input.

Table 3.1: Description of six state-of-the-art DSLs

DSL	**Features**	**Advantages**	**Disadvantages**
Architecture abstraction DSL [19]	Mapping the UML class diagram of a program to the architectural rules declared by the DSL to generate a view of the architectural element	Increased traceability, Verified usability	Input format limited in UML
TamDera [18]	Declaring architectural rules with DSL to detect the architectural violation and drift at the same time	Improved modularization, High reusability	Limited expressiveness
Dictō [9]	Connecting the DSL with a third-party tool through a customized adaptor to complete various checking tasks	Improved modularization, High reusability	Adaptors as the mandatory components between DSL and extension
DepCoL [16]	Using the dependency relationships between the plugins as the rules to check architectural violations	Easy to learn, High extensibility	Exclusive for plugin-based systems, Low portability
DCL 2.0 [34]	Declaring the rules in a modular manner to analyze architectural violations with an external checker	High reusability, High expressiveness	Input source code limited in Java
ArCatch [12]	Using exception handling workflow as rules to check architecture components violating the rules	Effective for exception handling checking	Limited usability

3.3 Code Analysis and Transformation with RASCAL

Since a DSL has been regarded as a reliable solution for AE analysis, we would like to explore further by investigating a specific metaprogramming language. Here, we choose RASCAL as it has been extensively studied for a wide range of tasks related to source code operations. On the one hand, it has forward engineering features for making the syntax and semantics of the DSL. In [27], a DSL is designed to modernize the legacy code with RASCAL. With the DSL, users can formulate the transformation rules, which specify how the legacy code should be updated.

On the other hand, RASCAL has the reverse engineering features for extracting the information from the source code that is needed to execute the code analysis, transformation and generation. Back to [27], the implemented DSL is coupled with the reverse engineering features of RASCAL to modernize the legacy code. This process includes inputting the legacy code into the RASCAL program, matching source code with the components used by transformation rules, and generating the modernized artifacts based on the rules. Additionally, in [35], RASCAL is used to migrate test code. The original test code implemented within the STX framework is transformed to the new code conforming to GoogleTest APIs. This transformation includes the conversion of C to C++, the replacement of headers, and the rewriting and refactoring of CMAKE files. Similarly, [21] reveals how a concise RASCAL script takes Java source code as input and performs complex refactoring with Eclipse Java Development Tools(JDT) within a short time.

Combining the forwarding and reverse engineering features, RASCAL allows the users to implement the DSL as well as to conduct C++ source code analysis in the same framework. As a summary of the above studies, we can find that RASCAL enables DSL users to have an efficient and flexible solution to analyze the source code and transform the input code while maintaining the semantics.

If we explore these related cases at a higher level of abstraction, we can find a common paradigm when designing and implementing RASCAL programs for code analysis and transformation tasks. This paradigm is summarized in [23] and called the EASY paradigm. It contains three steps: **e**xtract, **a**nalyze and **sy**nthesize. As shown in Figure 3.1, the source data is extracted and abstracted only to maintain the relevant information. Here, RASCAL accepts a wide range of source data types as input, including source code, logs, documents, etc. Then, the intermediate data undergoes multiple iterations of analysis and transformations based on the requirements of the tasks. Ultimately, the results are synthesized into the desired format to output.

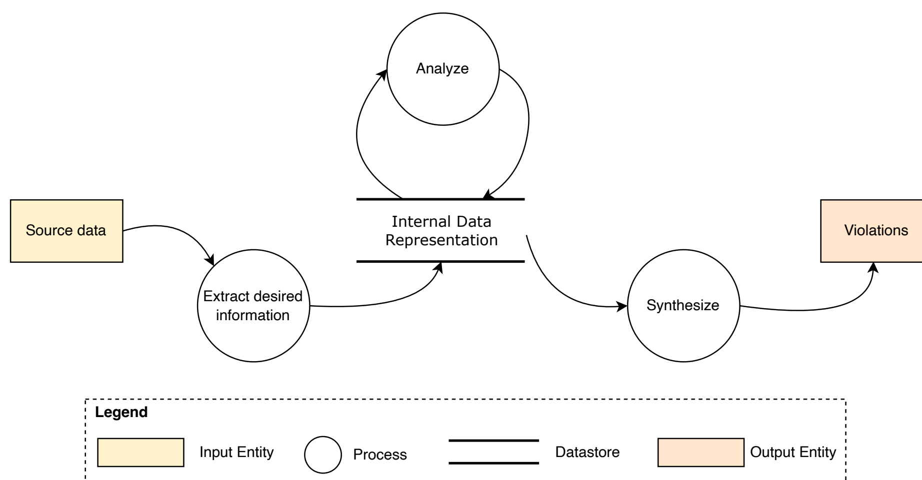


Figure 3.1: Data-flow diagram of EASY paradigm for the design of DSL tooling with RASCAL (adapted from [23])

3.4 Rule Formalization with DSL Syntax

The previous two subsections have revealed the power of explicit design rule formulation with DSLs for analyzing software architecture and performing modifications on the source code. It seems DSL is a desirable candidate to tackle Philips' tasks of detecting AE. The last part we must investigate is whether the design rule formalization is competent in implementing architectural rules. In Section 3.2, we have ascertained that DSLs can be used to define the rules by designing a context-free grammar ([18], [34]).

Furthermore, we find a common pattern they used to define the syntax. Figure 3.2 is the pattern summarized in [34]. The architectural rules can be generalized with two modules and one constraint. Here, modules are artifacts, such as functions, classes and packages, representing the rule's subject and object. The constraints are the modifiers defined in DSL syntax that constrain the behavior of the modules. By composing two modules with the constraint, a statement of the architectural rule is generated. This is a unified way to formalize the rules regarding the DSLs we studied.

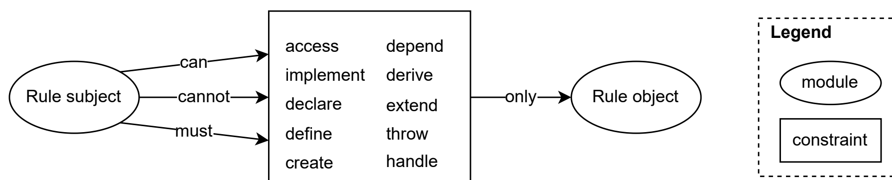


Figure 3.2: A common pattern for architectural rule language design (adapted from [34])

As suggested by some researchers ([18], [9] and [34]), this pattern can be further modularized by encapsulating the rule declaration into a package. This package is then imported into other independent files. This design increases the reusability of the DSL and the conciseness of the rule definition.

3.5 Conclusions

The above literature study provides us with a clear view of which state-of-the-art methods are being applied to analyze software architecture and detect AE. After weighing the usability and practicability of all the above methods, we decided to use DSLs to solve the problems in Philips' codebase. The main reason for choosing this method is that architecture conformance checking requires the formulation of the architectural rules, while the literature studies show with a DSL, the rules can be explicitly formulated. These use cases give us confidence that DSL is also a feasible solution for Philips' case. Another factor contributing to our decision is the ability of DSL to express the architectural rules. Methods other than DSL, like reflexion model, are also capable of AE detection. However, the assignment's goal is not only conformance checking but also representing the rule in a formalized and human-readable format. To implement this requirement, a DSL implemented with RASCAL is a suitable approach RASCAL provides both forwarding engineering features in language design for rule formulation and reverse engineering features to analyze the C and C++ code. From the technical perspective, we consider RASCAL a suitable tool to implement the DSL because the EASY paradigm offers a feasible template for our tool design. The study on related work with RASCAL has also proven that this metaprogramming language is practical for large-scale source code analysis [35].

In addition, the extensibility of the DSL retains the potential for adapting the developed tool to the new requirements proposed in the future. The implementations in [35] and [27] show that DSL performs well for both AE detection and code refactoring. Once the AEs in the codebase were expected not only to be detected but also to be managed and eliminated, a DSL-based tool could still be extended to meet the requirements. In comparison, the usability of the other mentioned methods is limited as they only focus on AE detection.

In conclusion, considering the feasibility, expressive capacity, practicability, usability and extensibility, we believe a DSL toolkit implemented in RASCAL is a suitable solution to check AEs in Philips Positioning Software.

Chapter 4

Design and Implementation

In this section, we elaborate on how our tool READ is designed and implemented to check AEs in Philips' codebase. Specifically, the first subsection describes the procedures for tool development and outlines the structure of the tool. Then, each part of the tool is elucidated in the subsequent subsections. In the end, we summarize the overall design and propose future work based on the current implementation.

4.1 Development Procedures

As illustrated in Figure 4.1, the development of the DSL tool is comprised of several steps. Before the start of the procedures, it is essential to have a clear delimitation of what system is under study in this thesis. During the development phase, the entire Philips' positioning software is the target for investigation. In other words, all the C and C++ source and header files, the CMAKE files, are the objects of READ to analyze and identify architectural violations. As for the evaluation phase, we use the same system to evaluate the READ's implementation and the quality of the checking results. Instead of directly on the source code and CMAKE files, the resources used for evaluation may be abstract, such as M3 models and the extracted include paths, based on the content of the evaluation. We will interpret it in detail in Chapter 5.

As the goal is to develop a tool that is capable of executing automated checking on a codebase to collect the artifacts that violate the architectural rules, we need first to collect the requirements by interviewing Philips' engineers. Then, we need to have knowledge about the rules before starting to design the tool. These rules are then deconstructed and represented by the components in the source code. This process requires us to have prior knowledge about the codebase such that we can map the rule subject, object, and constraints to concrete code components. In this way, we can have a preliminary idea about how to describe the rules with DSL grammar and how to check the rule with RASCAL.

As READ is developed to formulate and check the architectural rules, the first two steps of tool development are to identify the requirements and the architectural rules. These two steps are complemented by interviews with the architects and the developers. Additionally, some architectural rules are also recorded in the software design documents [1]. After a list of architectural rules is obtained, we prioritize the rules and check the rules based on the prioritized order. It is necessary to have a priority of the rules as we can only check a limited number of rules within the given time. The criterion of the prioritization will be discussed in Section 4.3.

When the preparation work for architectural rules is finished, we employ an iterative and incremental strategy to develop the tool. More specifically, every time a rule is selected, we design how to express the rule in the DSL and decide how the check should be implemented for this specific rule. The outcome is a list of violations from the source code. Based on the accuracy of the results, the implementation of the rule checking is iteratively evaluated and improved. More specifically, the inaccuracy can be both from imprecise rule definition and imprecise rule checking. The former requires a more precise rule description and refinement of DSL syntax, while the latter requires bug inspections. After

finishing the checks on one rule, we progress toward the next rule. The reason we adopt an incremental way of development is that we would like to check the feasibility of the design with regard to the whole workflow. That means when the first rule is checked and the implementation is validated to be correct, we are confident that the implementation of other rules can follow a similar pattern without radical changes in the design.

Once the tool is capable of performing conformance checking for multiple rules, it is deployed and executed from the command line periodically to detect the violated segments from the newly modified source codes.

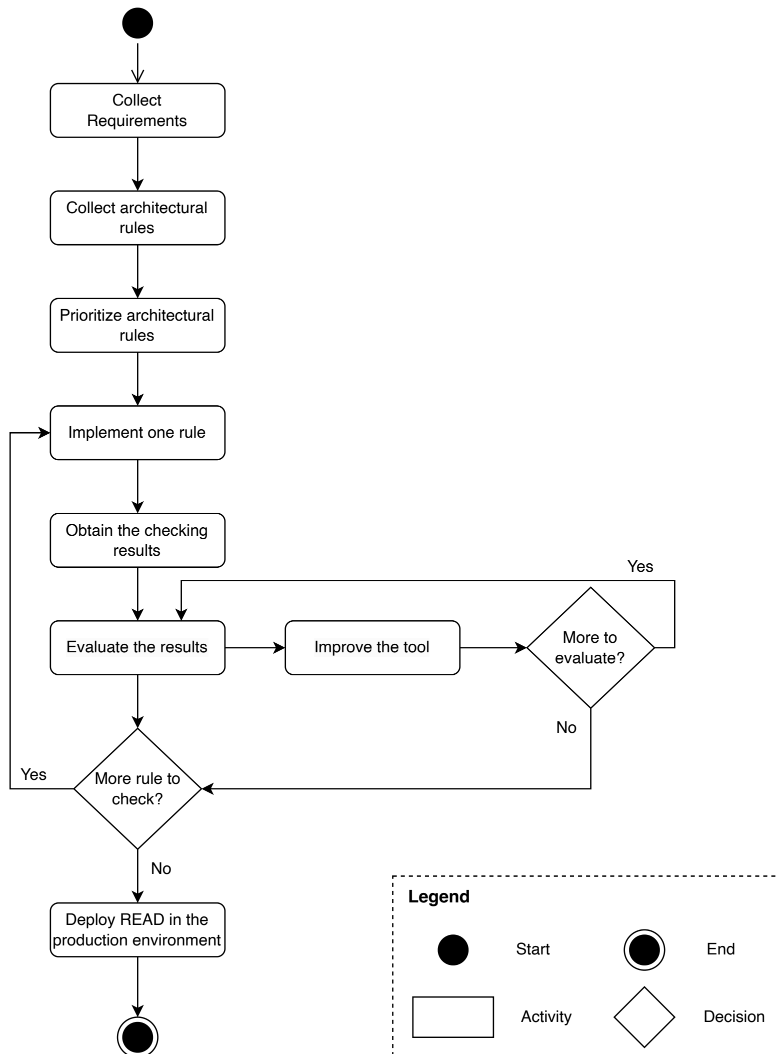


Figure 4.1: A UML activity diagram illustrating steps to develop the DSL tool

4.2 Requirements

To develop a tool that can identify architectural violations and deliver the results to the users automatically, we need first to collect and analyze the requirements. In this assignment, the requirements were collected by interviewing the Philips engineers. In the following, we will present the requirements.

REQ1 The tool must be able to formulate the architectural rules decided to be investigated in this assignment.

It is expected to collect a set of architectural rules defined for Philips' positioning software. Nevertheless, due to the limitation of time and the workload required to implement each rule, it may not be possible to implement all rules. That's why the rules are prioritized before the implementation of the tool. We must ensure that for rules selected to be checked, the syntax of the implemented DSL must be expressive enough to describe the rule precisely.

REQ2 The tool must be expressive and extensible to define other architectural rules.
In terms of the architectural rules that are not selected in this thesis to be implemented in the tool, it would be best if they could be formulated with the already designed DSL syntax. In case the DSL syntax is not expressive enough to describe the rule, the tool must support the DSL users to extend the syntax to accommodate the new rules.

REQ3 The tool must be able to report accurate results of architectural violations with respect to each architectural rule.
We need to ensure that the tool is reliable and that the identified violations are indeed violations of the rules. The false negatives in the results mean the tool fails to detect the existing violations, while the false positives mean the tool reports the facts that are actually not violations. Both two kinds of false results should be avoided as much as possible to improve the reliability of the tool. Otherwise, we are not confident with the checking results, and the tool is not usable.

REQ4 The tool must be able to be integrated into the production environment of the positioning software.
The ultimate target of the tool is to facilitate architects' work in maintaining the architecture of the codebase and notifying the developers of the newly introduced violations after they change the code. Therefore, the tool must be able to be deployed on the server and deliver the new results when the source code is changed.

REQ5 The tool must be able to deliver the checking results within a reasonable time. As a lowest requirement, if the tool is running overnight, then the results should be available the next morning. The tool is expected to be efficient in reporting the violations. Otherwise, new changes may be introduced in the source code during the long execution time of the tool. In this case, the results may be outdated, reducing the usability of the tool.

4.3 Architectural Rules

Collecting architectural rules is the premise for all of the subsequent design and implementation work. The rules are mainly proposed by architects responsible for designing and maintaining the positioning software. Additionally, some rules are also mentioned in the internal documentation [1]. In the following, we will introduce the first three of the prioritized architectural rules collected through interviews with three architects. The complete list of the architectural rules can be found in Appendix B. Each rule description is attached with the rationale for why these rules are proposed specifically for Philips' codebase.

Table 4.1: Dependency relationships between the layers

Dependent Layer	**A**	**B**	**C**	**D**	**E**	**F**	**G**
Dependee Layer	B, E	C, D, E	D, E	E	-	B, C, D, E	D, E

RULE 1 **Each layer must only depend on the other allowed layers. The prescribed dependencies are demonstrated in Table 4.1 and Figure 4.2.**
In the dependency table, dependent means the layer requiring the services from the other layer. Dependee denotes the layer that provides services to the dependents. The E layer is

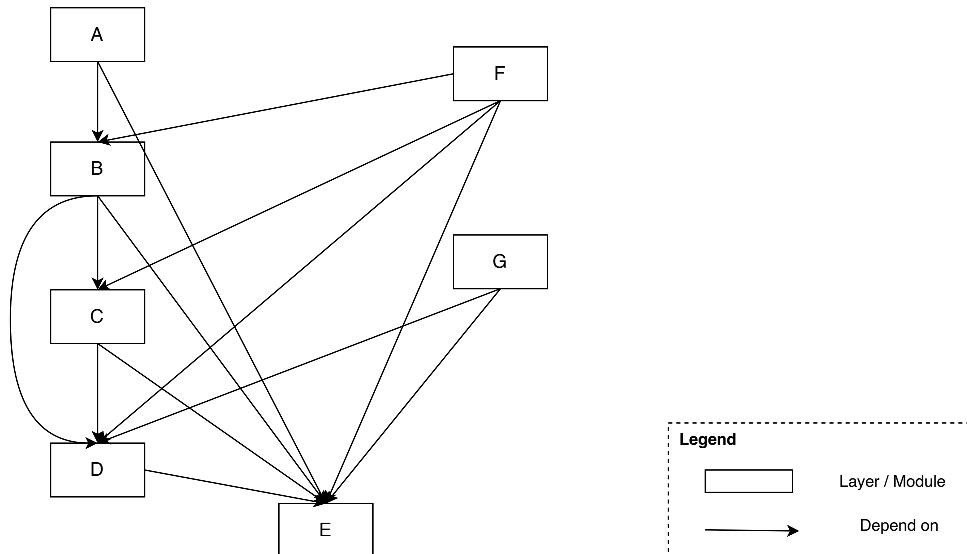


Figure 4.2: Visualization of the desired layer dependency

not allowed to have any dependee as it is a utility layer that cannot rely on the implementation of all the other layers. As mentioned in Section 2.2, the layers are presented in a vertical order. This rule is necessary to be able to guarantee uni-directional data flow, but not enough to guarantee it by itself. In other words, this rule contributes to avoiding invalid data access and layer skipping. Overall, the intuition of this rule is to decouple the layers and ensure each layer is only responsible for its designed functionality.

RULE 2 **The communication between the layers must be conducted exclusively through the interfaces defined in `pub` folder.**
In Philips' codebase, the header files are categorized into two groups based on their scopes: public headers in `pub` folders and included headers in `inc` folders. The public headers can be accessed by the external modules or layers, while the included headers are exclusively for internal usage. As the interface is designed as an adaptor for the external modules to access the components defined in the module that provides the interface, it belongs to the public header. This rule is designed to maintain layer decoupling and control the hide implementation so that only the interfaces defined as public are visible to external users.

RULE 3 **It is not allowed to have dynamic memory allocation after the startup phase of the applications.**
Each layer of the positioning software has its own application, which is a C++ class implementing the same abstract class. The life cycle of the application includes a startup phase, an execution phase, and a shutdown phase. Each phase is characterized by one or multiple functions declared in the abstract class. This rule specifies dynamic memory allocation based on functions `alloc`, `malloc`, `calloc`, or operator `new` is only permitted during the startup phase. This rule is proposed to prevent the applications from crashing during clinical use, which corresponds to the application execution phase. If dynamic memory allocation is allowed in the execution phase, the related functions may be called depending on the real operation scenario. In this case, the exact amount of memory utilized is unknown. It is possible that memory exhaustion occurs, leading to the termination of clinical operations due to application crashes. Prohibiting all memory allocation in the execution phase forms a stronger guarantee that such a calamity will never happen.

By comparing all the rules, we can find some rules, like RULE 1 and 2, are generic and can be directly reused in other software, while some are highly specific to a module in Philips' positioning

software, such as RULE 3. In summary, among the 13 collected architectural rules, four rules are specific to the codebase, and nine rules could be applicable and reusable to constrain the architecture of other software.

As mentioned earlier, the architectural rules are prioritized for subsequent development. The priority is determined by three factors: the risk associated with the rule violation, the practicability of rule definition and violation checking with a DSL tool, and the availability of ground truths to evaluate the implementation quality. For instance, we decided to prioritize placing the rule related to layer dependency in the first position, as the architect has already developed another tool to check it. Thus, we can use the results of the architect's tool as ground truth to evaluate the accuracy of our implementation. With regard to the risk of violation, the rule with high impact on the operation of the software. For example, a rule associated with deadlock is assigned high priority as it may lead to a system freeze. Regarding the relevance to analysis on an architectural view, some rules are ranked lower. This is especially the case for the last rule aiming at avoiding hardcoded passwords. The hardcoded passwords are hidden below the architectural level of the source code, while the target of this assignment focuses on architecture erosion. Therefore, this rule is out-of-scope and assigned the lowest priority.

4.4 Workflow

Given that we have collected all the architectural rules, we proceed to the design and implementation of the DSL tool. In this section, we first illustrate a high-level view of the DSL tool with respect to workflow and data flow. Then, the detailed design of each tool component is interpreted in several subsections.

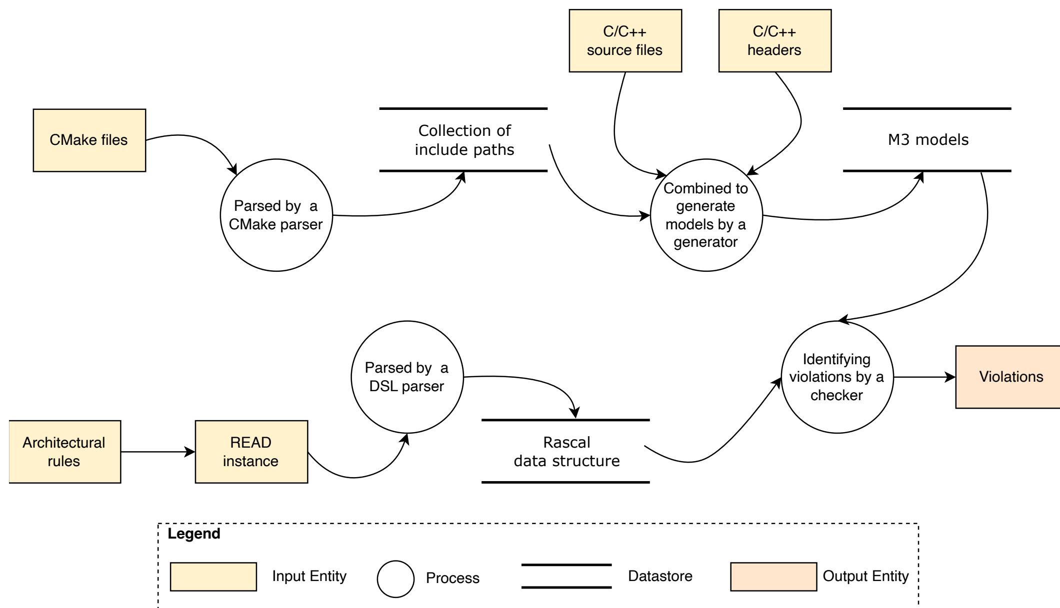


Figure 4.3: Data-flow diagram of the DSL tool

Since the study on the related work promotes the use of the EASY-paradigm [23] is an effective design for a DSL tool aiming at software static analysis. We decided to follow this paradigm to construct our own tool to check rule violations. As depicted in Figure 4.3, this tool operates mainly with four components: a CMAKE[1] parser, a model generator, a DSL parser and a violation checker. DSL parser is a context-free grammar written in RASCAL that generates a parser. A parser translates the input source text to parse trees, from which the desired rule components can be extracted for further analysis.

[1]https://cmake.org/

The model generator and violation checker are the components that complete extraction, analysis and synthesis. The generator extracts architectural information and composes it into models while the checker inspects problematic code segments and summarizes violation facts. As for the other two components, they make the preparation for the work by transforming the source data into desired formats. With the cooperation of these four components, the DSL tool is capable of completing tasks for rule violation checking.

From the aspect of data flow, the execution of the tool requires three kinds of data. The first one is the C and C++ header and source files provided by the codebase. They are the research subjects of this project. The second type of data is the paths collected from the CMAKE files. These CMAKE organize the source files in the codebase and facilitate the build process. The last essential data source is architectural rules. We need to apply domain knowledge about the positioning software to resolve the rules and map the important rule information to a representation described with the DSL. In Figure 4.3, we can find two parallel dataflows starting from the codebase and architectural rules, respectively, eventually converging at AE-checker for violation checking. That means the checker essentially compares code implementations and rule designs to find the divergences in between. Overall, the implemented tool accepts the CMAKE files, C/C++ code and the architectural rules are input and exports the rule violations as output.

To have a better understanding of how the tool composes the data together to derive the violation facts, a concrete example is illustrated in Figure 4.4. In this example, a rule specification is implemented aiming at checking the disallowed dependency of a module called `C`. This module is permitted to include the headers from two folders in module `A` and one folder in module `B`. The checks are expected to be performed on the C++ files. With the DSL parser, the instance is parsed to build a CST, which is further abstracted and stored in RASCAL data structures. Here, the single string in the instance is reserved as a string, while the variables in the instance, such as directories and functions, are converted into a binary relation. In this way, we have an internal representation of the instance input.

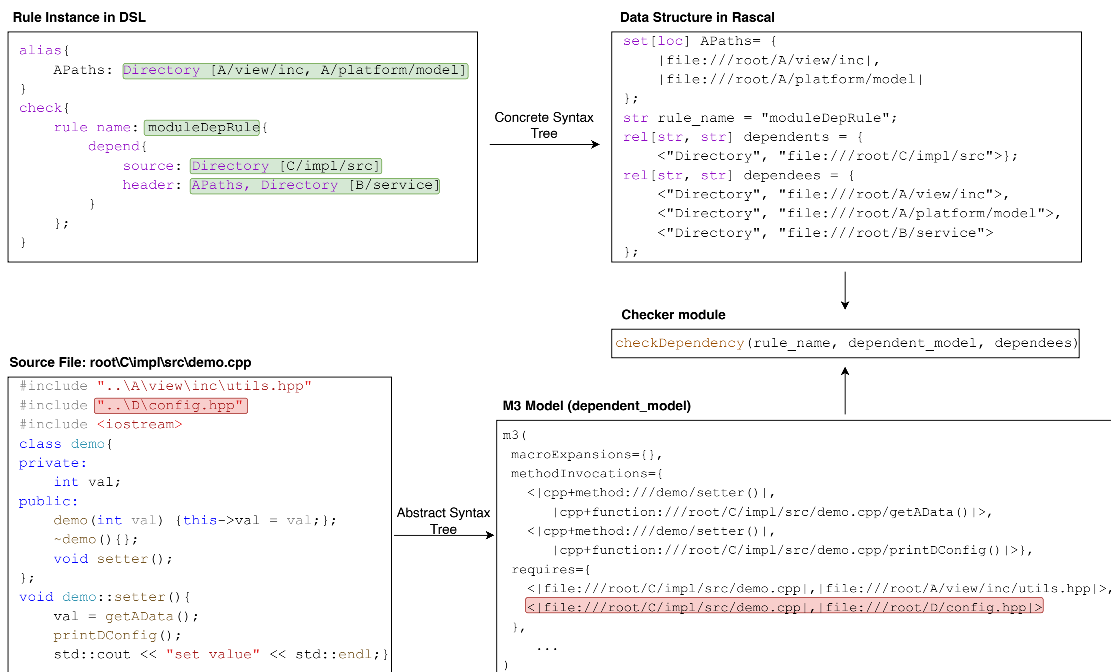


Figure 4.4: A concrete example of architectural rule checking

As the target of this rule is to check the invalid inclusion in the source files of module `C`, the checker checks the source files within the specified `dependent` folder one by one. In Figure 4.4, a source file named `demo.cpp` is checked. This file includes a header from module `D`, which is a violation as it builds a disallowed dependency between module `C` and `D`. To mine this violation, the source file is first

parsed to an AST and then composed to a M3 model, in which the `requires` field contains the fact of violation. In the last step, a checking function is implemented to compare the data in the `requires` field with the allowed `dependees` specified in RASCAL data structure. In this way, the disallowed headers are extracted and stored.

Having known the workflow of the DSL tool, we can now have a detailed view of the aspect of the implementation from READ's UML package diagram in Figure 4.5. This figure reveals how the project is organized and how the modules relate to each other. Here, a module is a term in RASCAL denoting the collection of codes in the source file. Each source file is encapsulated as a module that can be imported by the other source files. For the sake of simplicity, the main function that includes all the modules is not depicted in the figure. This function determines the direction of the data flow and the execution order of the modules. As shown in the figure, the DSL tool is composed mainly of two packages. The CMAKE parser package contains a syntax module to match the lexical grammar and syntax of the CMAKE. The matched CMAKE elements are extracted and composed to a CST by a built-in parser in RASCAL. As the operations of the include path extractor are based on the AST of the CMAKE code, the generated CST is further transformed to an AST through the RASCAL function `implode`. This function generates an abstract view of the CST such that the nodes are presented with RASCAL built-in types or data structures. They are then used to extract the include paths to collect the directories of the source files, headers and include paths. Although RASCAL also supports code analysis directly on CST, our CMAKE parser's implementation relies on the AST. The decision was made during the development stage when we were unaware that the include paths could be directly extracted from the CST. As transforming the implementation from being based on AST to CST would require considerable effort, we decided to stick with the original AST-based implementation. To summarize, the CMAKE parser takes the CMAKE files as input and exports the desired directories as output from the extractor to the external modules.

In terms of the DSL, a syntax file is implemented such that the architectural rules can be formulated with the provided grammar. Another source of the input, C/C++ source files and headers, is forwarded to the model generator module, which requires the include paths from the CMAKE parser to transform the source files to M3 models. The DSL parser, on the one hand, receives a DSL instance of the architectural rules and parses the content to build a CST. This process is the same as the parsing phase of the CMAKE parser. On the other hand, every time a rule is parsed, the parser requests the checker to check the rule by calling the related checking functions. In the checker module, multiple checking functions are implemented to collect violations by comparing the rule constraints with the facts in M3 models. Here, no explicit dependency between the model generator and the checker exists because all the generated models are saved to disk during model generation and read from the disk during rule checking. Each parsing process is followed by a checking process until all the rules are parsed and checked. Eventually, a list of violations is collected. Additionally, a language register is implemented to register the DSL and provides some IDE features in the instance, such as syntax highlighting and code navigation, enabled by the standard libraries in RASCAL and TYPEPAL. We also implemented an internal utility package for the DSL modules. This package offers functions facilitating the filtration on the fields of the M3 models and the generation process for the paths and models.

Except for the two main packages mentioned above, a utility package accessible for both packages is implemented. Besides, three standard libraries or extensions of RASCAL are required for the tool's implementation. RASCAL library is the foundation of all the implementations, while modules from TYPEPAL and CLAIR are imported depending on the specific requirements of the DSL. The implementation of the CMAKE parser is maximally decoupled from the DSL tool, with only the path extractor providing all the necessary data to the external modules. As the CMAKE parser merely depends on the RASCAL library and custom utilities functions, it may be reused in other CMAKE projects that have a similar naming convention as the codebase.

4.4.1 Model Generation

This section demonstrates how the M3 models are generated given the C and C++ source files. It consists of two components: a CMAKE parser to collect the include paths for each source file and a model generator that extracts information from source code and transforms it into M3 models.

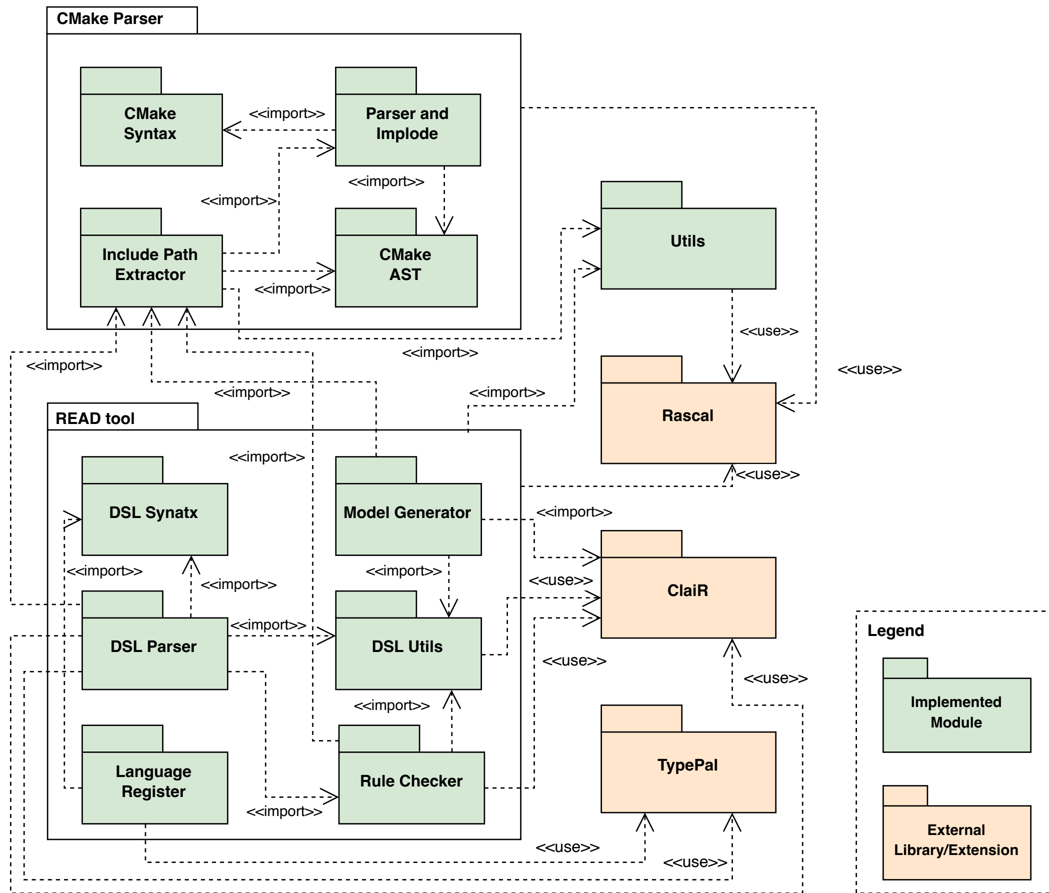


Figure 4.5: UML package diagram of READ

CMake Parser

The goal of model generation is to transform source code into M3 models. This process mainly relies on the functions `CreateM3FromCFile` and `CreateM3FromCppFile` provided by ClaiR. The functions require not only the directory of the source file but also the include paths used by the source file. For C and C++ programs, include paths are the directories to the folders that contain the header files included by the source file. These paths are equivalent to the parent directories of the headers specified with `#include` syntax in C and C++ source files. Include paths are important for the accuracy of READ because name resolution and type resolution for source code artifacts will be incomplete without an accurate include path. Incompleteness will lead to both false positives and false negatives in the design of rule violation detection, with unknown quantities.Therefore, to successfully construct M3 models, we need first to collect include paths.

The include paths can be characterized by three data sources: codebase header files, headers of the third-party dependencies and the standard libraries. As the include paths are required by the pre-processor during the compilation, we decided to parse the CMake files to extract the paths. CMake is a tool to construct compilation commands and manage building processes with a C++ compiler [28]. It enables the developers to use CMakeLists.txt to describe the build configurations by specifying the include path, source files and compiler options. These configurations are grouped within a CMake project, which is the elementary unit in CMake that determines the scope of the defined variables and paths. Software may contain multiple CMake projects, which are connected by calling the command `add_subdirectory`. Each project represents a target, which can be a library or an executable. They are a composed representation of the specified, include directories and dependencies. In Philips' code-

Listing 4.4.1 An example of CMakeLists.txt in the codebase

```
project(A)
set(TARGET_NAME A)
set(${TARGET_NAME}_IDE_FOLDER "A")
add_subdirectory(SubA)

set (${TARGET_NAME}_DIRS
    ${EXTERNAL_PATH}
    ${EXTERNAL_PATH}/tools
    ${A_PATH}/service
)

set(${TARGET_NAME}_SOURCES
    ${A_PATH}/include/AConfig.hpp
    A.hpp
    A.cpp
)
configureLibrary(${TARGET_NAME} FLAG)

```

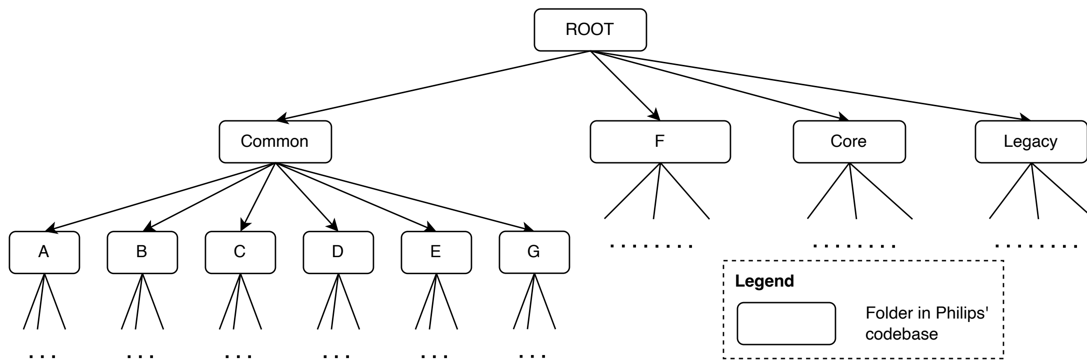


Figure 4.6: A illustration of part CMAKE tree in Philips' codebase

base, the CMAKE projects are eventually encapsulated into a Visual Studio Solution, which leverages Visual Studio's build system to build and execute the program. Listing 4.4.1 is a simplified CMakeList.txt in the codebase. Here, a project named TEST with the target name and required include directories as well as source files are defined. The specified paths are further encapsulated into a library. The project is linked to another subproject `SubTest` such that the whole software is configured recursively. That means, to collect all the include paths, we need to start from the root project and build a CMAKE tree as illustrated in Figure 4.6. The tree traversal is a depth-first search (BFS) process which we only begin visiting a subtree when the include paths at all the leaf nodes of the last subtree are extracted. In this way, the parsing process starts from folder A and ends with the folder for legacy code.

From a practical standpoint, the implementation of the CMAKE analyzer should simulate the working process of CMAKE based on the BFS algorithm. We first implement a syntax file in RASCAL to match the CMAKE grammar and build a parse tree. In this way, each component presented in CMAKE text is mapped to a node in CMAKE AST. After that, operations are applied on specific AST nodes. Figure 4.7 reveals the workflow when parsing the files. As the CMAKE project determines the visibility and accessibility of the include paths and source files indicated in the CMAKE files, we use a tree to store the defined variables by projects. Specifically, the project name is the key of the outer map, while the variable name and a list of values constitute the pair of the inner map. Besides, we construct another map to record the project hierarchy with the name of the child project as key and the parent project name as value. A third map is created to store the specified directories by CMAKE file. In this way, every time a `add_subdirectory` command presents in a CMAKE file, a new key-value pair is added

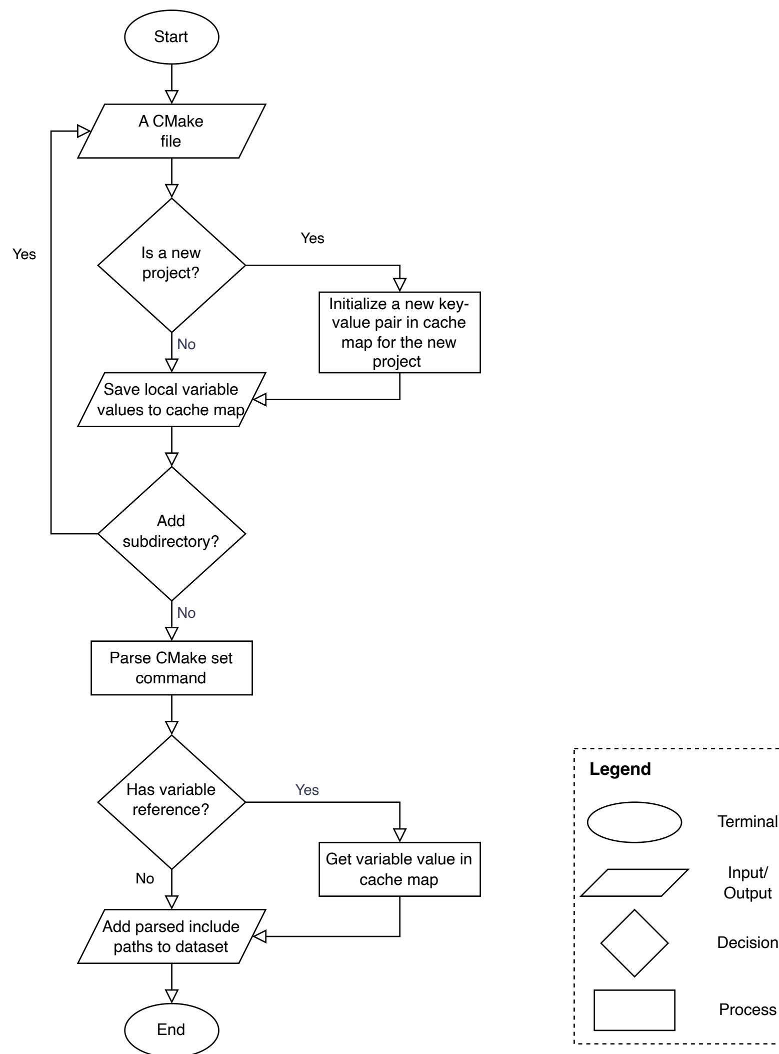


Figure 4.7: The simplified flowchart for CMAKE parsing process constructed in accordance with ISO 5807 [13]

to the maps. This is a recursive process until a leaf CMAKE file is reached.

Moreover, CMAKE allows the directories to be specified with variable references, as shown in lines 7 to 9 in Listing 4.4.1. The resolution of the references is achieved by checking the key-value pair of the current project in the data map. In case the reference is not defined in the current project, the parser will proceed to search the paths provided in the parent projects successively until reaching the root project. At the end of the analysis, each CMAKE project is transformed into a list of directories representing the include paths and source files. These lists are forwarded for the model generation process described in the next subsection.

Model Generator

After identifying the include paths for each source file, a model generator is implemented to build M3 models by applying the functions `CreateM3FromCFile` and `CreateM3FromCppFile` provided by CLAIR. Internally, both functions forward the directory of the source file and are required include paths to Eclipse CDT to parse the C or C++ code in the source file. As a result, each source file is transformed into a single CLAIR AST. However, AST is not an optimal representation of the source code

as it is time-consuming to visit all ASTs, considering a large number of source files in the codebase. Besides, compared to the M3 model, AST is a large and dense data structure that requires a larger space to store the data. Thus, the CLAIR ASTs are further extracted and composed into M3 models. M3 models provide us with the flexibility to search for different kinds of source code information with RASCAL primitives. Eventually, each source file corresponds to a M3 model, which is stored as a binary file for efficient IO operations. We also build for each layer a composed M3 model with the function `composeCppM3` to prepare for the layer-based analysis in the checking phase.

Algorithm 1 M3 model generation

Input:
A {`sourceToIncludePath`} map with source file location as key and a list of include paths as value
A {`sources`} list containing the location of all the C and C++ files in the codebase
A {`stdLib`} list containing all the locations of all standard libraries required by the source files
A {`defaultIncludePaths`} list containing the default include paths
Output:
Binay files of M3 models

```
1:  for each s ∈ sources do
2:      model ← emptyModel()
3:      if s ∈ sourceToIncludePath.keys then
4:          if s is a C file then
5:              model ← createM3FromCFile(s, stdLib, sourceToIncludePath[s])
6:          else
7:              model ← createM3FromCppFile(s, stdLib, sourceToIncludePath[s])
8:          end if
9:      else
10:         if s is a C file then
11:             model ← createM3FromCFile(s, stdLib, defaultIncludePaths)
12:         else
13:             model ← createM3FromCppFile(s, stdLib, defaultIncludePaths)
14:         end if
15:     end if
16:     path ← createSavePath(s)
17:     writeBinaryFile(path, model)
18: end for
```

Algorithm 1 provides a more concrete view of the implementation for the model generation process. As mentioned in the last subsection, each source file is mapped to a list of include paths after the CMAKE parsing process by implementing the data structure `sourceToIncludePath`. However, not all source files are presented in CMAKE, as some source files are directly used to build a solution or generated during the build process. Therefore, a default list of include paths `defaultIncludePaths` is prepared for these undetected source files.

Moreover, the C and C++ standard libraries are excluded from the CMAKE files because the paths to these libraries are, by default, included in the compiler search space. To enable the model generation, these paths, stored in the list `stdLib`, are forwarded explicitly to CLAIR. From the algorithm, it is clear that each source file is evaluated whether it is detected by the CMAKE parser. If so, then the M3 model is generated with the corresponding collected include paths. Otherwise, the default paths are used. In the end, the generated M3 models are written to disk as binary files for subsequent checking processes.

While generating the models, RASCAL may report messages that certain include paths for a specific source file are missed. The possible causes could be that the CMAKE parser does not parse the paths correctly, or some CMAKE features defining include paths are not discovered by the CMAKE parser. To diagnose the problems, we compare the parsing results with the corresponding CMAKE files and improve the implementation iteratively. This process contributes to the minimized number of missed

include paths and enables the M3 models to present more comprehensive information given the source files.

4.4.2 Design of the Domain Specific Language

Apart from converting source code into M3 models for better information querying, another preparation work for rule checking is to translate the representation of architectural rules from natural language expression to data structures readable for RASCAL. In the context of this assignment, we mainly focus on the implementation of the first three architectural rules listed in Section 4.3. By applying incremental development, we initially designed the DSL syntax for the first rule and then extended it to accommodate the description of the other rules. To facilitate the description of the DSL design, we first demonstrate the overall language structure with a DSL instance. Then, the functionality of each language component is interpreted.

Language Layout

A READ instance is comprised of three blocks: an alias block, a template block, and a checking block. As presented in Listing 4.4.2, the alias block allows the user to define an alias name for a list of variables. The declaration of the variable is associated with the type, including function, operator, macro, string, file and directory. It is allowed that aliases represent multiple types of variables as the subject and object of an architectural rule may be related to different kinds of source code components. By replacing complex and lengthy statements with an alias, the long lists representing the code artifacts like directories and function names can be avoided. This mechanism contributes to the enhanced readability of the program.

As revealed in the example in Listing 4.3.2, READ supports two kinds of statements —depend- and use-statement —to specify the rules. Additionally, a template block is provided to reuse the body of a rule definition. Following is a brief description of the syntax features of READ. A detailed discussion about them is given in the next subsections.

- **Depend-statement** describes the intended dependency between the files. A dependency is built when a source file includes a header file. The checker inspects all the included headers to determine whether they are the allowed dependencies of the given source file.
- **Use-statement** is related to the function calls in the program. If a function invokes in its definitions another function, then the caller function uses the callee function. The use-statement is used to specify the allowed pairs of caller and callee, while the unspecified pairs are regarded as violations by the checker.
- **Combinators** are used for both depend and use primitive to formulate rules. These combinators include:
  - **Negation** is used in depend-statement to indicate whether the dependent is allowed or not allowed to depend on the dependee. If it is allowed, further checking on the allowed public interface could be performed. Otherwise, it makes no sense to check the interface as the dependency is already not allowed. Negation is also used in use-statement with the combination of `ANY` and `EXIST`.
  - **Keywords** `ANY` and `EXIST` are used in use-statement to constrain the users. Combining with negation, four use cases are supported by the language: 'ANY user not use target' means any one of the specified users cannot use all the listed targets. 'ANY user use target' means any one of the specified users must use at least one of the listed targets. 'EXIST user not use target' means at least one of the specified users does not use all the listed targets. 'EXIST user use target' means at least one of the users uses at least one of the targets.
- **Template** block serves as a modularization part in DSL to improve code reusability. Each template consists of a template name, one or more parameters, and a template body. It works like defining

macro functions in C programming which a rule template is defined with the given parameters. The parameters are marked with a pair of angle brackets, indicating that they can be replaced by arguments in template invocation to formulate different rules. This block enables multiple similar rules to be declared in a compact way.

Listing 4.4.2 An example of READ instance

```
alias{
    DMA: Function [alloc, malloc, calloc], Operator [new, new[]], Macro [CREATE,
    ARRAY_ALLOCATE];
    startupPhase: Function [appInit, appPrepare, appStart];
    execPhase: Function [appRefresh, appRefreshFrgnd2, appExecute, appExecuteFrgnd2];
    iLayerD: Directory [
        D/samc/pub, D/interface/pub, D/configuration/pub, D/filter/pub,
    ];
}

// templates for rule definition
rule templates{
    dependency(ruleName, dependent, dependee, allowedInterfaces, allowedFormat){
        rule name: <ruleName>{
            depend{
                dependent: <dependent>
                depend on: <dependee>
                interface: <allowedInterfaces>
            }
            format of files to be checked: <allowedFormat>
        }
    }

    noUse(ruleName, sources, constraint, user, target){
        rule name: <ruleName>{
            check source files: <sources>
            constraint{
                <constraint> user in: <user>
                not use target: <target>
            }
        }
    }
}

check{
    dependency(Str [layerE_Rule], [Layer [layerE], Directory [layerD/pub]], NONE, NONE,
              Format [cpp]);
    noUsage(Str [sleep_Rule], Directory [layerE/application.cpp], ANY, ALL, Function
            [sleep]);

    rule name: layerC_Rule{
        depend{
            dependent: Layer [layerC]
            depend on: Layer [layerD, layerE]
            interface: iLayerD
        }
        format of files to be checked: Format [cpp]
    };

    rule name: dynamicMemory_Rule{
        check source files: ALL
        constraint{
            ANY user in: execPhase
            not use target: DMA
        }
    };
}
```

The checking block is responsible for the execution of rule checkings. The DSL supports two methods to execute the checking processes. One is calling the rule templates with arguments. The argument can be either an alias defined in the alias block or concrete variables. As shown in lines 36-39 and 51 of the instance, the DSL provides built-in keywords `NONE` and `ALL` to specify a rule that is applicable for none or any case. For example, we can specify all of the source files to be checked by using `ALL` in line 51. Similarly, the dependee in depend-statement can be set to `None` to indicate that the dependent is not allowed to depend on any layer. Besides, the language offers a list of predefined variables with the type `Layer` for the user to implement layer-dependent rules. The reason for this predefinition is that layer-based checkings require composed layer models, which entails a significant time investment in the model composition process. To mitigate the response time during the checking, we decided to construct the layer models during the model generation phase. Another way to check the rules is to directly define the rules in this block. The rules are automatically checked after definitions. This corresponds to lines 40-61 in the instance that `layerC_Rule` and `dynamicMemory_Rule` are checked subsequently after they are defined.

As mentioned in the description of the syntax features, the language enables the combination of negation and constraints to formulate the rules. Depend-statement allows the use of negation to indicate whether the dependency is permitted or not. In terms of Use-statement, as mentioned before, it requires the combination of negation and the keywords `ANY` and `EXIST` to specify the constraint. Once any fact is identified that does not satisfy the constraint, the checking function will return a false value, and the checker records a violation.

Having described the functionality of each block, we will now proceed to demonstrate the structure of the DSL syntax with UML class diagrams. Although in RASCAL language syntax is not grouped in classes, we find class diagram a suitable approach to model the relationships between syntax components. Specifically, each class corresponds to a syntax definition, which can be either a nonterminal or a terminal in a context-free grammar. Each syntax definition can be refined by the composition of other nonterminals and terminals, which can be illustrated by the aggregation notation in the class diagram. Additionally, RASCAL supports declaring alternatives for syntax definitions. This feature can be mimicked by the inheritance relationships between the blocks. Based on this mapping, we transform the textual syntax definition into a graphical representation. The class diagram of the entire DSL syntax is in Appendix C. As a part of the language, Figure 4.8 illustrates how the syntax definitions are derived from the root syntax. The language syntax diverges from the start point `Build`, forming the syntax definitions of the three blocks. Moreover, we distinguish the definition with different colors to indicate whether it is a nonterminal or terminal. For example, an alias statement is a pair of alias names and a list of values. Similar to `AiasPair`, `TemplateStatement` and `CheckStatement` are also associated with a subtree of syntax definitions, which will be discussed in the subsequent sections.

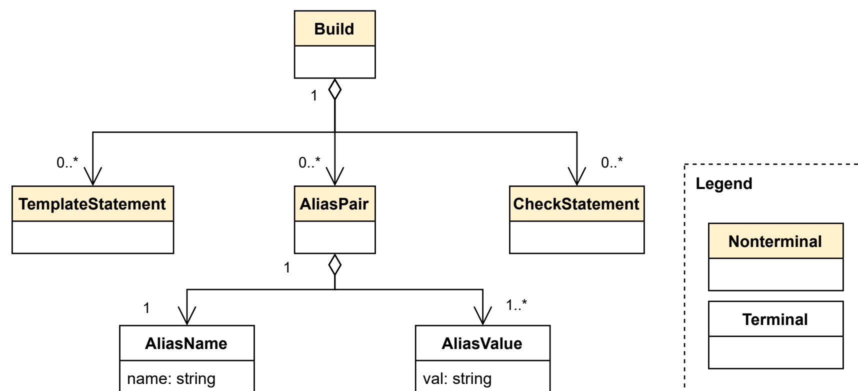


Figure 4.8: Syntax structure of the three blocks

Depend-statement

In the following two subsections, we demonstrate the syntax design for `CheckStatement`. This syntax definition is the core of the DSL where architectural rules are defined. The statement consists of two variants: depend- and use-statement. The former one is interpreted in this subsection, which is designed to implement RULE 1 and 2. Both rules constrain the dependency relationships between the layers. Thus, the headers included by the source files are the targets for investigation. To design the depend-statement, we first analyze the rule descriptions to extract the rule subject and object and determine the constraints with the required dependency. With the abstracted rule content, we then design the statement layout to formulate concrete rules.

As a result, as shown in lines 40-47 of Listing 4.4.2, the rule contains two components `dependent` and `depend on` specifying which components depend on which components. Here, we compose the definition of RULE 1 and 2 into one rule definition within the depend-statement. To define RULE 2, an optional `interface` statement is proposed to describe the directories of the allowed interfaces. Additionally, as both C/C++ source files and CMAKE files provide the information about which headers are required, the checks of RULE 1 and 2 can be conducted on two kinds of input files. Thus, the syntax contains a choice to indicate which types of source data to check.

In terms of syntax structure resolved from `CheckStatement`, Figure 4.9 and Listing 4.4.3 illustrate that depend-statement is composed of two nonterminals to describe allowed and disallowed dependency. These nonterminals have the same structure consisting of four terminals `RuleName`, `Dependent`, `Dependee` and `Format` to formulate the skeleton of the rule. Optionally, the nonterminal `InterfaceStatement` can be added to declare the allowed interface of the source files under investigation.

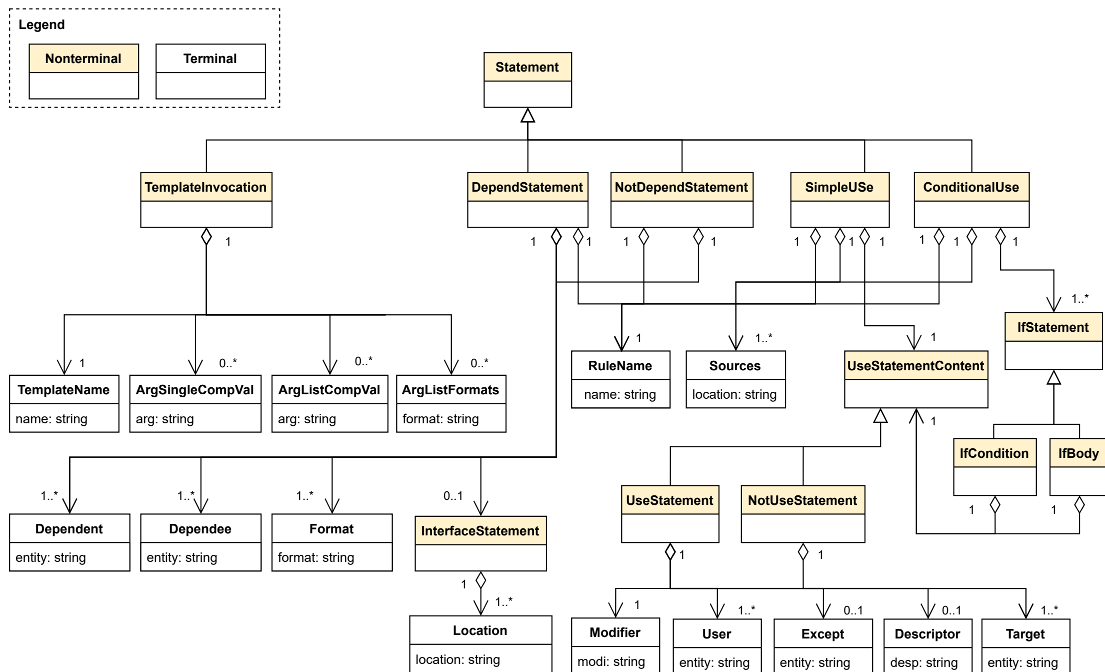


Figure 4.9: Definitions of depend- and use-statement in DSL syntax

Listing 4.4.3 Syntax of template invocation, use- and depend-statement

```
syntax Statement
= tplInvocation: TemplateInvocation
| dependDef: Depend
| useageDef: Use
;
syntax TemplateInvocation = tmpInvoke: TplName templateName "(" {ArgumentValue ","}* argVal
       !>> "," ")" ";" ;
syntax Depend = depend: "rule" "name" ":" Id ruleName "{" DependStatementContent content
       "format" "of" "files" "to" "be" "checked" ":" "Format" "[" {Format ","}+ formats !>>
       "," "]" "}" ";";
syntax DependStatementContent
= dependStatement: DependStatement
| notDependStatement: NotDependStatement
;
syntax DependStatement = dependStat: "depend" "{" "dependent" ":" {ComponentStat ","}+
       dependentCompStat !>> "," "depend" "on" ":" {ComponentStat ","}+ dependeeCompStat
       !>> "," InterfaceStat? interfaceStat "}";
syntax NotDependStatement = dependStat: "not" "depend" "{" "dependent" ":"
       {ComponentStat ","}+ dependentCompStat !>> "," "depend" "on" ":"
       {ComponentStat ","}+ dependeeCompStat !>> "," "}";
syntax Use
= simpleUse: "rule" "name" ":" Id ruleName "{" "check" "source" "files" ":"
  {ComponentStat ","}+ sources !>> "," "constraint" "{" UseStatementContent content "}" "}"
| conditionalUse: "rule" "name" ":" Id ruleName "{" "check" "source" "files" ":"
  {ComponentStat ","}+ sources !>> "," IfStatement+ statements "}"
;
syntax UseStatementContent
= useStatement: User user "use" "target" ":" Modifier? modifier {ComponentStat ","}+
  usedCompStat !>> ","
| notUseStatement: User user  "not" "use" "target" ":" Modifier? modifier
  {ComponentStat ","}+ usedCompStat !>> ","
;
syntax IfStatement = ifStat: "if" "(" IfCondition condition ")" "{" "constraint" "{"
       UseStatementContent content "}" "}" ;
syntax IfCondition = use: UseStatementContent content;
```

Use-statement

The use-statement is specifically designed to describe the constraints between the functions. Compared to depend-statement, which relies on the checking of directories of the layers, the use-statement possesses finer granularity and inspects the function calls or variable usages within the defined scope. Therefore, when a specific architectural rule is to be checked, we must first distinguish whether the rule constrains the relationships between the files or function calls. This distinguishment enables us to determine which syntax statement to construct the rule in the DSL instance.

The objective of the use-statement is to verify, for each specific source file, whether at least one of the target functions is subsequently called after calling a user function defined in this file. The design workflow of use-statement is the same as depend-statement in that we resolve the architectural rules suitable to be described with use-statement and design a generalized layout to present the rules. For example, RULE 3 and RULE 5 can be formulated with the use-statement. A concrete example is between lines 49 and 55 in Listing 4.4.2. This example shows a fundamental structure of use-statement layout consisting of a collection of source file directories, target functions and user functions that invoke other functions. To ensure unambiguous constraint definition, it is mandatory to specify the modifier `ANY` or `EXIST` before the statement of the user in the DSL. This modifier clarifies that the constraint is fulfilled either when all the using functions in the source file conform to the rule or when at least one user adheres to the rule.

Concerning syntax structure, Figure 4.9 and Listing 4.4.3 provide an elucidation of the use-statement, comprising two nonterminals denoting "use" or "not use" scenarios. Both of them are composed of five mandatory terminals, namely (`RuleName`, `Sources`, `User`, `Target` and `Modifier`). Additionally,

two optional terminals, `Descriptor` and `Except`, are designed to facilitate flexible filtration of user functions. The DSL also allows the incorporation of pre-conditions in a use-based rule by employing `IfStatement`. It consists of a condition and a body, enabling the reuse of the use-statement. This syntax is implemented to define the improved formulation of RULE 3, which will be discussed in Section 5.2.3.

Modularization

To enhance code reusability, we introduce a template block in language syntax that enables the user to define the architectural rules in a more modular and compact way. As illustrated in Figure 4.10, a rule template consists of a template name, a list of parameters and a body `RuleTplDefineStatement` representing the content of the rule template. The templates present a similar layout as the rules defined with use- and depend-statements, except that rule subjects and objects are replaced with parameters. Therefore, a `RuleTplDefineStatement` can be resolved into four types of template bodies based on the content of the desired rule. In this way, a rule can be implemented with template reference, which favors code reuse and improves maintainability.

By combining Figure 4.9 and Figure 4.10, we can find the template invocation is linked with the template definition through the unique template name. When the template name is matched, the parameters are mapped to the arguments based on the order they are declared. This matching process requires additional checking, which will be introduced in the next subsection.

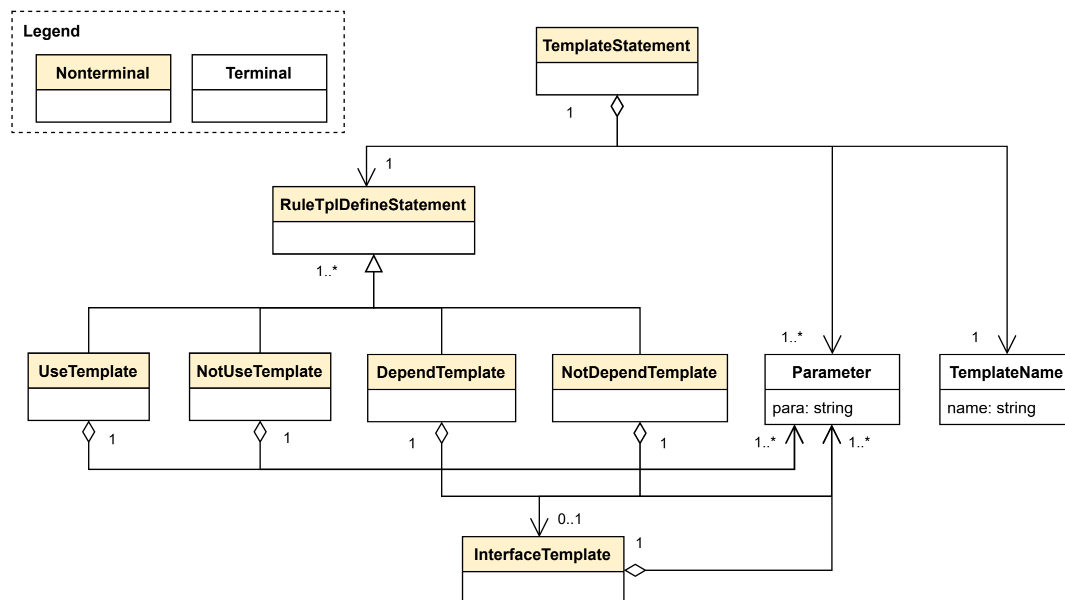


Figure 4.10: Definitions of template block in DSL syntax

Well-formedness Checking of READ Instance

A static semantics is regarded as well-formed if it conforms to the prescribed syntactic rules. If a READ specification is not well-formed, then its execution is undefined. Therefore, before checking the rules, the rule formulation must be well-formed. The goal of the checkings mentioned in this subsection is to restrict the input code and avoid any unnecessary overhead for solving ambiguity and type mismatch problems in the phase of violation detection. In terms of the `READ` instance we implement, the following rules are defined:

- No duplicated declarations of an alias name, template name and architectural rule name;

- No use of alias before they are defined;
- No use of rule template before they are defined;
- The Paths specified with the type `Directory` and `File` must exist in file system;
- The supported formats for depend-statement are limited to C, C++ and CMAKE;
- The inspection targets specified in the depend-statement must be within the types: `Layer`, `Directory` and `File`;
- The number of the arguments in template invocation must be equivalent to the number of the parameters declared in the template.

The well-formedness of the DSL is checked by investigating the CST of the inputted DSL instance. The first three rules are comparatively more straightforward to verify by using some features in TYPEPAL. For example, Listing 4.4.4 checks use-before-define violations in the instance. When parsing the `AliasPair` syntax, a collector defines the alias name to be a role `alias()`. In the case the alias name is called, the collector states this use case with the role of the alias. Through this define-use mechanism, TYPEPAL checks whether the required alias is already defined.

Listing 4.4.4 Implementation of no-use-before-define rule with TYPEPAL

```
void collect(current: (AliasPair) `<Id name> : <{ComponentFull ","}+ aliasComp> ;`,
             Collector c){
    c.define("<name>", aliasId(), current, defType(aliasType("<aliasComp>")));
    collect(aliasComp, c);
}

void collect(current: (ComponentAlias) `<Id compAlias>`, Collector c){
    c.use(compAlias, {aliasId()});
}

```

Compared to the first three rules, the implementation of the remained rules is independent of TYPEPAL. While matching the instance with syntax, we extract the nodes containing the statements related to directories and verify their existence. Similarly, we query the nodes representing depend-statement to analyze whether the specified depended components and source data formats comply with the rules. As for the last rule, we compare the parse tree of `TemplateStatement` and `TemplateInvocation` to ensure the number of arguments is identical to the number of parameters. This rule can also be implemented with TYPEPAL but at the time of development, we were learning this library and unaware of this feature.

4.4.3 Architecture Erosion Checking

In this section, we will explain the last but also the most significant component of the DSL tool, namely a checker for AE detection. The implemented checker is mainly for the checks on the three architectural rules mentioned in Section 4.3 but is also useful and extensible for the other rules. The checking process can be summarized as a comparison between the source code facts in M3 models and the rule constraints in the data structure generated during DSL instance parsing. As mentioned in Section 4.4.1, the source code is transformed into M3 models, which are stored as binary files on disk. Each checking process mentioned in this section requires reading the models to memory and performing the analysis on the models. In the following section, the design and implementation of the checker are elaborated on rule by rule.

Disallowed Layer Dependency

As stated in RULE 1, the dependencies between the layers are restricted to the explicitly specified directories. If a source file includes a header defined in another layer, then a dependency between the layer of the source file and the layer of the header is established. Therefore, to identify the violated dependency relationship, the included headers of the source files need to be checked. This goal can be realized either by directly checking the source files or analyzing the CMAKE files. In the following, we explain the implementation of both methods with the example shown in lines 40-47 in Listing 4.4.2. In this example, it is defined that the `C` layer can only depend on `D` and `E`. As the preparation work, a composed M3 model `dependentModel` for all the source files defined in `C` is constructed.

Algorithm 2 Checking of the disallowed layer dependencies

Input:
A {`isAllowed`} bool variable indicating the dependency is allowed or not allowed
A {`dependentPaths`} set containing the directories of all the source files specified for the dependent
A {`dependeePaths`} set containing the directories of all the source files specified for the dependee
A {`dependentModel`} M3 model representing the data of all the source files specified for the dependent
A {`allPaths`} set containing the directories for inspection

Output:
Violations represented in a binary relation with the scheme <source file in dependent, header file in dependee>

1: violations ← `emptyRelation`()
2: sourceToIncludedHeader ← dependentModel.requires
3: **for each** $< source, header >\in$ sourceToIncludedHeader **do**
4:     **if** isAllowed is $true$ **then**
5:         **if** $header \in$ (allPaths\\`union`(dependeePaths, dependentPath)) **and** $source \in$ allPaths **then**
6:             violations.add($source$, $header$)
7:         **end if**
8:     **else**
9:         **if** $header \in$ dependeePaths **and** $source \in$ allPaths **and** $header \in$ allPaths **then**
10:             violations.add($source$, $header$)
11:         **end if**
12:     **end if**
13: **end for**
14: **return** `violations`

If the source files are the focus of inspection, we query the `requires` field, which is a binary relation presenting a mapping from the source file to its includes from the composed M3 model to extract all the included headers of `C`. Subsequently, the violations are determined by identifying the intersection between the extracted headers and the headers defined in `D` and `E`. To implement this design, we have developed Algorithm 2, which offers a more generalized solution to check layer dependencies. This algorithm requires five input parameters. The first three are directly obtained from the parsing results of the DSL instance, while the last two data structures are generated prior to the checking process. The `allPaths` set restricts the scope of checking, as not all the source files and headers in the codebase need to be examined. The `isAllowed` indicates whether the rule defines an allowed or a disallowed dependency. If the dependency is allowed, then we filter out any included headers that are not part of the dependee and dependent. Conversely, if the dependency is not allowed, we inspect all the headers provided by the dependee.

If the CMAKE files are the source data for dependency checking, then a similar process is applied to the CMAKE files. The CMAKE files are categorized into different layers. The specified header files in each CMAKE file are compared against the set of disallowed directories to obtain the ones resulting in violations.

Non-public Interface Usage

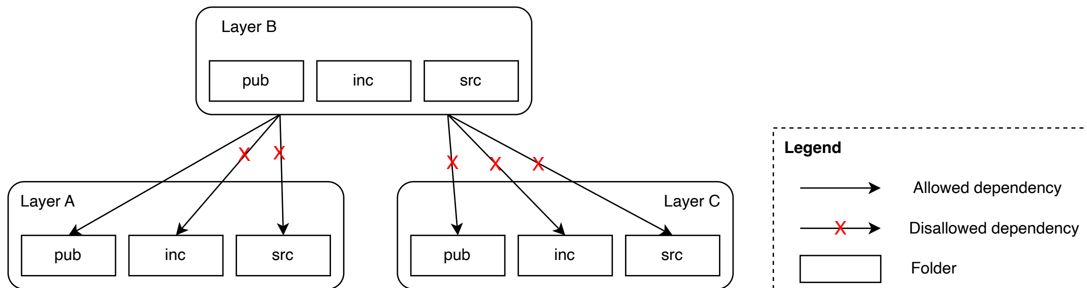


Figure 4.11: Folder structure of the code base

In the codebase, each layer comprises multiple modules organized with similar folder structures. As illustrated in Figure 4.11, C and C++ source files are grouped in `src` folder. As for header files, they are segregated into `inc` and `pub` based on their purpose. Headers indented for inclusion in the source files from the same module are stored in `inc`, while headers specific to external source files from other layers are placed in `pub`.

It is important to note that the examination of RULE 2 always follows RULE 1. When there are no violations of RULE 1, this allows us to have the assumption that disallowed dependencies would not exist in the system when checking the use of non-public interfaces. Considering Figure 4.11, if it is specified that Layer B is permitted to depend on Layer A but not on Layer C, then the source files in Layer B can only access the headers in the `pub` folder of Layer A and none headers from Layer C. With the assumption made above, we only need to focus on whether non-public interfaces are present in the set of included headers provided by Layer A.

Algorithm 3 Checking of the uses of non-public interface

Input:

A {`isAllowed`} bool variable indicating the dependency is allowed or not allowed
A {`interfaces`} set containing the directories of all the source files specified for the dependent
A {`dependeePaths`} set containing the directories of all the source files specified for the dependee
A {`dependentModel`} M3 model representing the data of all the source files specified for the dependent
A {`allPaths`} set containing the directories for inspection

Output:

Violations represented in a binary relation with the scheme <source file in dependent, header file in dependee>

1: violations ← `emptyRelation()`
2: sourceToIncludedHeader ← dependentModel.requires
3: **for each** $< source, header > \in$ sourceToIncludedHeader **do**
4: **if** isAllowed is $true$ **then**
5: **if** $header \in$ (allPaths \\`union`(interfaces, dependentPath)) and $source \in$ allPaths **then**
6: violations.add($source$, $header$)
7: **end if**
8: **else**
9: **if** $header \in$ interfaces **and** $header \in$ allPaths **and** $source \in$ allPaths **then**
10: violations.add($source$, $header$)
11: **end if**
12: **end if**
13: **end for**
14: **return** `violations`

Algorithm 3 provides a description of the implementation of this rule. It is almost the same as RULE 1 in that we query the `requires` relation of the dependent's M3 model to filter the include paths that do not serve as a public interface. In addition to source files, the algorithm can also be applied to CMAKE files to investigate the violations.

Dynamic Memory Allocation

As described in RULE 3, dynamic memory allocation is only allowed during the application startup phase. To clarify the implementation of the rule, we need to answer three questions:

- How is dynamic memory allocation implemented in C and C++?
- How are the startup and execution phases defined in the codebase?
- How to define the boundary after which the startup phase ends?

The first question is solved with the documentation that in C++ and C, dynamic memory is allocated by using the functions `alloc`, `malloc`, `calloc`, and the operator `new`. In the codebase, a macro, which serves as a wrapper for `alloc`, is defined to complete dynamic memory allocation. In Listing 4.4.2, an alias called `DMA` is defined to refer to the built-in operator and functions responsible for dynamic memory allocation. Therefore, the rule conformance checking is converted to explore the presence (and absence) of the calls on these functions or operators after the startup phase.

To answer the second question, we investigated the codebase and found that the phases of an application can be characterized by the specific functions called in this phase. At lines 3-5 in Listing 4.4.2, two aliases `startupPhase` and `execPhase` are specified to represent a list of functions invoked in the corresponding phase, respectively. All of these functions are declared as virtual functions in an interface of Layer E. The implementation of these virtual functions varies based on the requirements and expected functionalities of the application in different layers.

As for the third question, the operations in the startup phase involve three steps, starting from initialization to preparation and ending up with application start. Following the start, the application enters the execution phase by calling the execution and refresh functions. Therefore, we can check the functions defined in the execution phase whether they invoke the functions or use the operators that indicate dynamic memory allocation. Here, the method invocation includes not only the methods directly called in the definition of execution functions but also the transitively called ones.

Having determined the answers to the questions, we designed the rule checking by querying the data in the M3 models. As shown in Algorithm 4, our target is to extract all the function calls with the logical names starting with the `cpp+new` scheme or ending in the function name `malloc`, `alloc` and `calloc`. The extraction is performed on the call graph of each M3 model, in which the functions of the execution phase are defined. This is equivalent to checking all the source files where the virtual functions in `execPhase` are implemented to investigate whether the `DMA` functions are reachable. Practically, we construct the transitive closure of the codebase's call graph and query all reachable callee functions by inputting the `execPhase` functions. If any `DMA` function is present in the collected callees, it is considered a violation. This is the initial implementation to check RULE3, which formulates the foundation of this rule checking. A series of improvements are performed based on this implementation in the evaluation phase (Section 5.2.3) to enhance the accuracy of the results.

Algorithm 4 Checking of the dynamic memory allocation in the execution phase of the application

Input:

A {`model`} M3 model representing the source file for inspection

A {`userFunction`} set containing the name of the user functions defined in the execution phase, including `appRefresh`, `appRefreshFrgnd2`, `appExecute` and `appExecuteFrgnd2`

A {`targetFunction`} set containing the name of all functions for dynamic memory allocation, including `alloc`, `malloc` and `calloc`

A {`targetScheme`} set containing all the schemes corresponding to the operators for dynamic memory allocation, including `cpp+new`

A {`transitiveClosure`} relation representing the transitive closure of the call graph of the entire codebase, serving to obtain all the reachable callees given a caller function

Output:

Violations represented in a binary relation with the scheme <user function, target function>

1: violations ← `emptyRelation()`
2: callGraph ← model.callGraph
3: **for each** $< caller, _ > \in$ callGraph **do**
4:     **if** $caller$.file ∈ userFunction **then**
5:         candidates ← transitiveClosure[$caller$]
6:         **for each** $cCallee \in$ candidates **do**
7:             **if** $cCallee$.file ∈ targetFunction **or** $cCallee$.scheme ∈ targetScheme **then**
8:                 violations.add($caller$, $cCallee$)
9:             **end if**
10:         **end for**
11:     **end if**
12: **end for**
13: **return** `violations`

4.5 Deployment

After the completion of tool implementation, READ is deployed on a server. Then, it is executed from the command line automatically through Jenkins. Jenkins[2] is an automation server designed to handle task scheduling and automate program execution. In the case of READ, a new Jenkins job is created to orchestrate the READ's operations. The tool is configured with multiple build steps, including CMAKE parsing, model generation, READ parsing and rule checking, along with a target location for execution results. Based on this configuration, the Jenkins job is scheduled as a daily routine that runs overnight and delivers the outcome the next day. The results are written to the server's disk and can be accessed by the tool users through the network. Jenkins also offers a dashboard that presents the successful and failed job executions. This dashboard provides the user with a lucid overview of the progress of rule checking and the final status of the execution.

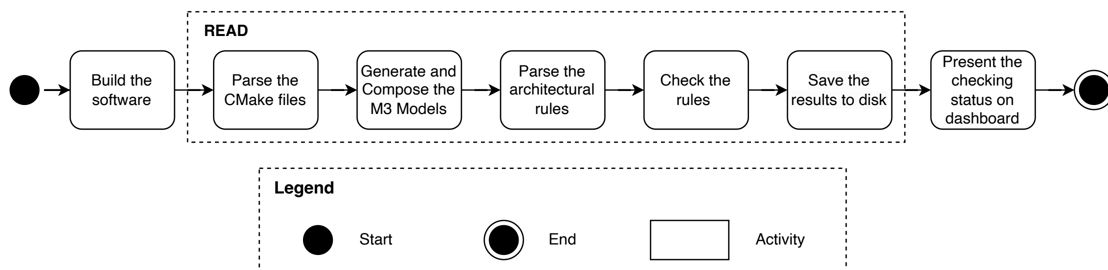


Figure 4.12: Execution steps of the tool on Jenkins

[2]https://www.jenkins.io/

Figure 4.12 illustrates how READ is executed as a Jenkins job. Before starting READ, the software must be built. This process includes downloading and installing the dependencies and generating some source files. This process is executed before the tool starts, as some generated files are required by the architectural layers. After the build, READ starts to process the CMAKE files and source code to generate the models and identify the violation by checking the architectural rules. By the end of the checking, the results are written to the disk of the server. As the last step, the execution status is reported on the Jenkins dashboard. If no error arises, then the dashboard reports a green sign with the execution time. Otherwise, the dashboard shows a red sign and an error message.

The execution results contain a collection of M3 models and the violations in the current codebase. As for the detected violations, they consistently align with the latest version in the central source code repository of the positioning software. Therefore, by comparing the violations between consecutive days, we can find the impact of the recent code changes on the architecture erosion in the codebase. Based on the currently detected violations, we can make an exception list, which allows the focus to be solely on newly arising rule violations attributed to the most recent changes in the source files.

4.6 Conclusions

Based on inspiration from literature ([19], [18], [9], [16], [34], [12], [27] and [35]) and concrete examples proposed in the meetings with Philips engineers, a DSL was designed to express 13 important architectural rules for the positioning software. Combining the DSL with a checker, a toolkit called READ was built that used Three high-priority rules to steer the design of the internals and to test the implementation on the actual code of the positioning software.

Additionally, the development of the tool verifies that the DSL can be implemented using Clair M3 models as input facts about the code and executing the semantics of the patterns against them to detect violations.

Moreover, READ can be deployed in a continuous integration environment like Jenkins to provide relatively early and automated feedback to the engineers. This feedback helps the engineers to gain knowledge about the newly introduced violations and to reduce the violations in future development.

Chapter 5

Evaluation and Results

To deliver a tool fulfilling the requirements mentioned in Section 4.2, it is important to evaluate the tool's performance. The evaluation enables us to have a qualified and quantified understanding of the tool and find the deficiencies in READ. To cope with deficiencies, improvement measures may be proposed to fix the tool components where unexpected behaviors or unsatisfied results arise. The evaluation commences with three fundamental evaluation questions. We propose the following evaluation questions, coupled with their respective motivations:

- **What is the quality of the generated M3 models?**
  To identify the rule violations in the architecture, instead of directly inspecting the source code, we build a M3 model for each source file as it is a more compact and efficient representation for information querying. As a result, all detected violations are facts filtered from the M3 models. However, this transformation is based on the assumption that the generated models can correctly and completely represent the data in the source code. In case the information related to the rules is incorrect or missing, our results are inaccurate. This deteriorates the fulfillment of REQ3. Therefore, it is necessary to evaluate the quality of the models to ensure the models, as the foundation of the subsequent checkings, are correct.

- **How accurate are the reported violations with respect to each architectural rule?**
  As mentioned in Section 1.2, the tool is expected ired to be reliable so that the false positives and false negatives in the results are minimized. Moreover, as stated in REQ3, a tool incapable of reporting correct results is unreliable and useless. Therefore, this evaluation question aims at exploring the preciseness of READ, proving that the collected violations are accurate and enabling the architect to maintain the architecture relying on the tool results.

- **How efficient is the DSL tool?**
  This evaluation question aims at analyzing READ whether it fulfills REQ5. We would like to reduce the overall execution time of READ as much as possible. In this way, less time is required for the users, and the user experience is improved. Therefore, in practice, the response time in each stage of the tool execution needs to be calculated and evaluated. Based on the evaluation results, measures may be proposed to enhance the execution efficiency and reduce the overall operation time of the tool.

The above three evaluation questions provide us with a guiding framework to design the experiments and assess the tool's performance. Depending on the objects on which the tests are conducted, these questions may be further divided into several subquestions. In the subsequent subsections, we will elaborate on the procedures and outcomes of each evaluation experiment.

5.1 Quality of the M3 Models

As mentioned in the last section, the M3 models are the foundation for rule checking, as the problematic or missing model data may eventually lead to incorrect results of architectural violations. Specifically, if data is missing, then the results are under-approximated as some facts in the source code are not represented in the models. This leads to more false negatives for positive rules and more false positives for negative rules. Conversely, if data is superfluous, then there would be more false positives for positive rules and more false negatives for negative rules. The quality of the model is determined during the model generation when the include paths are required by the Eclipse CDT parser. Here, quality can be represented by the number of names in the source code which are correctly resolved. If a name fails to be resolved, it will be marked as a problem in the model. In case this name is a fact violating a certain architectural rule, then it becomes a false negative in the results obtained from READ. Therefore, the evaluation in this section focuses on the problems that arose in the model generation phase. During the development and maintenance of the tool, we found two factors negatively influence the generation of the models: one is the quality of the CMAKE parser, which may fail to parse the CMake file and extract the include path. Another is the possible include paths required by the source files but erroneously collected by the CMAKE parser. Subsequently, we propose the evaluation question for each causal factor and describe the evaluation design as well as the results.

5.1.1 Parsing of the CMAKE Files

Evaluation-1: Syntax of the CMAKE Parser

Introduction and Evaluation Question
In this thesis, a CMAKE parser is implemented to parse the CMAKE files and extract the facts. Its main contribution is to obtain a collection of include paths for the source files specified in the CMAKE files. The parsing is based on the syntax that matches the code in the CMAKE files and transforms the code into CSTs. However, the syntax requires continuous maintenance, as a full and perfect CMake parser can not be made after one iteration of work. This is because we will only know whether and which include paths are missed by the CMAKE parser after the parser is executed. Therefore, we have to start with an imperfect prototype, which needs multiple rounds of execution and improvement to become more complete. Besides, the usage of new grammar or the definition of new functions may be introduced into the CMAKE files during the codebase evolution. If the updates of the syntax fall behind the changes in the CMAKE files, the paring may fail, leading to the termination of the tool operation. To avoid severe termination, the syntax should be sufficiently general such that it still works for the future evolution of the CMAKE files. Therefore, we propose an evaluation question: How general is the grammar for parsing CMAKE files we use in READ?

Evaluation Method
The generality of the syntax can be characterized by the number of times when the CMAKE parser fails in parsing the CMAKE files during the evaluation of the codebase. The test set is composed of all of the CMAKE files in the codebase. Specifically, the CMAKE syntax was implemented in the first month of the assignment. This implementation was capable of successfully parsing all the CMAKE files in the codebase. Then, for each subsequent new version of the codebase, we tested whether the same implementation of the parser can still parse all the CMAKE codes, especially the newly added ones. The added codes include the codes not present in the previous CMAKE files and the small modifications on the existing codes. We count the number of lines of code leading to failed parsing as the metric to evaluate the quality of the parser.

Evaluation Results
Table 5.1 demonstrates the changes in the codebase with the number of failures. During nearly five months of development, the up-to-date codebase contains 619 CMAKE files. Compared to the version based on which our CMAKE parser was implemented, 19 new files are added to the codebase. The modification in all the CMAKE files contributes to 3208 lines of added code. The number of removed codes is not counted as they have no impact on the operation of the parser. The result is that the CMAKE parser failed three times due to parsing errors. One error is attributed to the changed signature of a

custom macro function. The other two problems are because the implemented syntax cannot match the parameters of some built-in macro functions. For example, `source_group` is a CMAKE function that organizes a group of source files. This function offers optional parameters that vary depending on the specific use scenario. Macro functions are hard-wired into the grammar since otherwise, parsing CMake files would be Turing complete, as knowing what the input syntax is requires the execution of the macro function. Thus, our CMAKE parser supports only a subset of all parameter combinations and fails when an unimplemented parameter list is requested.

Table 5.1: Performace of the CMAKE parser during software evolution

#(CMAKE files)	#(Added CMAKE files)	Lines of added code	Lines of code failed to parse
619	19	3208	3

Conclusions and Possible Threats to Validity
From the aforementioned experience, we can find that our CMAKE parser is capable of dealing with most parsing tasks while the software evolves while 0.09% of the lines of added code lead to parsing failures. This is because, on the one hand, no radical change has been made in the CMakes files during the five-month development. On the other hand, the syntax we developed is generic in most cases, even though changes are introduced into the files.

The fail cases show that the CMAKE parser is vulnerable to the changed or newly implemented macro functions, as we cannot expect what those functions will be like. To mitigate this negative impact, we generalized the syntax implementation of the built-in macro functions used in the codebase. As for the issues caused by custom functions, they can only be fixed after a parsing error arises.

5.1.2 Evaluation of Include Path Extraction

The DSL tool we implemented requires two kinds of source data: C/C++ code and architectural rules. The CMAKE parser is the entry point to process the source data and prepare for the subsequent checkings. It is designed to collect the include paths of each source file by parsing the CMAKE files of the codebase. These include paths are essential as they determine the accuracy of the model with respect to the syntax, static semantics and dynamic semantics of the code. When the include paths are missing, it leads to the failed name resolution of the source code. This occurs because the Eclipse CDT parser is unable to find the headers where these names are declared and defined. These failures are marked as `problem` in the M3 model. That means the names in the source code cannot be correctly resolved and are not presented in the models. As our analysis is based on the M3 models instead of the source code, unresolved names make the models less informative and useful in architectural violation checking.

Evaluation-2: Impact of not found include paths on violation checking

Introduction and Evaluation Question
The first evaluation question related to include paths is how impactful the include paths are on the results of the rule checking. The motivation for this question is that we would like to know what the most severe problems the not found include paths can cause with respect to the accuracy of violation identification. As the ultimate goal of the tool is to detect AE in the codebase, if the detection is influenced significantly by the absent include paths, then measures must be proposed to solve the problems.

Evaluation Method
To perform the evaluation test, we collect all the source files in which AE is identified with respect to the three architectural rules checked in this assignment. The source files are grouped by architectural rules. Then, for each group, we removed an include path from the collection, which is used to generate the models of the collected source files. The criterion to select the include path is we chose the include path, which leads to the header files that are most frequently present in the violation results as we inspect the worst case when a path is missing. After the removal, we generated the new models and

verified whether the violation could still be detected. Eventually, we calculated the difference between before and after the removal as the metrics to indicate the impact of the not found include paths.

Evaluation Results

Having performed evaluation tests, we obtained the results that in the ideal case, a not found include path has no impact on the final results of the identified architecture erosion, as the artifacts in the headers led from the include paths are irrelevant to the architectural rule. In comparison, in the worst case, if we remove a specific include path for all the source files during the model generation, the number of detected violations is decreased, as revealed by Table 5.2.

The evaluation result is specific to the architectural rule as the violation results of some rules may be derived from a common header such that if its include path is removed, then the detected violations decrease dramatically. This is the case for RULE 2, which checks the use of non-public interfaces between the layers. As most of the violations are due to the invalid inclusion of a header, if the include path of this header is removed during model generation, then 60.94% of the results disappear. Similarly, the number of violations of RULE 3, which restricts the use of dynamic memory allocation, is reduced by 90.48% if the include path of a header responsible for creating instances is removed.

Table 5.2: The worst case results of the identified violations after the removal of an include path

	RULE 1 Layer Dependency	RULE 2 Public Interface	RULE 3 Dynamic Memory Allocation
#(Violations before removal)	32	94	21
#(Violations after removal)	25	37	2
Reduction (%)	21.88	60.64	90.48

Conclusions and Possible Threats to Validity

This experiment reveals that a missing include header can significantly deteriorate the usability and reliability of the checker. Without an accurate include path, READ does not function at all since, in the worst case, 9 out of 10 true positives are lost, and the false negative rate increases dramatically. Thus, the number of not found include paths must be minimized.

Evaluation-3: Not found include path during model generation

Introduction and Evaluation Question

During the phase of model generation, ClaiR proposes the messages when a header file required by a source file is not found due to an absent include path. In this section, we use these messages to evaluate the quality of the models.

First, we discuss four causes of the not found include paths as follows:

- **Non-existent headers in codebase**: These headers are the files required by the source files but absent from the codebase. The positioning software is implemented for both Windows and VxWorks systems. VxWorks[1] is a real-time operating system used by Philips positioning software in the production environment. However, some header files only exist and are used in a specific OS. In the context of this assignment, the codebase is installed and built in Windows. Therefore, the headers of VxWorks cannot be found in the file system. Even though in the source code some VxWorks headers are conditionally included, the CDT parser analyzes the source code statically and requires all the header files specified with `#include` to be provided without checking whether the condition is fulfilled. Another case is due to "dead inclusion" — the header file is deprecated and removed from the codebase, but the inclusion is not deleted from the source file. As a result, warnings are generated indicating that include paths are missing during the phase of model generation.

[1] https://www.windriver.com/products/vxworks

- **Incomplete default include paths**: Not all source files are collected by the CMake parser. Some source files are used to create Visual Studio Solution[2] without being specified in CMake files. As a result, we do not have a list of include paths for these source files. To tackle this issue, we create a list of default include paths for the not found source files. This default list requires consistent maintenance as the positioning software evolves. In case the list is out-of-date, the newly added headers are marked as not found include paths during model generation.

- **Bugs**: The CMake parser may contain bugs that the header files failed to be added to include path list, even though the headers are specified in the CMake files. This is usually caused by dynamic path concatenation when the path of the header files in CMake is presented in a complicated format with multiple variables and strings.

- **Not found headers in CMake**: While the source file is specified in the CMake, its required headers may not be fully detectable. In this case, our CMake parser builds an incomplete list of include paths for the source file. However, our CMake parser does not parse the definition of macro functions because the function body can be too complicated to maintain the semantics after parsing. It is possible that some unknown mechanism in CMake is applied to include the headers but is not covered by our parser. For example, when designing the CMake parser, we assumed that the paths of all the header files are explicitly specified in the CMake text by checking the keyword `DIRS`. However, further investigation enables us to find that some source headers are included implicitly by calling a custom macro function. This function is newly introduced during the evaluation of the CMake files and include header files with dynamic path concatenation. As a result, when some include paths are specified in these functions, our parser is incapable of detecting them.

Based on the causal factors described above, we propose for each causal factor an evaluation question:

- How impactful are the non-existent headers in relation to not found include paths?
- How impactful is the incomplete default include paths in relation to not found include paths?
- How impactful are the bugs in relation to not found include paths?
- How impactful are the not found headers in CMake in relation to not found include paths?

The motivation of this evaluation is to categorize the records of the absent include paths based on the causing factors in each source folder of the positioning software. Then, for each category, we may develop a generic solution to reduce the presence of absent include paths and improve the model quality. As we apply the same evaluation method by labeling the not found include paths with the causal factors, these four evaluation questions are merged into one evaluation experiment.

Additionally, as the number of not found include paths is categorized by the source folder of the positioning software, we assume that the number of not found include paths is positively related to the number of source files in the folder. Therefore, it is necessary to normalize the number of not found include paths with respect to the number of source files in the folder such that we can know which causal factor is most impactful.

Evaluation Method

The evaluation is based on the output log prompted in Rascal terminal after the ClaiR function `createM3FromCppFile` or `createM3FromCFile` is called. Specifically, the include paths for each source file are collected by the CMake parser and forwarded to the model generation functions. While generating the models, Rascal prints the output log about the generation information. We then parsed the log with the keyword 'not found' to extract all the not found headers and categorized the results with the four causes. The experiment was conducted for all the source files except for the legacy code. The source files are grouped by folders. Each layer or module corresponds to a folder in the codebase.

[2]https://learn.microsoft.com/en-us/visualstudio/extensibility/internals/solution-dot-sln-file?view=vs-2022

Additionally, two other folders, `Core` and `Build`, store the source files that have not been assigned to a specific folder yet and the source files generated during the software build, respectively.

As for the normalization of the number of not found include paths with respect to the number of source files, we calculate the ratio between the number of the not found include paths and the number of source files in each folder. It is essential to note that a folder is not equivalent to a layer in the architecture. A layer comprises the source files in the folder with the same name as the layer and the source files from the Core folder.

Evaluation Results

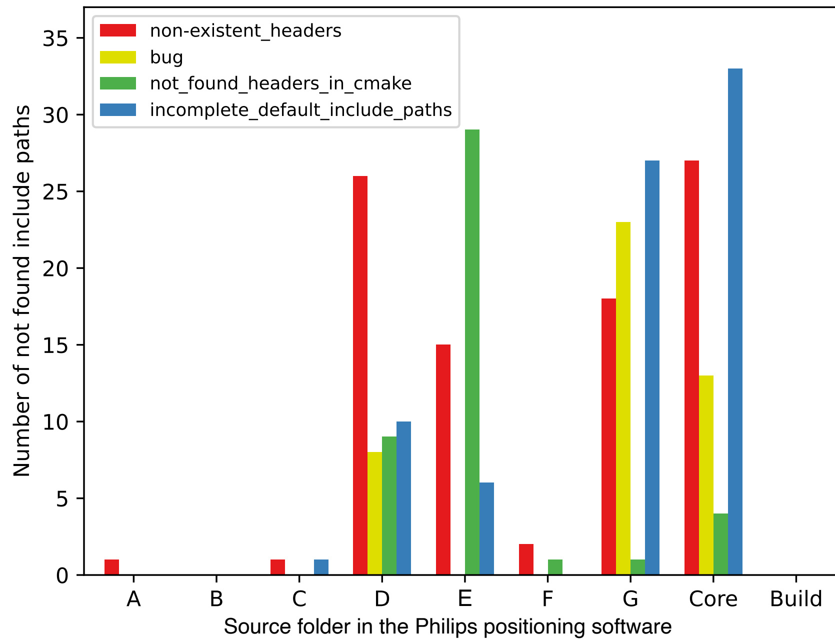


Figure 5.1: Number of not found headers of each folder of source files categorized by four causing factors

Figure 5.1 illustrates the number of not found include paths categorized by causal factors in each folder. It is clear that most not found headers are collected during the model generation for the source files in folder `D`, `E`, `G`, and `Core`. The main reason for the higher value of not found headers in these four modules is they contain more source files, which account for nearly 90% of the total files. It is clear in the figure that incomplete default include paths is the dominant factor leading to not found include paths for `G` and `Core`. In comparison, non-existent headers contribute to half of the not found issues in `D`. Although the non-existent headers account for more than one-third of the not found include paths, we consider it has no impact on the operation of the software. The reason is non-existent headers are either dead inclusions or headers of VxWorks that do not influence the execution of the system. Besides, the software is built successfully even though these headers are required but do not exist. Therefore, we believe the non-existent headers can be excluded from our investigation.

Table 5.3: Comparison of source files with not found headers to the total number of source files categorized by folder

Folder	A	B	C	D	E	F	G	Core	Build
#(Filtered sources with not found include paths)	1	0	1	78	36	3	88	58	0
#(Total source files)	85	0	51	924	783	236	535	1498	120
Percentage (%)	1.18	0.00	1.96	8.44	4.60	1.27	16.45	3.87	0.00

Moreover, Table 5.3 compares the number of source files with at least one header missed during model generation to the total number of source files. Here, the source files with not found headers are filtered as the results resulting from the non-existent headers are excluded. Each source file may have more than one absent header, and each not found header may appear during the model generation of more than one source file. We can find that the values of B consistently remain at zero since this folder contains no source file. Comparing the number of filtered source files to the total number, the folder G possesses the highest percentages of the impacted source files (16.45%), while the percentages of the source files with absent include paths are all below 9%.

Conclusions and Possible Threats to Validity
We can draw a conclusion based on Figure 5.1 that the most not found include paths are in folder D, E, G and Core. The most impactful causal factors in these folders are non-existent headers and incomplete default include paths.

From Table 5.3, we can conclude that the proportion of the not found headers is relatively low. Therefore, even though Evaluation-2 shows not found include path can be a severe problem in identifying architectural violations, considering the scale of the entire codebase and low ratio of the not found include paths, we know that the generated M3 models already present most of the information in the source codes. Thus, these models are reliable for the subsequent rule checking. As mentioned before, not found headers deteriorate the name resolution of the source code, leading to more 'problems' marked in the M3 models and reducing the usability of READ. A low proportion of not found include paths means limited negative impacts on the usability and reliability of the tool.

Evaluation-4: Not found include path during the model generation after improvement

Introduction and Evaluation Question
The last evaluation test has shown that not found include paths exist during the model generation phase, especially for the source files from the folder D, E, G and Core. To mitigate the failed name resolution and improve the usability of the M3 models, measures must be taken to reduce the number of not found include paths. This evaluation is about how the not found include paths can be reduced and what impacts the reduction has.

Evaluation Method
To mitigate the influence of the not found headers, measures are proposed based on the category of the causing factors as follows:

- **Non-existent headers in codebase**: No measure is proposed for this cause as the required headers are absent in the codebase.
- **Incomplete default include paths**: We update the list to make it aligned with the newest codebase. The paths are added iteratively as some new not found path warnings may arise after one include path is added due to transitive inclusion.
- **Bugs**: The bugs are attributed to the problems in the path concatenation of the parser or incorrect resolution of the CMake built-in variable PROJECT_BINARY_DIR. Based on these causes, the bugs are solved.
- **Not found headers in CMake**: We investigate the possible approaches that include paths that are introduced but not captured by the parser. For example, checking the implementation of the macro functions to find whether include paths are specified in the function body.

After applying these measures, we performed the same evaluation as the last evaluation. The intuition of this evaluation is to explore to which degree the missed include paths are solved and what factors leading to not found include paths require more effort for enhancement. We used the same test sets and method in this evaluation and obtained the results revealed in Figure 5.2.

Evaluation Results
It is evident that all the not include paths caused by the bugs or incomplete lists of default paths are

solved after the improvements. All the records due to non-existent headers illustrated in Figure 5.1 remain unchanged as no measure is performed for it. If the non-existent headers are not considered, the total number of absent headers is decreased from 165 to 25. Moreover, the number of absent headers resulting from not found headers in CMAKE is reduced by 43.18%. The most absent cases are still in the folder E. As mentioned in Evaluation-3 that not found headers in CMAKE can be due to some unknown mechanism introducing the include paths. It is possible that even though we have discovered some of them, it still other mechanisms that we are not aware of and should be implemented in future work.

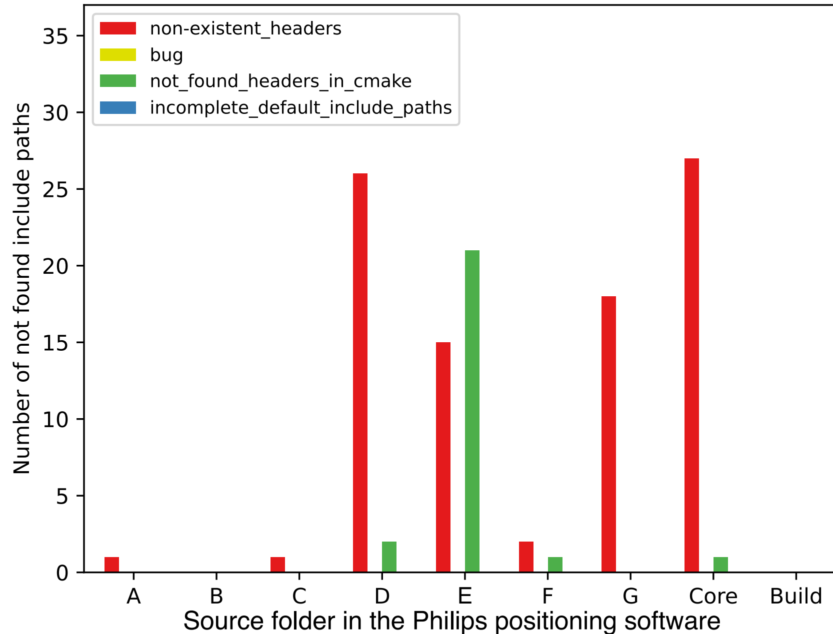


Figure 5.2: Number of not found headers of each folder after the improvements categorized by four causing factors

Conclusions and Possible Threats to Validity
From this evaluation, we can conclude that the improvement measures are effective in reducing the number of the not found include paths, contributing to higher quality of the generated M3 models. Additionally, more investigation needs to be conducted for the case that the required headers exist in the codebase but are not collected from the `CMake` files.

5.2 Implementation of Architectural Rules

READ is designed and implemented to check at least three architectural rules. In the following subsections, we will evaluate the checking processes of each rule and discuss the final results obtained after multiple iterations of improvement.

5.2.1 Disallowed Layer Dependency

This rule specifies which layers a given layer can depend on. The checking is based on the header files included by the source files of the given layer. If a required header is from a disallowed layer, it is counted as one violation. In this way, we obtain the initial lists of violations, which are demonstrated in the first row of Table 5.4. It is clear that layer `A` accounts for more than half of all the detected violations, followed by layer `D` and module `G`.

Table 5.4: Number of layer dependency violations in each layer or module before and after improvements

	A	B	C	D	E	F	G	**sum**
#(initially detected violations)	784	1	48	196	26	0	288	1343
#(violations in production code)	10	1	0	31	1	0	288	331
#(violations in production code of VxWorks)	0	1	0	31	0	0	1	**33**

Evaluation-5: Evaluation of the initial implementation of RULE 1

Introduction and Evaluation Question

With the initial implementation, we propose an evaluation question of how accurate the results are with respect to the violated dependencies between the two layers. The motivation is we must ensure that the obtained results are precise so that the architect can manage the architecture based on these results. The inaccuracy of the checker reduces the reliability and usability of the tool. We regard violation detection as a binary classification problem. The inaccuracy is derived from the false positive and false negative in the results. In the context of RULE 1, false negative means the undetected dependency violation between the layers. In terms of the false positive, it means the checker considers some results as a fact of violation between the layers while they are actually not. Both cases can be caused by defects in the implementation of the checker and an incorrect understanding of the architectural rule.

Evaluation Method

The accuracy of the results is characterized by the ratio of false negatives and false positives in the results. The following results presentation only focuses on false negatives because we do not compare the results with the ground truth in this evaluation test. Therefore, we have no idea whether the results contain false negatives. As for false postives, they can be obtained by comparing the results before and after the first improvement. To explore the problems in the results, we inspected all the collected violations, which are a collection of pairs from the source file to a disallowed header. The inspection lets us find that some source files in the result set are implemented for tests. As the architectural rule is specified for the production code, the results related to the source files specific to the test code are false positives. Based on this fact, we designed an evaluation experiment that takes all the pairs of the source file and the disallowed header as a test set and checks if the source file is part of the test code. A source file is labeled as test code during the parsing of the CMake files when it is only used to configure the test libraries. Therefore, the false positives resulting from the test code are eliminated by verifying whether the source file is labeled as test code.

Evaluation Results

As shown in Table 5.4, the total number of violations is reduced by 76% to 331. Except for module `G`, the number of violations originating from all the other layers is decreased. As for module `G`, although it is the codebase's test module, some of its source files are required in production. Therefore, we can still find violations from this module even though the source files for the test code are excluded.

Conclusions and Possible Threats to Validity

The initial improvement for the implementation of RULE 1 contains a large number of false positives due to the test code. It is vital to remove this part of the code from the research scope as we cannot know whether the results reported by READ are real violations or not. In other words, a high number of false positives reduces the confidence in using READ, and thus, must be mitigated.

Evaluation-6: Evaluation of the first improved implementation of RULE 1

Introduction and Evaluation Question

A further evaluation based on the above filtration is to compare the obtained violations with the ground truth. The ground truth is from another tool developed by the architect for daily architecture monitor-

ing of the same architectural rule. With the ground truth, we can have a knowledge of false positives and false negatives in the results after the first improvement. This knowledge promotes the evaluation question: How close are our results to the ground truth?

Evaluation Method
The design of the evaluation is straightforward in that we converted the ground truths to the same format as the representation of the results returned by our checker. In this way, we got two sets of results, with one containing the ground truth and another containing the detected results. Then, we computed the difference between the two sets.

Evaluation Results
As a result, we have about 300 hundreds more violations than the ground truths. We ascertained that the architect's tool only covers part of the codebase used for VxWorks, leading to fewer violations than our results. This does not mean that our results contain false positives. The reason for the difference is the constraints based on which the two tools are implemented are different. The positioning software is designed for both Windows and VxWorks. The implementation of the architect's tool focuses only on the code related to VxWorks, while our tool checks all the code. The architect's design is because most of the development work is made for VxWorks currently. As the goal is to prevent the new changes in the source code from violating the architectural rules, we focus more on the part of the codebase that is actively modified. Besides, as the entire codebase may contain many violations, starting from a part of the codebase is a more feasible approach to analyze the erosions in the architecture. Based on these two reasons, the checks of the architect's tool are only conducted on the source code related to VxWorks.

Conclusions and Possible Threats to Validity
The first improvement for the implementation of RULE 1 contributes to the removal of a large number of false positives by excluding the source files for the test code. This improvement is proposed because the initial implementation is imprecise and also includes the test code. However, the results that deviated from the first improvement may still contain false positives. This requires further improvements in the tool implementation to align the final results with the ground truths.

Evaluation-7: Evaluation of the second improved implementation of RULE 1

Introduction and Evaluation Question
The last evaluation shows the gap between the first improvement of the tool and the architect's tool because our implementation covers both Windows and VxWorks, while the architect's tool checks only the VxWorks part. To align our checking with the architect's tool design, we added one more filter to extract the violations from the source files used for VxWorks. This filtration is implemented with the CMake parser. When configuring the libraries, CMake requires a flag to be specified for different compilation configurations. We collect all the libraries configured with the flag `OBJ_VXWORKS` and regard the source files in these libraries as designed for VxWorks. Ultimately, we obtained a subset of the violations collected after the first improvement. We will discuss these violations in the subsequent sections.

As the alignment with the architect's tool is completed, we decided to compare the results with the ground truth again because we need to investigate whether our design of the checker leads to correct results when checking layer dependency. Due to this reason, we designed the third evaluation to verify the correctness of the implementation.

Evaluation Method
The design of the evaluation is the same as the last evaluation in that we compared our results with the ground truth. The test set is the violations collected by our checker.

Evaluation Results
The evaluation result is our checking results are the same as the outcomes from the ground truths. By comparing the content of the 33 detected violations, we know that these violations are exactly the same as the ground truths. That means no false positives and false negatives remain in the result set constructed by this implementation. The checking results are presented in the last line in Table 5.4.

This verifies our tool fulfills the requirement of detecting disallowed layer dependency and providing accurate information on rule violation.

Table 5.6: Confusion matrix for the first implementation of RULE 1

Total N/A	**Predicted Positives** 1343	**Predicted Negatives** N/A
Actual Positives 33	33	0
Actual Negatives N/A	1310	N/A

Additionally, we construct a confusion matrix (Table 5.6) to illustrate the changes in the detected violations before and after the improvements of the checking implementation. In the matrix, the term 'positive' denotes the cases where RULE 1 is violated, while 'negative' indicates the implementation conforms to the rule. It is evident that the result of the first implementation contains 1310 false positives, while the eventual 33 layer dependency violations are the true positives. These true positives are identified in the last implementation, wherein violations originating from test code and Windows-specific code are removed. As these 33 violations are also the same results that deviated from the architect's tool, the number of false negatives is 0. Besides, as the goal of the checker is to identify all the rule violations in positioning software, which are presented as positive results in the confusion matrix, obtaining the number of true negatives by counting the number of header inclusions is out of the scope of the implementation of the checker. That is why the number of true negatives is marked as 'N/A' in the matrix.

Conclusions and Possible Threats to Validity

After two iterations of improvements, we can conclude that the implementation of RULE 1 meets REQ3 to obtain the correct number of violations in Philips' positioning software.

It is important to note that the construction of the confusion matrix is specific to the definition of ground truths. Table 5.6 is created based on the ground truths from the architect's tool, which categorizes the violations resulting from the production code of the Windows part as false positives, given the architect's exclusive investigation on the VxWorks part. If the definition of the ground truths covers the production code of both OS, then the number of false positives will be reduced.

To summarize, we adopted two improvements in the implementation of checking disallowed layer dependency. We have the first improvement because our initial implementation, covering both production and test code, is imprecise. The second improvement tackles the problems caused by the different rule definitions between our and the architect's tool design. These improvements led our tool to be aligned with the architect's tool, proving the correctness and accuracy of the implementation of the rule checking.

Checking Result Discussion

Figure 5.3 shows the layer dependency violations after two iterations of improvement. We used the same layout as Figure 2.1 to indicate the hierarchical order between the layers and modules. This figure reveals that nearly all of the violations exist in layer `D` that illegally requires the headers defined in module `G`. Within layer `D`, most violations are from the source files implementing the motion control interfaces. Additionally, three violations occur when the header files in the service module `F` are included by the disallowed layers. These violations have been verified by the architect to be true violations existing in the codebase. Some of them are caused by the source files from the `Core` folder. As mentioned in Section 2.2, this folder contains the source files whose layer is known but is not assigned to the corresponding folder designed for the layer. In the future, the architect will fix these source files by moving them to the folder they should belong to. In this way, the detected violations are expected to be removed.

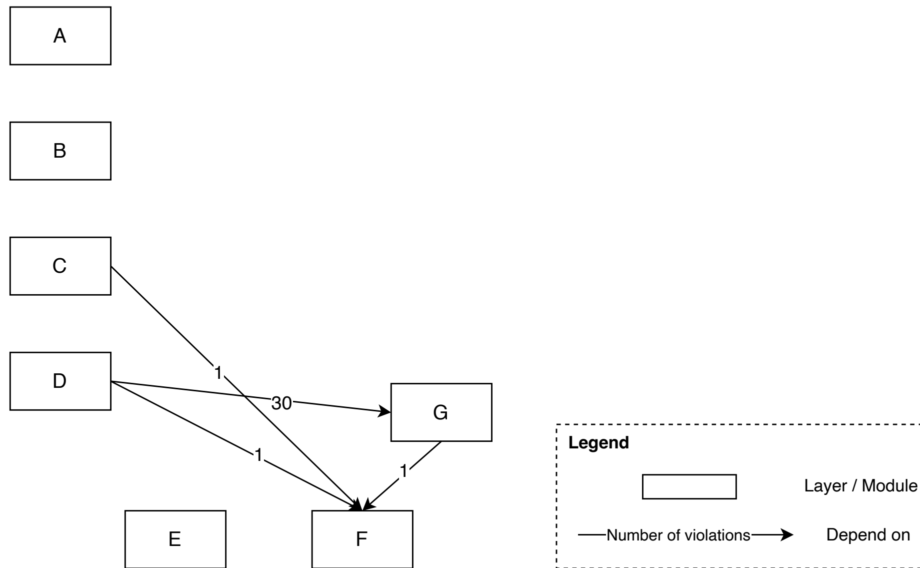


Figure 5.3: Violations of disallowed dependency between layers or modules

5.2.2 Non-public Interface Usage

Evaluation-8: Evaluation of the implementations of RULE 2

Introduction and Evaluation Question
As the mechanism to check the violations of RULE 1 and RULE 2 is the same, the evaluation concerning RULE 2 regulating the utilization of the specified interfaces between the layers is the same as RULE 1. We also completed two improvements regarding the test code and the VxWorks part of the codebase to align our results with the ground truths. After that, we propose the evaluation question: How is the accuracy of the results of each implementation for RULE 2?

Evaluation Method
We used the same method as the evaluation of RULE 1 to evaluate the accuracy of the results of RULE 2. Specifically, we compare the results before and after each improvement and also with the ground truths from the architect's tool to obtain the number of false positives and false negatives. The first two evaluations contribute to two iterations of improvement to filter the violations resulting from the production code used for VxWorks. The test set of both evaluations is composed of the violations collected by the checker.

Evaluation Results
As shown in Table 5.8, the first improvement enables us to remove nearly 1500 false positives from the result set, with the most false positives from layer `A` and module `G`. After the second improvement, the total number of violations was reduced to 94. These two improvements reveal that most of the violations collected in the initial implementation are false positives from the test code. This is especially the case for module `G` as it is the test module of the codebase. The final set of violations is the same as the results obtained from the architect's tool, verifying that the implementation of this architectural rule is precise and reliable.

Similar to RULE 1, a confusion matrix (Table 5.10) is built for RULE 2 to present the true and false positive results of the initial implementation. We use the same definition for positives and negatives as the matrix of RULE 1. Again, the ground truth is from the architect tool targeting the violations caused by the production in the VxWorks part of the codebase. It is clear that the results from the first implementation contain a large number of false positives due to the test code. After the evaluation process, the number of true positives decreases to 94 after the second. The negative results are unknown as the checkings only investigate the violations in the codebase.

Table 5.8: Number of non-public interface violations in each layer or module after improvements

	A	B	C	D	E	F	G	**sum**
#(initially detected violations)	795	2	204	0	0	91	1408	2500
#(violations in production code)	10	2	15	0	0	63	835	925
#(violations in production code of VxWorks)	0	2	15	0	0	63	14	**94**

Table 5.10: Confusion matrix for the first implementation of RULE 2

Total N/A	**Predicted Positives** 2500	**Predicted Negatives** N/A
Actual Positives 94	94	0
Actual Negatives N/A	2406	N/A

Conclusions and Possible Threats to Validity
Similar to the evaluation of RULE 1, we can conclude that the implementation of RULE 2 meets REQ3 because the final collected violations are the same as the ground truths provided by the architect.

The results are also under threat that they are dependent on the definition of the ground truth. If the ground truth contains not only the implementation of the positioning software for Windows OS, then the number of false positives will be reduced.

Checking Result Discussion

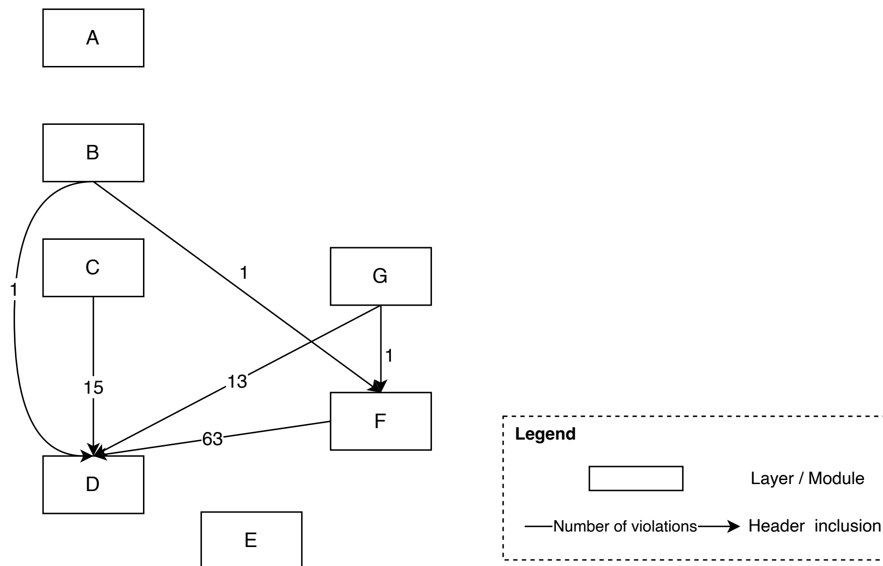


Figure 5.4: Violations of non-public interface usage between layers or modules

Figure 5.4 is a graphical representation of the inclusion of non-public interfaces between the layers.

98% of the violations are because of the invalid inclusion of the headers from layer D. Specifically, 67% of the violations are from the module F that includes the disallowed headers from layer D. All of these violations are caused by the same source file in module F. Moreover, the headers in D are also frequently invalidly included by Layer C and G. It is possible that the developers suppose some source files belong to layer D while they may pertain to other layers or modules in reality. Similar to RULE 1, some of the violations are caused by the source files from the Core folder and may be fixed after these files are moved to the folder that represents the layer these files belong to.

Additionally, we have observed certain violations that exist in both the rule checking of disallowed layer dependency and non-public interface use. This is because the definition of the two rules overlaps partially. If a layer illicitly depends on another layer, the header files it includes from this dependee layer must be a disallowed interface.

5.2.3 Dynamic Memory Allocation

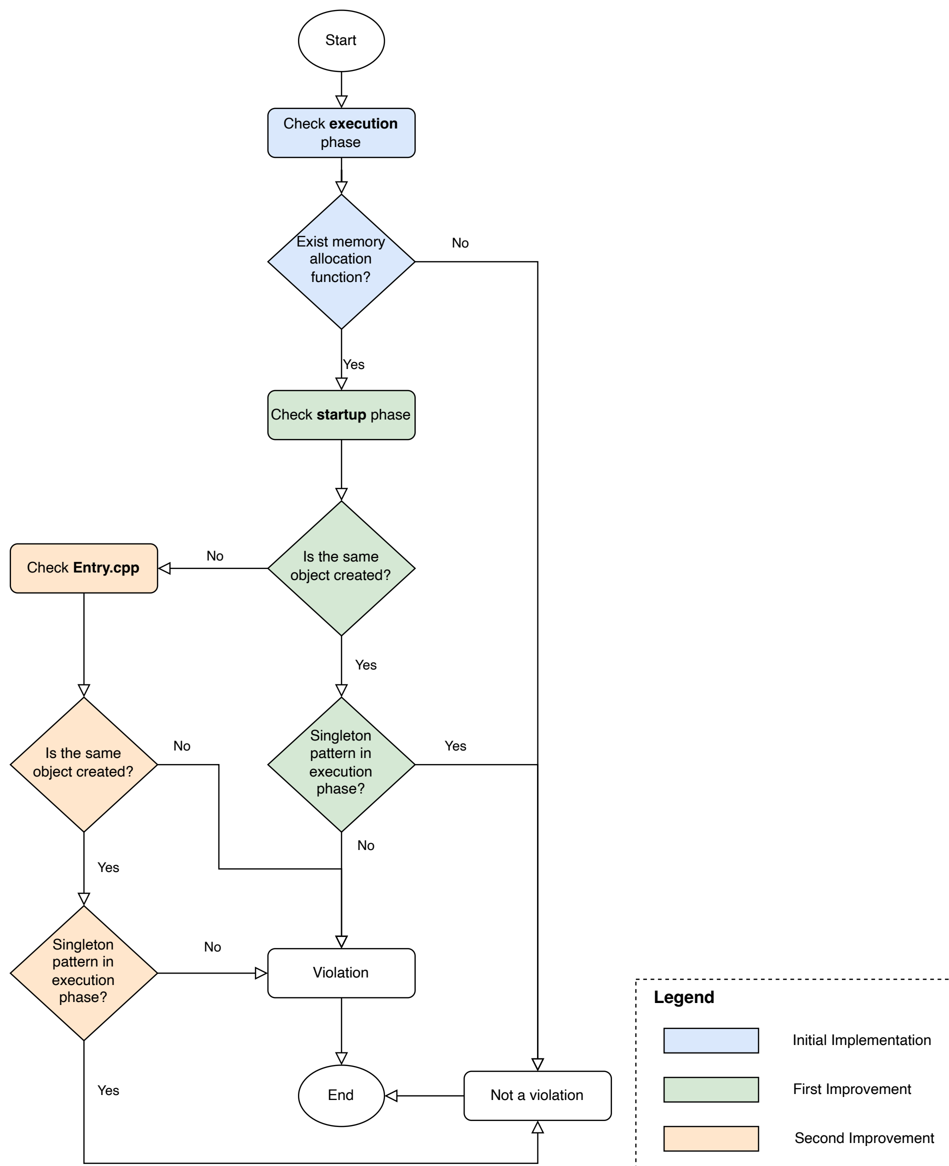


Figure 5.5: Workflow of checking dynamic memory allocation

This architectural rule forbids the dynamic memory from being allocated after the startup phase of the application. In this section, we count and investigate the false positives and the false negatives in the detected violations.

The goal of RULE 3 is to detect the use of dynamic memory allocation functions or operators in the body of the functions belonging to the execution phase of the applications in the positioning software. However, an important fact to note is although the functions for dynamic memory allocation are used in the execution phase of the application, they may not be executed during the runtime. This is because dynamic memory allocation is always associated with the instantiation of the new object in the codebase, and the instantiation of the object is conditioned by the singleton pattern. Therefore, if we ignore the singleton pattern and assume all dynamic memory allocation functions must be executed in runtime, the results of the rule checking may contain false positives. As for the false negatives, they correspond to the case that the memory is allocated in the execution phase but not captured by our checker. This section will proceed with the evaluation of RULE 3 from the two aspects mentioned above.

To implement READ, we query the call graph of the target source file to check if `malloc`, `alloc` and the `new` operator are called in the execution phase of the applications. This formulates the initial implementation as illustrated in the blue components in Figure 5.5. This implementation equates with the definition specified in Listing 4.4.2.

Table 5.12: Number of violations of dynamic memory allocation collected in each implementation

	Initial Implementation	First Improvement	Second Improvement
#(Violations)	45	21	1

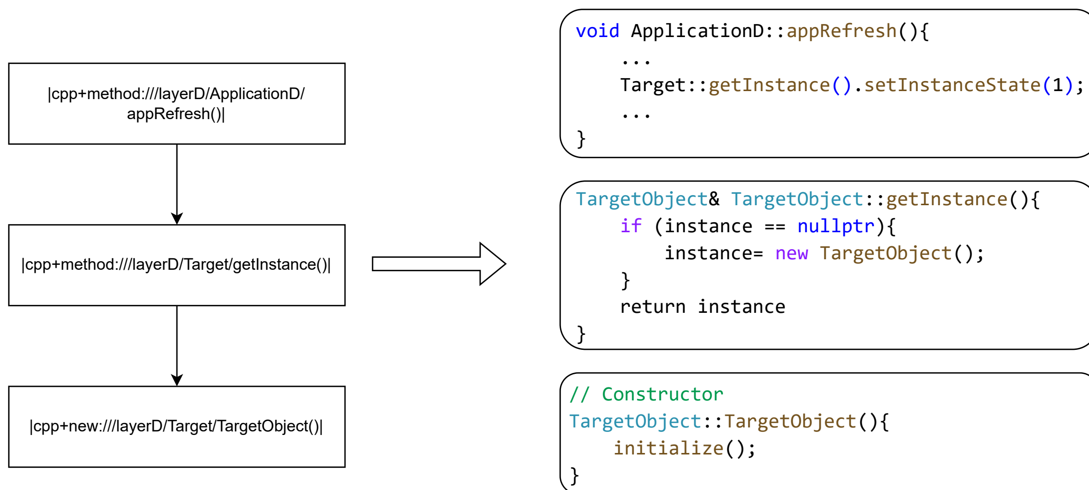


Figure 5.6: An example of the violation results for RULE 3 (Left: results in RASCAL; Right: source code in C/C++)

As shown in the first column of Table 5.12, we obtain 45 violations from seven source files. In each source file, one or more functions in the execution phase are implemented. These functions invoke other functions, reaching the dynamic memory allocation functions or the `new` operator. An example of the violation can be found in Figure 5.6, which illustrates a chain of the function calls from a function of the execution phase through multiple intermediate function calls and, finally, to a call on the new operator. This chain can be verified by inspecting the corresponding C/C++ source code, as shown

on the right side of the figure. We mapped the logical representation of the function to its physical location in the source files. It validates the existence of the chains obtained from the checker.

Evaluation-9: Evaluation of the initial implementation of RULE 3

Introduction and Evaluation Question
The memory is allocated to prepare the space for one or multiple objects. The allocation process is usually along with the invocation of the constructor, as shown in the last function call in Figure 5.6. However, if we observe the function call in which the `new` operator is used, we can find the objects are created with the Singleton pattern. A comprehensive investigation of the checking results shows that for all of the collected violations, the objects are created with the Singleton pattern. The singleton pattern is a common design pattern in software development to ensure only one instance of the target class is created [14]. In the context of our codebase, if the object is created before the application's execution phase, then even though dynamic memory allocation functions are used in the definition of functions of the execution phase, they are not executed in the runtime. Therefore, we need to filter the case that the functions for dynamic memory allocation are actually not called because of the singleton pattern after the positioning software is compiled and executed. These filtered results are false positives in our checking results and must be removed. Based on the analysis, the first evaluation question of RULE 3 is proposed: How many false positives resulted from the singleton pattern do the results contain?

Evaluation Method
The test set is all the source files in the result set collected by the initial implementation. The design of the evaluation is to check the definitions of the startup phase functions and whether the same objects created in the execution are already created in the startup phase. The green part in Figure 5.5 reveals the logic of the checks related to the object creation before the execution phase. Specifically, we check the startup phase and extract all the instantiated objects. Then, we build the AST of the source file to validate if the objects created in the execution phase conform to the singleton pattern. Here, the validation relies on the AST instead of the M3 model because the model does not contain information about the content of the condition in the if-statement. If the AST demonstrates that the same object is expected to be created in the execution phase but with a singleton pattern, then it is not a violation, as the statement with dynamic memory allocation will not be executed.

Evaluation Results
The evaluation result is the outcomes of the initial implementation contain 24 false positives. As revealed in Table 5.12, the number of violations reduces to 21. This means that the singleton pattern indeed influences the instantiation of the objects in the runtime stage and should be considered in the checking phase.

Conclusions and Possible Threats to Validity
In the initial implementation of RULE 3, the singleton pattern is not considered, leading to false positives in the checking results. The first improvement of the implementation is a process of refining the definition of the rule in READ. This process requires the inspection of the source code and extension of the DSL syntax. From this improvement, we learn that it is important to notice the difference before and after the code is compiled and executed. Our tool should simulate the real execution of the positioning software to obtain the real violations.

Evaluation-10: Evaluation of the first improved implementation of RULE 3

Introduction and Evaluation Question
Based on the results of the first improvement, we have another round of evaluation. The intuition is that the startup phase is not the initial entry point of the applications. In the positioning software, a source file known as `Entry.cpp` is implemented to launch the applications of all the layers. In other words, it is the entry point of all the applications, and the functions defined in this file are invoked before the functions in the startup phase. Therefore, we proposed the evaluation question of whether

the objects supposed to be created in the execution phase are already created before the startup phase in `Entry.cpp`.

Evaluation Method

The test set of this evaluation is all the source files where the violations are collected in the first improvement of the implementation and the `Entry.cpp`. The yellow part in Figure 5.5 demonstrates the workflow of the evaluation. The logic is similar to the first evaluation in obtaining the objects constructed both in the `Entry.cpp` and the execution phase. If the obtained object is constructed in the execution phase with the singleton pattern, it is a false positive and removed from the result set.

Evaluation Results

As a result of the evaluation, the number of violations decreases to 1, as shown in Table 5.12. To verify its correctness, we construct a call chain, like in Figure 5.6, starting from the function in the execution phase to the location where the memory is allocated. This is the final result of the checking of the dynamic memory allocation rule.

Table 5.14: Confusion matrix for the first implementation of RULE 3

Total N/A	**Predicted Positives** 45	**Predicted Negatives** N/A
Actual Positives 1	1	N/A
Actual Negatives N/A	44	N/A

Table 5.14 reveals the confusion matrix of RULE 3 with regard to its first implementation. As shown in Figure 5.6, we obtain one truth positive after two improvements of the initial implementation, while the remaining 44 violations collected by the initial implementation belong to the false positives. Nevertheless, as the ground truth is absent for this rule, we have no idea about whether some violations are undetected by our tool, leading to 'N/A' in the 'negatives' column. In fact, we identified some factors causing false negatives in the result set by analyzing the source code and discussing it with Philips engineers. These factors will be discussed in Evaluation-14.

Conclusions and Possible Threats to Validity

In conclusion, to evaluate the accuracy of the results, we investigated the source files and proposed two improvements to filter the false positives resulting from the singleton pattern. The reason for both improvements is that the previous implementations are imprecise as the knowledge about the codebase was insufficient. Additionally, both improvements are specific to the codebase such that they may not be reusable for the other projects. However, the lesson we learned from the singleton pattern is applicable in the research of other codebases. As it is a widely used design pattern, when we analyze the call graph to extract the calls on the constructors, we may need to consider the singleton pattern, which determines whether the constructor is executed.

Evaluation-11: Evaluation with the Top-down Mutation Testing for RULE 3

The above-described measures promote the decrease of false positives in the detected violations for RULE 3. However, the result set may still contain problems as the false negatives are introduced while the rule is checked. We did not discuss the false negatives in the previous two rules because the comparison with the ground truths has verified the number of false negatives remains 0. In terms of Rule 3, false negative means the functions and operator executing dynamic memory allocation are used in the execution phase but not detected by our tool. To fulfill REQ3, we must investigate the possible false negatives contained in the results. In this subsection, we explain how the false negatives are analyzed and evaluated with mutation testing.

Mutation testing is a well-known method for assessing the quality of software testing. The testing relies on introducing mutants into the program's source code. These variables are artificial defects or can eventually lead to problems. The program with the defects is called a mutant. The target is to use the test suite to detect the introduced defects in the mutants. To quantify the test result, a term mutation score is proposed. It is the ratio between the number of detected mutants and the total number of injected mutants. The higher this score is, the more intentionally inserted problems are captured by the test suite [31]. In our setup, every mutant that is not detected is a false negative that can be counted for our evaluation.

Introduction and Evaluation Question

The evaluation question is whether the call graphs, based on which the checking for RULE 3 is performed, miss the information of the source code, leading to undetected violations. The motivation has been discussed above that the false positives have been eliminated from the results obtained from the initial implementation of RULE 3, while the false negatives may happen during the rule checking, which could be detected by mutation testing.

Evaluation Method

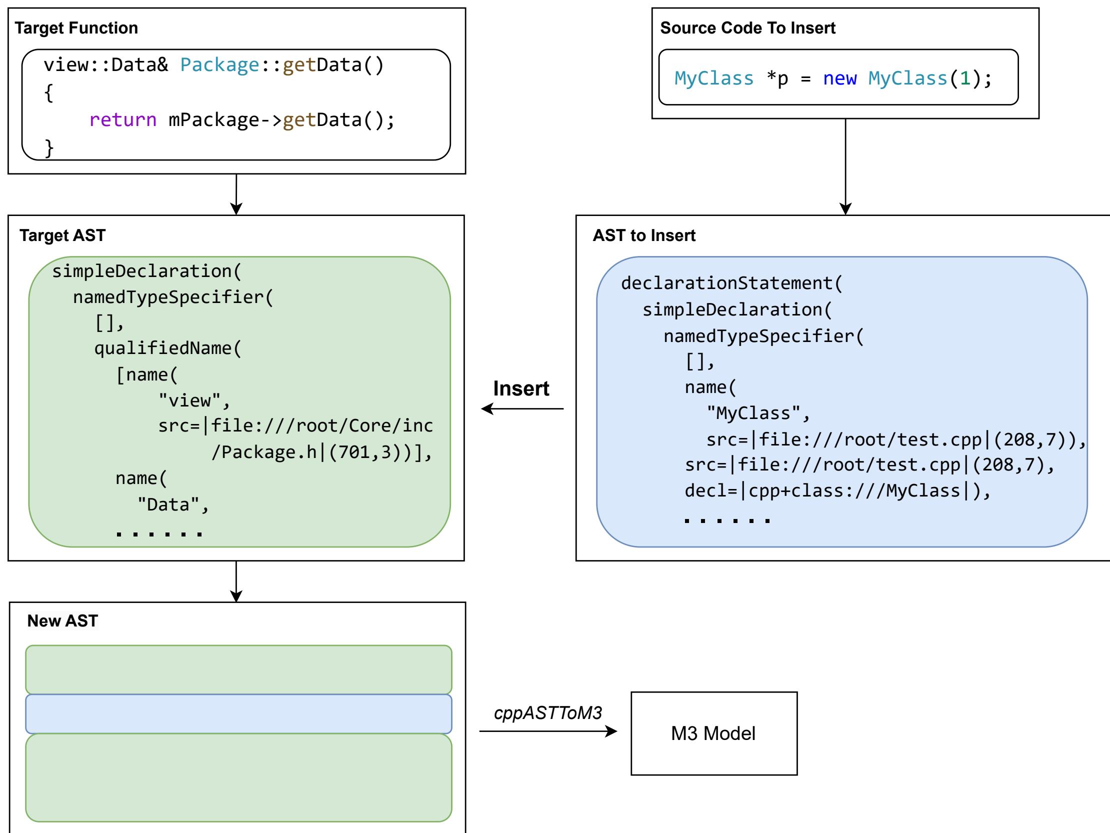


Figure 5.7: Design of the top-down mutation test

The design of the mutation testing is as follows: In the context of this assignment, the functions and operator for dynamic memory allocation, like `new` operator, is the variant of the test. We plan to manually add dynamic memory allocation to the source code so the modified programs become mutants. To design the mutation test, we choose a function `appRefresh` from the execution phase of an application in the codebase and inject a line of C++ code with `new` operator into the definition of some functions. These functions are deliberately selected so that they must be invoked after `appRefresh` is executed. Figure 5.7 illustrates the implementation of our idea. Here, instead of direct operation

on the source code, we manipulate the ASTs to inject the `new` operator. Specifically, we transform the line of code with `new` into an AST and insert this AST into the AST of the target functions, which are reachable from the `appRefresh`. This operation with AST is equivalent to adding the source code with `new` into the function body of `getIApp`. The benefit is it is a more convenient way to add the mutant without changing the source code. Eventually, a new AST is generated and transformed into a new M3 model. The final step is to check whether it exists at least one path in the call graph starting from `appRefresh` to the added `new` operator. As we have ensured the source code used for insertion does not exist anywhere in the codebase before the injection, if the desired `new` operator is found, then its appearance must be due to the mutation test.

Evaluation Results

The result is shown in Table 5.16. We inserted the `new` operator into nine functions, seven of which were found in the chain of function calls starting from `appRefresh`. The mutation score for our test is 0.78. The function call chains of two functions are disrupted halfway for the same reason. The intermediate function, where the chain is broken, is not successfully resolved such that in the call graph of the M3 model, the logical name of the function is marked as a problem. The failed function resolution is caused during the model generation when a list of standard libraries is required to parse the built-in functions and artifacts used in the codebase source files. However, the standard library has multiple different implementations, such as MSVC from Visual Studio and implementations from the compilers. If a wrong implementation is provided to the Eclipse CDT parser, the function cannot be resolved, leading to the problem messages in the M3 models. To avoid this problem, we tried the combination of different standard library combinations in different orders. The order plays an important role because, in our standard library list, two implementations may implement the same function, and we need to ensure the correct one has higher priority for use when the source code is parsed. In this way, a new list of standard libraries is determined and used to regenerate the models for the source files containing parsing problems. This improvement enables all nine insertions to be detected in the call graph.

Table 5.16: Results of the top-down mutation testing

#(Mutants)	#(Killed mutants)	Mutation score
9	7	0.78

Conclusions and Possible Threats to Validity

In conclusion, mutation testing is an effective method that enables us to find false negatives in the collected checking results. These false negatives are caused by the implementation of RULE3. More specifically, the problem happens when another instance of the include path is used to resolve the C and C++ artifacts defined in the standard library.

In Evaluation-2, Evaluation-3 and Evaluation-4, we have investigated the impacts of not found include paths on the final checking results. However, these evaluations did not inspect the include paths of the standard libraries. In this evaluation, it is more clear that unexpected different header files for different compiler versions of C and C++ can lead to resolution problems, further leading to unresolved messages in the M3 models and increasing the number of false negatives in the checking results. Therefore, this evaluation not only improves the implementation of RULE 3 but also improves the overall quality of the M3 models.

Evaluation-12: Evaluation of the Problems in the M3 models

Introduction and Evaluation Question

This mutation test is not specific to the checking of the dynamic memory allocation rule. Instead, it is a generic way to evaluate the quality of the M3 models. A higher number of problem messages in the model indicates more parsing problems during model generation. In the last test, we learned that

the correct implementation and the order of the standard libraries have a significant impact on the generated call graphs, contributing to the lower number of problems in the model. This lesson inspires us with a new evaluation question: How will the quality of the M3 models be improved if we improve the list of standard libraries for all the source files during the model generation?

Evaluation Method

The test targets in this evaluation are all the source files, excluding the legacy code stored in the `Legacy` folder. The metric to evaluate the quality of the M3 models is the number of records marked as a 'problem' in the models. As discussed in Evaluation-11, the problems in the M3 models can be caused by false include paths of the standard libraries. We regard these include paths as an old list of include paths. The models were improved and updated by using a new list of include paths that point to the correct implementation of the standard libraries. The process to update the models is described by the pseudocode in Algorithm 5. When generating the models, we always use the new list of include paths for the standard libraries first. If a parsing error arises, then we replace the new list with the old list to get the models. The error happens here because the new list of standard libraries may not be suitable for all the source files. Certain source files may require the standard libraries from the old list. After all the models were updated, we counted the number of the problem messages 'Attempt to use symbol failed' of all the models and compared the value with the one from the old models.

Algorithm 5 Model generation with the improved list of standard libraries

Input:

A {`source`} location to the C++ source file
A {`stdlib_new`} list containing the new composition of the standard libraries
A {`stdlib`} list containing the standard libraries used for previous model generation
A {`includePath`} set containing the include paths of the given source file

Output:

A M3 model corresponding to the given source file

```
1: model ← emptyM3Model()
2: try
3:     model ← createM3FromCppFile(source, stdlib_new, includePath)
4: catch Exception
5:     model ← createM3FromCppFile(source, stdlib, includePath)
6: end try
7: return model
```

Evaluation Results

The comparison result is revealed in Table 5.18. The total number of problem messages in the 4232 M3 models is decreased by 62%. This is a significant improvement in the model quality. Despite the improvement, the total number of problems still remains high. The possible reason is that the new list of standard libraries is still not ideal for all the source files, and thus, some artifacts of the standard libraries are incorrectly resolved. Nevertheless, considering the total number of source files, it is impractical to determine the most suitable list of standard libraries such that the number of problematic function calls is minimized.

Table 5.18: Number of distinct problems before and after the improvement of the list of standard libraries

	Before the Improvement	After the Improvement
#(Problems in the call graph)	406715	154233

Conclusions and Possible Threats to Validity

In conclusion, the improved list of include paths for C and C++ standard libraries contributes to fewer problems in M3 models. This improvement enhances the model quality and may also contribute to the checking of other architectural rules as the failed resolved names are now presented in the models. Despite the significant improvement, it still problems in the model, which require further investigation to fix them.

Evaluation-13: Evaluation of Checking Results with the New M3 Models

Introduction and Evaluation Question

In Evaluation-12, we generated a new collection of models which contains significantly fewer problems. As these problems may lead to false negatives in the results, the removal of these problems may contribute to the identification of new violations. Due to this reason, we conducted an evaluation with the question: Do we get more violations with respect to each architectural rule after the quality of the M3 models is improved?

Evaluation Method

The test targets are all source files in the codebase. We checked RULE 1 and RULE with the new models and the rule implementation after the second improvement. As for RULE 3, We checked the dynamic memory allocation with the improved implementation of the checker, as illustrated in Figure 5.5. The results of each rule were compared with the results obtained before the models had been updated.

Evaluation Results

The results of this evaluation are that the number and the content of the violations of each rule are the same before and after the update of the M3 models. This means although the old M3 models contain a large number of problems, these problems do not eventually influence the checking process.

Conclusions and Possible Threats to Validity

In conclusion, the checking results of all three architectural rules are not affected by the failed name resolution contained in the old M3 models.

Evaluation-14: Evaluation with the Bottom-up Reachability Test for RULE 3

Introduction and Evaluation Question

In the above mutation test, we start from the `appRefresh` function and explore whether the inserted `new` operators can be detected. In this process, data is from the caller at the top to the callee functions at the lower side. Next, we interpret another evaluation test in which the data flows from the bottom (callees) to the top (callers). Essentially, this is not a mutation test, as no mutant is injected into the source code. Nevertheless, the design of the test is similar to the mutation testing. Therefore, we introduce the test in this section.

The intuition of this test is that we have assumed that the `Entry` source file is the entry point of the codebase applications. Each application is implemented in a single source file. To express RULE 3, we've had to assume that all the functions defined in these source files are reachable for the functions defined in `Entry` file. That means, for all of the function call paths ending at a function defined in an application source file, at least one path starts from `Entry`. These assumptions need to be tested for validity: whether all the functions defined for the applications of the positioning software are reachable from the functions defined in `Entry`?

Evaluation Method

In this case, the test set is the application file in which an application is implemented with the startup and execution phase functions. The `Entry.cpp` is another target for testing. The workflow of the experiment is illustrated in Figure 5.8. Assume we select a source file containing the definition of functions in different application phases. This source file is designated as a target source. Then, we build an inverted call graph from the callee function to the caller function for the entire codebase. We calculate the transitive closure of this call graph such that with a given callee, we can immediately obtain all the reachable callers. In the next step, we query the callers of each function defined in the

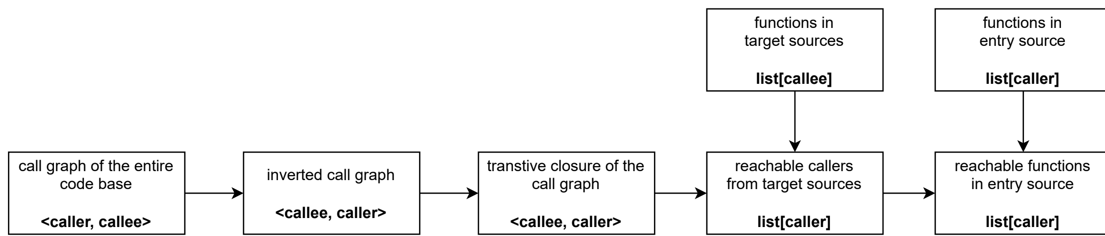


Figure 5.8: Workflow of the bottom-up test

target source and compare these callers with the function defined in Entry. If none of the callers belongs to Entry, the initial callee is then unreachable from the entry point of applications.

Evaluation Results

Table 5.20: Results of the bottom-up reachability test

Category	Number
#(Failures due to function pointer)	8
#(Failures due to template)	7
#(Failures due to calls from another entry point)	16
#(Failures due to destructor)	4
#(Failures due to defined-not-used)	7
#(Total failures)	42
#(Total functions for test)	227
#(Failure rate)	18.5 %

As demonstrated in Table 5.20, we tested 227 functions defined in three source files, each of which implements an application. For 18.5 % of the functions, no path from the bottom callee to the top caller was found. We summarize the reasons for the unreachability into five categories:

- **Function pointer**: With static analysis, it is difficult to know the exact executed function when a function pointer is called. This is especially the case for the interprocess communication module in the codebase, in which the callback function is registered through a function pointer. As a result, the function is called but not recorded in the call graph in the M3 models. In our test, eight failures belong to this category.
- **Template**: The C++ template is tricky to resolve. In our test, all constructors failed to have a path leading to the top caller. The reason is the constructors are invoked implicitly through the C++ standard library function make_unique<constructor_name>. The constructor is used for the template argument of make_unique, which is not captured in the M3 models.
- **Calls from another entry point**: Our assumption that Entry.cpp is the sole entry point of the applications is not always true. The software allows users to operate it through an external console by inputting commands. Through analyzing the name of the failed function, we found that some of them are designed to support the external console. In other words, these functions may be called through external commands, and the top callers should be from the modules of the external console.
- **Destructor**: The destructors are invoked implicitly. As a result, they are absent in the call graph of the M3 models.

- **Defined-not-used**: Some functions are defined in the source files but not used anywhere in the codebase. A possible cause is the bugs that some features are not correctly implemented, and the defined function is not used.

Above are five factors that lead to the broken paths between the bottom callees at the application source files and the top callers in ENTRY.CPP. These broken paths may finally lead to false negatives for RULE 3. It is evident that the first two factors are due to the limitation of the M3 model and the parsing capability of CLAIR. Consequently, no measure is proposed to improve the evaluation results. As for the broken chains due to function calls initiated from the external console, we could consider it as the second entry point to use the applications. The broken chains resulting from the last two factors could be omitted as the problems are not due to incorrect implementation or the limitation of the method.

Conclusions and Possible Threats to Validity
Similar to Evaluation-12, this test is also a generic method that enables us to understand why the chains of function calls are broken in the call graph. The main causes include the use of function pointers, templates and external inputs for function invocation. This test provides us with a lesson about the limitation of the call graph in processing function pointers and templates. We also have a deeper understanding of the coverage of the method we applied to analyze the function uses that `Entry` file is not the only entry point of the application functions.

The results from the second test may contain false positives. In the call graph of the M3 model, if a method, which is an implementation of a virtual method, is called, its logical name in the call graph is the same as the one of the virtual method. This is because the virtual function may have multiple implementations. When it is non-deterministic, which implementation is used as the callee, it is presented as the same as the virtual method in the call graph. However, we need the actual method to build the call chains. Otherwise, the chains may terminate at the virtual methods. To tackle this, we replace the virtual methods with all the possible actual methods in the call graph. This measure favors the composition of the function calls but also introduces a problem. When the callee is used to join two chains of function calls in an implementation of a virtual method, we may select an actual method different from the one in the runtime. As a result, we may obtain results that are actually unreachable and thus lead to false positives in the detection of RULE 3.

Evaluation-15: Expressiveness of the DSL Syntax

As part of the evaluation of the implementation of the architectural rules, we also have a test for the DSL syntax, as it regulates how the architectural rule is formulated and integrated into our tool. We would like to check whether the DSL fulfills REQ1 and REQ2. The evaluation is mainly on the expressiveness of the syntax.

Introduction and Evaluation Question
The syntax of the READ specification language is deliberately tailored to define at least the three required architectural rules. Nevertheless, it is also applicable to the formulation of additional rules without modifications to the existing syntax or rule checker. To explore to which extent the language can formulate the other rules, we proposed the evaluation question: How expressive is the current DSL syntax?

Evaluation Method
The expressiveness of READ syntax is measured by to which degree the current syntax can describe the other unchecked architectural rules. The test set comprises all the unimplemented architectural rules mentioned in Appendix B. For example, we can implement RULE 5, which forbids the use of the built-in `sleep` function in the codebase, with the use-statement. Listing 5.2.1 as a naive implementation of this rule. It specifies for all of the source files in the codebase, all the functions except `TOS_p_TSK_sleep` are not allowed to call `sleep` and `Sleep` functions. Due to the unavailability of ground truths for unchecked rules, evaluating the results remains infeasible. Nevertheless, based on the current definition of RULE 5, the tool yields outcomes consistent with our expectations.

Listing 5.2.1 DSL instance implementing the checking on function `sleep`

```
rule name: sleepRule{
    check source files: ALL
    constraint{
        ANY user in: ALL except Function [TOS_p_TSK_sleep]
        not use target : Function [sleep, Sleep]
    }
}
```

Evaluation Results
As a result, except for the already implemented rules, if we have the information about the objects and subjects of the rule as well as the mapping between the objects and subjects, the current syntax supports a naive implementation similar to Listing 5.2.1 for RULE 5, 7, 8, 9, 11, and 12. The description of the rules is in Appendix B. Although the actual implementations may be significantly different from this naive implementation if the rules are further refined with deeper investigation, the syntax reveals its expressiveness in defining 60% of the remaining rules based on the current rule descriptions.

Conclusions and Possible Threats to Validity
In conclusion, the DSL syntax demonstrates a certain degree of expressiveness by describing the rules beyond the scope of this assignment. Some rules cannot be fully implemented with the current syntax as they require checks on the level of the source code for code smell detection, such as RULE 13. However, READ is a tool specialized in inspecting the violations at the architecture level. Extending READ from architecture erosion to smell detection would entail changing several design decisions.

5.3 Response Time

Regarding the third aspect for evaluation and REQ5, we explore the response time of the tool. The operation of the tool mainly consists of four phases: CMake parsing, model generation, model composition, and rule checking. The time to parse the CMake files is much shorter than the other three phases that it takes less than one minute to parse 606 CMake files. Therefore, in the following, we mainly discuss the time required by the other three phases.

Evaluation-16: Impact of the search space of the include paths

Introduction and Evaluation Question
Model generation has a significant impact on the overall response time as it is the most time-consuming process in the workflow. With a machine with 48 GB memory (12 GB for Java virtual machine (JVM)) and six cores, generating a M3 model needs an average of 12 seconds. As one factor, the generation time is positively correlated to the file size of the source code. The file size can be represented by the LOC, including the headers included by the source files, as the content of the headers is copied to the source files at the beginning of the parsing. As the file size is an inherent property of the source file that cannot be changed, no improvement is proposed with respect to this factor. The evaluation in this subsection focuses on another factor: include paths.

In this assignment, a CMake parser is implemented to extract include paths for the generation of the M3 models. Another approach to obtain the include paths is to construct a list containing all the directories in the codebase. The directories here mean the locations of all the folders in the codebase. This approach is more straightforward and simple as it does not require the implementation of a CMake parser. Besides, as it contains all the directories, it is supposed that the not found include paths do not appear during model generation. Nevertheless, it does not mean the second approach is more efficient, as this set contains a large number of irrelevant include paths for the specific source file for model generation. When the parser parses inclusion in source files, it will search the paths until it finds the matched one. That means a higher number of irrelevant paths in the search space prompts a longer time of parsing and model generation. Therefore, the evaluation in this section is to compare

the time of model generation supported by include paths collected by CMAKE parser and by a general set of all the directories.

Evaluation Method
To implement the evaluation plan, we used the CMAKE parser to extract a list of include paths for each source file. Also, we stored all the directories in the codebase as a list, which serves as a general collection of the include paths. Then, 100 source files are sampled from the codebase. The sampling is conducted by the RASCAL function `sample`. This is a simple random sampling process that selects 100 source files randomly from all the source files that require a M3 model. The two sets of directories were used to generate the M3 models for these 100 source files, respectively.

Evaluation Results
We recorded the time of model generation in both cases, which is revealed in Table 5.22. It is clear the model generation with a list provided by CMAKE parser takes 6.7 minutes to generate 100 models for the sampled files, while it takes about 12 times the time to generate the models with the general list of directories.

Table 5.22: Comparison of the time required to generate the models for 100 sampled source files with different sources of include paths

Source of include paths	Paths collected by CMAKE parser	All directories in the codebase
Time of model generation (minutes)	6.7	79.4

Conclusions and Possible Threats to Validity
In conclusion, using the include paths from the CMAKE parser is a much more efficient solution than the other method. This means the list of directories in the second method, although it is a generic solution for include paths and prevents the absent paths from happening, contains too much irrelevant information such that the parser needs much more time to find the matched path. This evaluation reveals the necessity of the CMAKE parser, which provides a tailored list of include paths for each source file.

Evaluation-17: Impact of the partial composition of the M3 models

Introduction and Evaluation Question
As for the time of model composition, it depends on the size of the model. The models are required to be composed into layer models when RULE 1 and RULE 2 require checks based on models. As each M3 model contains 24 fields with sets of possibly 10,000 of tuples, the composition of the complete model can be time-consuming. A possible improvement for time and space efficiency is only to compose the fields of the M3 models, which are required by the rule checking while leaving the other fields unaffected. Then, an evaluation question is proposed: How is the efficiency of READ improved if only the selected fields are composed as the layer models?

Evaluation Method
For this evaluation test, all the M3 models formulate the test set. We compose the complete fields and the selected fields and calculate the time used for model composition, respectively.

Evaluation Results
In Table 5.23, it is evident that each layer or module requires less time to generate a composed model when only a subset of the fields are required for model composition. The composition time of the partial model is nearly 30% less than the composition of the entire model.

Conclusions and Possible Threats to Validity
In conclusion, composing the models with only the required fields is an effective approach, contributing to reduced execution time. Additionally, it is important to note that the effectiveness of this improvement may be reduced as READ is extended for more rule checking and requires more fields for model composition.

Table 5.23: Comparison of the time required to compose the models between partial and complete model fields

Layer/Module	A	B	C	D	E	F	G	**sum**
Time of partial model composition (minutes)	0.11	0.72	0.61	3.12	0.51	0.13	1.09	6.29
Time of complete model composition (minutes)	0.15	1.04	0.86	4.37	0.78	0.18	1.55	8.93

Evaluation-18: Impact of AST cache for checking RULE 3

Introduction and Evaluation Question
The time of rule checking varies on which architectural rule is inspected by the checker. Generally, the time required by RULE 1 and 2 is less than RULE 3 because the checking of the first two rules only depends on the `requires` field in the model, which is convenient to retrieve. In comparison, RULE 3 requires the visit of the AST to verify whether the dynamic memory is allocated with the singleton pattern. This process involves the construction of the AST, which is time-consuming and could be the bottleneck of rule checking. With further investigation, we found that some files are frequently used to construct the ASTs as they implement the functions responsible for creating the instances. Therefore, to improve efficiency, an AST cache is constructed to store the most recently used 20 ASTs. This is an in-memory cache that is built by a set in RASCAL that allows a maximum of 20 ASTs to be stored. Once the capacity of the set is reached, the earliest inserted AST will be removed. This improvement leads to an evaluation question: how much time is saved by applying the improvement?

Evaluation Method
The test set of this evaluation consists of all the source files in the codebase, as we need to inspect all source files to check if dynamic memory allocation exists in the execution phase of the application. The design of the evaluation is as follows: We use the same implementation of RULE 3 mentioned in Evaluation-10 twice. The variant is the AST cache. The first test was conducted with the AST cache, while the second was without.

Evaluation Results
Table 5.24 illustrates that the rule-checking process for over 4000 source files without the cache requires about 14 minutes. In comparison, checking with the AST cache contributes to a 25% reduction in time consumption.

Table 5.24: Comparison of the time required to check RULE 3 with and without the AST cache

	With cache	Without cache
Time for rule checking (minutes)	10.56	14.10

Conclusions and Possible Threats to Validity
The facts in Table 5.24 prove that AST cache is a practical improvement when a large number of source files are inspected continuously. This measure contributes to the reduced overall execution time required by READ.

5.4 Anecdotal Evidence

In this section, we will discuss the feedback from Philips engineers who have knowledge of or have already used READ. The target of READ is to formulate the architectural rules and compare the source code represented by M3 models against the rule definitions to collect the violations. As mentioned in Section 4.5, READ is deployed on Philips' server and executed as a Jenkins job. The architect maintains this Jenkins job and checks the results collected by the tool.

From the perspective of the architect, READ reports accurate violation results for RULE 1 and 2, consistent with his own tool. While fewer violations of RULE 3 are identified than expected considering

the software's scale, the collected violation is considered to be correct. Moreover, Jenkins' dashboard facilitates the presentation of the checking results from READ and improves the user experience. The architect also sees potential in using the generated M3 models to analyze and maintain the architecture of the positioning software.

Another feedback received from the developers shows some concerns about the maintenance of the tool. As READ is implemented in RASCAL, fixing the problems in the CMAKE parser as software evolves and accommodating the DSL syntax to new architectural rules require the knowledge and experience of using RASCAL. However, it's worth noting that these concerns could be mitigated as the developers of the software group are acquiring proficiency with RASCAL.

5.5 Conclusions

In this section, we investigated the usability of the DSL tool by evaluating the accuracy and efficiency of its individual components. Usability was primarily gauged based on the model quality, accuracy of checking results and the time required to identify the violations in adherence to architectural rules. The model quality was evaluated and further iteratively improved by counting the number of not found include paths.

To evaluate the accuracy, multiple evaluation tests were designed for each architectural rule. These tests enable us to find the problems in the implementations and improve the accuracy of the rule checking. Consequently, the evaluation provides evidence of the correctness of the correctness of the implementation for RULE 1 and 2. Furthermore, with the evaluation of RULE 3, we continuously enhanced the preciseness of rule definition and implementation, leading to improved result accuracy. With the mutation test and reachability test, the problems in the M3 models were deeply investigated. These problems are either from the model generation phase due to incompatible implementation of the standard libraries or the nature of the CLAIR that it under-approximates function pointers and templates, leading to false negatives in call edges. The lesson enables us to improve the model quality and have a better understanding of the limitations of the method used.

As for response time, it was calculated for each execution phase with the comparison of the improvement measures. The calculation results demonstrate our measures regarding reducing the overall execution time are effective.

In conclusion, the evaluation allows us to have a better understanding of the performance of each tool component from different aspects. It also promotes better design and implementation of the tool such that the tool becomes more accurate and efficient.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we propose an automated method for identifying architecture erosions in the Philips positioning software. This method is implemented with the meta-programming technology RACAL and its extension CLAIR, which enables the formulation of architectural rules in a DSL and the conformance checking on the abstraction of C/C++ source code. Users can easily define architectural rules with the DSL syntax, while the checkings are performed by the tool automatically. The tool is assessed evaluating the accuracy of the checking results and response time of the entire checking process. Based on the evaluation findings, new designs and implementations are proposed to eliminate the false results from the detected violations and enhance the checking efficiency. As a result, the implemented tool proves to be usable and reliable in identifying the violations of RULE 1 and RULE 2. As for RULE 3, an effective approach is proposed and implemented to detect the violations, while the obtained results may represent a subset of the ground truth due to the limitations in checking function pointers and templates. Overall, the tool's development and evaluation contribute to an improved understanding of manipulating DSL and M3 models for architecture conformance checking, as well as revealing the gap between the designed architecture and the actual software structure.

The tool's correctness in identifying rule violations proves that meta-programming is a more effective method for tasks required by Philips than the other technologies we studied during the literature study. For example, compared to the reflexion model, which may involve manual construction of the mapping between the intended architectural model and the actual implementation model [24], our DSL tool automates the checking process once the rule is expressed with the DSL syntax. Additionally, to implement the rule conformance checking, the reflexion model requires a one-to-one mapping between the components in the architectural model and the implementation model. This mapping limits the reusability of the models in dealing with different architectural rules [38]. In comparison, the DSL syntax can be reused to define rules with similar formulations. Hence, in the context of this assignment, a DSL-based method provides much more flexibility to restrict the behaviors of the architectural components.

Our work's major contribution is a DSL-based approach, which automates the processes of checking architectural rule violations in positioning software consisting of more than one million lines of code and provides results concerning architectural violations. Although the language is designed for rules specific to the codebase, it is generic and expressive to formulate other unchecked rules. Moreover, to the best of our knowledge, this thesis represents the first utilization of mutation testing to evaluate the quality of architecture conformance checking using a RASCAL-based tool. While some researchers proposed a tool to generate mutation tests for arbitrary DSLs [15], our requirement on the mutation tests differs as we focus on the investigation of missing information in the call graph. Through the mutation test, we successfully identified the flaws in model generation and the limitations in constructing chains of method calls within our tool.

Finally, we draw a conclusion for each research question proposed in Section 1.3:

- **RQ1: Which method is suitable to check AE in Philips' codebase?**
In this assignment, we determine that DSL is the most suitable solution for architecture conformance checking for Philips' codebase. This decision is based on the literature study that DSL has been successfully used to validate the conformance between software architecture and actual implementations. It provides language grammar such that the users can translate the architectural rules described in natural language into a programming language. In comparison, other methods do not provide the flexibility for implementing rules closely specific to the codebase. Additionally, we selected RASCAL to develop the DSL because previous experience with the same codebase has proven that RASCAL is a feasible solution for source code analysis. Combining theoretical study and practical experience, we are convinced a DSL implemented with RASCAL is an ideal solution to identify AE in the codebase.

- **RQ2: How are the architectural rules formalized in the tool?**
The architectural rules are formalized with the DSL syntax. The formulation of the rule conforms to a basic scheme that each rule must contain three elements: a rule subject, a rule object and a modifier, which are implemented with algebraic data type in RASCAL. Additional conditions or conditions may be required depending on the concrete definition of the rule. Therefore, a final definition of the rule in the DSL requires a pre-investigation of the rule content and scope, as the plain description in natural language can be ambiguous and informal. The investigation includes analyzing the purpose of the rules from the view of the architects and the related code snippets in the codebase.

- **RQ3: How is architecture erosion detected by the tool?**
The essence of AE in the context of positioning software is the conformance checking between the implementation and the designed software architecture. As the foundation of the detection, a CMake parser was implemented to collect the include paths. These paths are essential as they indicate the locations of the headers required by the source files. To perform the checking, the C/C++ source code, coupled with the include paths, is transformed into a M3 model, which is a generic representation recognized by RASCAL. Then, the violations are collected by querying the models against the DSL instance, in which the architectural rules are defined.

- **RQ4: How is the quality of the developed tool evaluated?**
The evaluation of the tool mainly focuses on the usability of the DSL tool. We investigated three aspects to explore the tool's usability: accuracy, response time and DSL expressiveness. The accuracy of the checking results was evaluated by comparing the collected violations with the ground truths provided by the architect. In case the ground truth is unavailable, mutation tests are designed. For the rule checking based on the call graph, mutation testing reveals the function calls that are supposed to be present but actually absent. As for the response time, the execution time of model generation, composition and rule checking was calculated, prompting the improvement of design for enhanced efficiency. The DSL expressiveness was evaluated by testing its applicability with other unchecked architectural rules using the implemented language syntax.

6.2 Future Work

In this assignment, we collect 13 architectural rules through interviews with architects. The first three of these rules are implemented and checked in our tool. The ultimate goal of our tool is to check all the architectural rules defined for the positioning software. During the evaluation of the DSL syntax, we found the current DSL syntax already supports a basic implementation of some rules, as these rules possess similar patterns that can be formalized in the DSL. The future work for rule checking could be based on our basic implementation and may involve further investigation and refinement of the rules. As discussed in the evaluation section, the imprecise definition of the architectural rule can lead to increased false results. Hence, it is of great importance to have a precise rule definition and determine the mappings between the rule components and source code artifacts before implementing the rule.

Additionally, the DSL syntax may be extended such that it can describe the rule components which cannot be specified by the current syntax designs.

Additionally, during the evaluation of RULE 3, we identified some limitations of using CLAIR in analyzing templates and function pointers. It would be advantageous if improvements could be incorporated into CLAIR such that it could present the information about templates and function pointers in M3 models. By achieving this enhancement, we anticipate more violations concerning RULE 3 would be detected, further enhancing the tool's performance and usability.

Bibliography

[1] *Internal Philips Documentation*. ix, 6, 7, 17, 19

[2] Rodin Aarssen. cwi-swat/clair: v0.1.0. Sep 2017. 9

[3] Rodin Aarssen, Jurgen J. Vinju, Ruichen Hu, Davy Landman, Jouke Stoel, and Omar Duhaiby. usethesource/clair: v0.8.0, August 2023. 9

[4] Sharon Andrews and Mark Sheppard. Software Architecture Erosion: Impacts, Causes, and Management. *International Journal of Computer Science and Security (IJCSS)*, 14(2):82–94, 2020. 5

[5] Vidudaya Bandara and Indika Perera. Identifying Software Architecture Erosion Through Code Comments. In *2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer)*, pages 62–69, September 2018. 5

[6] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice: Software Architect Practice_c3*. Addison-Wesley, 2012. 1

[7] Bas Basten, Mark Hills, Paul Klint, Davy Landman, Ashim Shahi, Michael J. Steindorfer, and Jurgen J. Vinju. M3: A general model for code analytics in rascal. In *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*, pages 25–28, March 2015. 9, 10

[8] Isela Macia Bertran. Detecting architecturally-relevant code smells in evolving software systems. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1090–1093, 2011. 13

[9] Andrea Enrico Francis Caracciolo and Oscar Marius Nierstrasz. *A unified approach to architecture conformance checking*. PhD thesis, Universität Bern, 2016. 14, 16, 39

[10] Lakshitha de Silva and Dharini Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151, January 2012. 5

[11] Pär Emanuelsson and Ulf Nilsson. A Comparative Study of Industrial Static Analysis Tools. *Electronic Notes in Theoretical Computer Science*, 217:5–21, July 2008. 7

[12] Juarez LM Filho, Lincoln Rocha, Rossana Andrade, and Ricardo Britto. Preventing erosion in exception handling design using static-architecture conformance checking. In *Software Architecture: 11th European Conference, ECSA 2017, Canterbury, UK, September 11-15, 2017, Proceedings 11*, pages 67–83. Springer, 2017. 13, 14, 39

[13] International Organization for Standardization. ISO 5807: Information processing — documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts. 1985. ix, 26

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995. 56

[15] Pablo Gómez-Abajo, Esther Guerra, Juan de Lara, and Mercedes G Merayo. Mutation testing for DSLs (tool demo). In *Proceedings of the 17th ACM SIGPLAN International Workshop on Domain-Specific Modeling*, pages 60–62, 2019. 69

[16] Timo Greifenberg, Klaus Müller, and Bernhard Rumpe. Architectural Consistency Checking in Plugin-Based Software Systems. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, pages 1–7, September 2015. 13, 14, 39

[17] IEEE Architecture Working Group et al. IEEE Recommended Practice for Architectural Description for Software-Intensive Systems. *IEEE std*, 1471, 2000. 1, 5

[18] Alessandro Gurgel, Isela Macia, Alessandro Garcia, Arndt von Staa, Mira Mezini, Michael Eichberg, and Ralf Mitschke. Blending and reusing rules for architectural degradation prevention. In *Proceedings of the 13th International Conference on Modularity*, pages 61–72, New York, NY, USA, April 2014. Association for Computing Machinery. 14, 16, 39

[19] Thomas Haitzer and Uwe Zdun. Semi-automated architectural abstraction specifications for supporting software evolution. *Science of Computer Programming*, 90:135–160, September 2014. 13, 14, 39

[20] Salima Hassaine, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Giuliano Antoniol. ADvISE: Architectural Decay in Software Evolution. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 267–276, March 2012. 13

[21] Mark Hills, Paul Klint, and Jurgen J Vinju. Scripting a refactoring with rascal and eclipse. In *Proceedings of the Fifth Workshop on Refactoring Tools*, pages 40–49, 2012. 15

[22] Paul Klint, Tijs van der Storm, and Jurgen 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, Edmonton, AB, September 2009. IEEE. xiii, 8

[23] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Easy meta-programming with rascal. leveraging the extract-analyze-synthesize paradigm for meta-programming. In *Proceedings of the 3rd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'09)*, LNCS. Springer, 2010. ix, 8, 15, 21

[24] Jens Knodel and Daniel Popescu. A comparison of static architecture compliance checking approaches. In *2007 Working IEEE/IFIP conference on software architecture (WICSA'07)*, pages 12–12. IEEE, 2007. 69

[25] Ruiyin Li, Peng Liang, Mohamed Soliman, and Paris Avgeriou. Understanding software architecture erosion: A systematic mapping study. *Journal of Software: Evolution and Process*, 34(3):e2423, 2022. 5

[26] Yannis Lilis and Anthony Savidis. A Survey of Metaprogramming Languages. *ACM Computing Surveys*, 52(6):1–39, November 2020. 8

[27] Tianyu Liu. Automatic Code Modernization with Rascal. Master's thesis, Eindhoven University of Technology, 2018. 2, 9, 15, 16, 39

[28] Ken Martin and Bill Hoffman. *Mastering CMake: Version 3.1*. Kitware Incorporated, 2015. 24

[29] Vinicius R. L. Mendonca, C. L. Rodrigues, Fabrízzio Soares, and A. Vincenzi. Static Analysis Techniques and Tools: A Systematic Mapping Study. In *International Conference on Software Engineering Advances*, October 2013. 7

[30] Gail Murphy, David Notkin, and Kevin Sullivan. "Software Reflexion Models: Bridging the Gap between Design and Implementation". *Software Engineering, IEEE Transactions on*, 27:364–380, May 2001. 13

[31] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019. 58

[32] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992. 1, 5

[33] Mehwish Riaz, Muhammad Sulayman, and Husnain Naqvi. Architectural Decay during Continuous Software Evolution and Impact of 'Design for Change' on Software Architecture. In *Advances in Software Engineering*, pages 119–126, Berlin, Heidelberg, 2009. Springer. 5

[34] Henrique Rocha, Rafael Serapilha Durelli, Ricardo Terra, Sândalo Bessa, and Marco Túlio Valente. DCL 2.0: Modular and reusable specification of architectural constraints. *Journal of the Brazilian Computer Society*, 23(1):12, August 2017. ix, 14, 16, 39

[35] Mathijs Schuts, Rodin Aarssen, Paul Tielemans, and Jurgen Vinju. Large-scale semi-automated migration of legacy C/C++ test code. *Software: Practice and Experience*, 52, March 2022. 9, 15, 16, 39

[36] Richard N Taylor, Nenad Medvidović, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 2010. TMD. 5

[37] Jilles van Gurp and Jan Bosch. Design erosion: Problems and causes. *Journal of Systems and Software*, 61(2):105–119, March 2002. 5

[38] Rainer Weinreich and Georg Buchgeher. Automatic reference architecture conformance checking for soa-based software systems. In *2014 IEEE/IFIP Conference on Software Architecture*, pages 95–104. IEEE, 2014. 69

Appendix A

Description of M3 Model Fields

Field	Data Structure	Description
callGraph	rel[loc caller, loc callee]	Mapping from the location where the function is called (caller) to the location where the function is provided (callee). Note that the term “function” in the C++ M3 model refers to C++ functions, methods and constructors.
uses	rel[loc src, loc name]	Mapping from the physical location where the artifact is used to the logical name of the artifact. Note that "artifact" refers to any declared C and C++ artifact like a compilation unit, a package, a class, a method, a function, a parameter, a field or a local variable, etc..
partOf	rel[loc decl, loc file]	Mapping from the logical name of the artifact to the physical directory of the source file in which the artifact is used.
containment	rel[loc from, loc to]	Mapping from the logical name of the container, such as class, function, method, to the logical name of the artifact that is contained, such as method, fields and parameter.
declarations	rel[loc name, loc src]	Mapping from the logical name of the artifact to the location where the artifact is declared.
implicitDeclarations	set[loc]	A set of the implicit declarations of the C++ `new` operator.
declarationToDefinition	rel[loc decl, loc impl]	Mapping from the physical location of the function/method declaration to the physical location where it is defined.

functionDefinitions	rel[loc name, loc src]	Mapping from the logical name of the function to the physical location where it is declared.
methodInvocations	rel[loc caller, loc callee]	Mapping from the logical name of the method (caller) requiring another method to the logical name of the required method (callee).
methodOverrides	rel[loc base, loc override]	Mapping from the logical name of the function defined in the base class to the logical name of the function in the derived class which overrides the function in the base class.
macroExpansions	rel[loc file, loc macro]	Mapping from the location where a macro is expanded to the logical location of the macro.
macroDefinitions	rel[loc macro, loc src]	Mapping from the logical name of the macro to the physical location where the macro is defined.
requires	rel[loc includer, loc includee]	Mapping from the location of the source file (includer) to the location of the header file (includee) included in the source file.
includeResolution	rel[loc directive, loc resolved]	Mapping from the logical name of the header to the physical location where the header locates.
includeDirectives	rel[loc directive, loc occurrence]	Mapping from the logical name of the header to the physical location where it is included.
inactiveIncludes	rel[loc directive, loc occurrence]	Mapping from the logical name of the unused header to the physical location where it is included.
unresolvedIncludes	rel[loc directive, loc occurrence]	Mapping from the logical name of the unresolved header to the physical location where it is included. A include is unresolved if its include path is not found during model generation.
extends	rel[loc directive, loc occurrence]	Mapping from the logical name of the parent class to the logical name of the child class.
declaredType	rel[loc decl,TypeSymbol typ]	Mapping from the logical name of an artifact to the corresponding part in the abstract syntax tree where it is declared.

modifiers	rel[loc directive, loc occurrence]	Mapping from the logical name of the artifacts, including field, parameter, method, etc. to its modifier, such as `public`, `virtual`, `const`, etc.
memberAccessModifiers	rel[loc directive, loc occurrence]	Mapping from the logical name of a method or field or constructor to its member access modifier, including `public` and `private`.
comments	list[loc]	A list of physical locations of the comments in the source code.

Appendix B

List of Architectural Rules

Following is the description of all the architectural rules. As the first three rules have been discussed in Section 4.3, this appendix mainly focuses on the remaining ones. At the end of each rule, we use the letter **S** and **G** to denote whether the rule is specific to the investigated codebase or generic to different software.

RULE 1 Each layer must only depend on the other allowed layers. (**G**)

RULE 2 The communication between the layers must be conducted exclusively through the interfaces defined in `pub` folder. (**G**)

RULE 3 It is not allowed to allocate dynamic memory after the startup phase of the applications. (**S**)

RULE 4 Deadlock must be avoided in remote procedure calls (RPC). (**G**)
Deadlock happens when a process requests resources from a remote process while the remote process also waits for the resource provided by the first process. When both processes are blocked, severe problems in the system operation arise, leading the system to become unresponsible and inefficient. Therefore, it is necessary to set a rule to detect the circular dependency between the processes and prevent deadlock in RPCs from happening.

RULE 5 The C++ built-in function `Sleep` is not allowed to be used in any context except in `TOS_p_TSK_sleep`. (**S**)
`Sleep` function can suspend a thread or process for a specified time interval. The function name and type of the parameter vary across different operating systems. In the case of positioning software, the system is executable in both Windows and VxWorks. To make the program compatible with different operating systems, a wrapper function `TOS_p_TSK_sleep` is implemented, which provides a uniform interface for different operating systems and makes the software more maintainable. Thus, instead of `Sleep`, this wrapper is expected to use anytime a thread or process needs to be suspended.

RULE 6 Every time a new class is created in the production code, a test class must be implemented to test it. (**G**)
The rationale of this rule is to ensure the functionality and quality of the implemented segments in production code. It is necessary to have a test class for the production class to detect bugs and unexpected behaviors before the code is deployed to the production environment. Compared with fixing the problems in production code after the delivery, it is more cost-effective to improve the implementations during the development phase.

RULE 7 It is not allowed to create new threads in the application execution phase. (**S**)
Essentially, creating a new thread involves dynamic memory allocation, which cannot happen during the execution phase, according to RULE 3. Therefore, thread creation is only allowed during the startup phase and must be avoided during the execution phase.

RULE 8 The newly implemented part of the codebase is not allowed to use the utilities specifically designed for the legacy code. (**G**)
As the software evolves, the programs implemented with C are migrated to C++. During this evolution, the software is characterized by two part: the new implementation and the legacy code that is or will be deprecated. That means the utilities designed for legacy code will be gradually disposed of and cannot be included in the new implementations.

RULE 9 If the same functionality is implemented in new and legacy code, always use the new methods. (**G**)
The newly implemented part of the codebase contains refactoring and rewriting of the functionalities implemented in the legacy part. As the legacy code will be eliminated eventually, future implementations must be based on the new functions or methods; otherwise, the software becomes difficult to maintain.

RULE 10 Timeouts on the connections must be avoided. (**G**)
Timeouts can cause unexpected problems during the operation of the system. Assume a connection timeout is set for the connecting process between the software and a system component. In case of a system component crash, the software starts timing until the timeout is reached. However, it is possible that the system component finishes restarting after the timeout, but the software does not try to reconnect it anymore after a timeout. This timeout mechanism is not reliable and needs to be avoided. A better solution is to check the availability of the service by polling constantly.

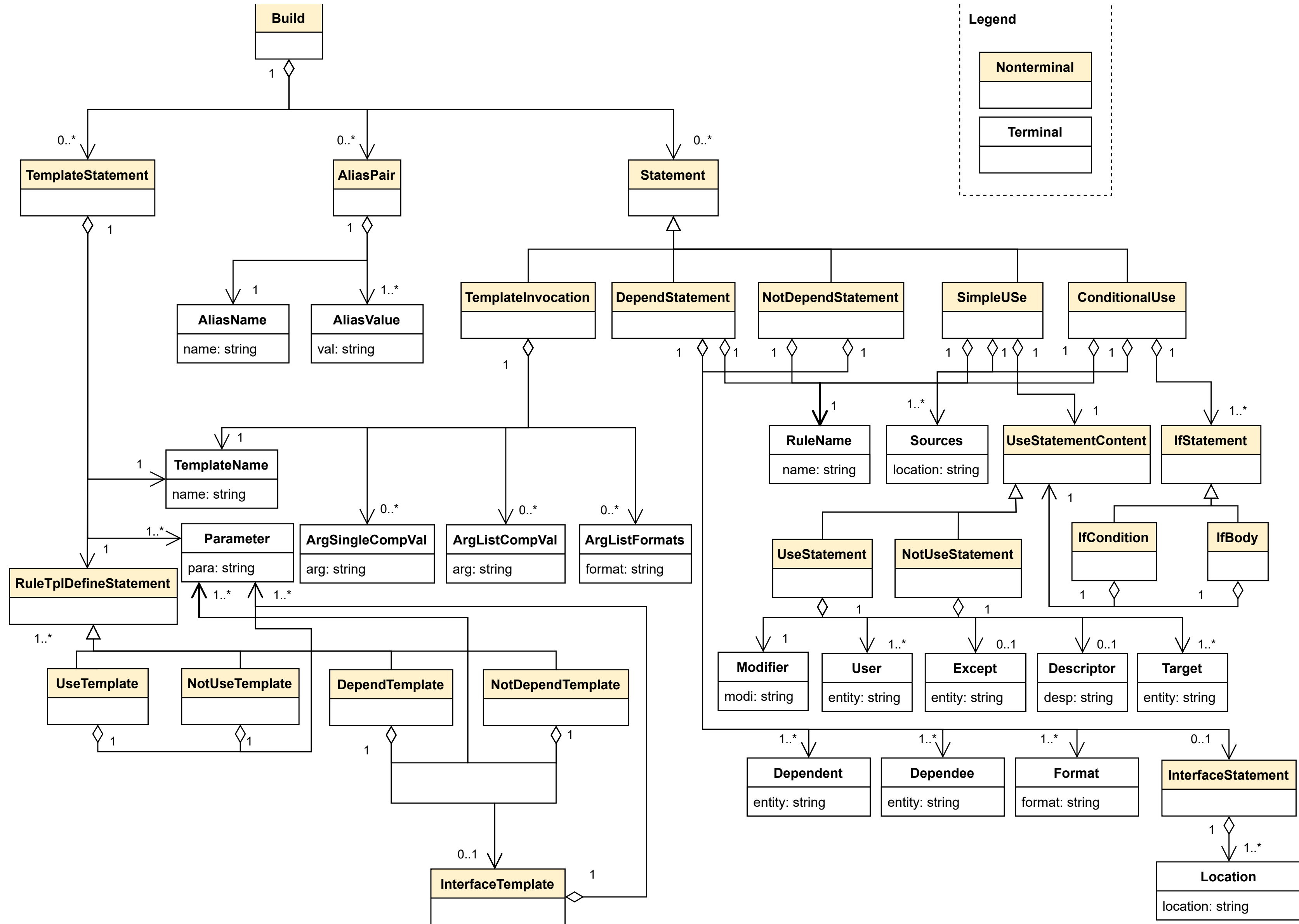
RULE 11 All the operations requiring OS primitives should use the implemented wrappers. (**G**)
In the software, an abstracted layer for OS primitives, such as read and write, is implemented to provide uniform interfaces for function invocation, disregarding which OS the software runs on. This abstraction promotes the maintainability of the software. Thus, instead of OS primitives, the wrappers must be utilized for manipulations on OS.

RULE 12 The calls on the component for interprocess communication should be via wrappers rather than the native functions. (**S**)
Similar to OS primitives, the software contains an abstracted layer implemented for the functions defined in the interprocess communication component. These wrappers are expected to use for better code maintainability.

RULE 13 Hardcoded passwords are not allowed. (**G**)
Some services in the codebase require a password for login authentication. Although embedding the plain text passwords into source code is a practical measure, it exposes the system to the threat of password guessing attacks, resulting in bypassing the authentication process. Therefore, this rule is proposed to minimize and eliminate the usage of hardcoded passwords.

Appendix C

Structure of DSL Syntax



Build
TemplateStatement
AliasPair
Statement
Legend
Nonterminal
Terminal
AliasName
name: string
AliasValue
val: string
TemplateInvocation
DependStatement
NotDependStatement
SimpleUSe
ConditionalUse
TemplateName
name: string
Parameter
para: string
ArgSingleCompVal
arg: string
ArgListCompVal
arg: string
ArgListFormats
format: string
RuleName
name: string
Sources
location: string
UseStatementContent
IfStatement
RuleTplDefineStatement
UseStatement
NotUseStatement
IfCondition
IfBody
UseTemplate
NotUseTemplate
DependTemplate
NotDependTemplate
Modifier
modi: string
User
entity: string
Except
entity: string
Descriptor
desp: string
Target
entity: string
Dependent
entity: string
Dependee
entity: string
Format
format: string
InterfaceStatement
InterfaceTemplate
Location
location: string
0..*
1
1..*
0..1