Static Versus Dynamic Polymorphism When Implementing Variability in C++

by

Samer AL Masri

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

© Samer AL Masri, 2018

Abstract

Software Product Line Engineering (SPLE) creates configurable platforms that can be used to efficiently produce similar, and yet different, product variants. To implement SPLs, multiple variability implementation mechanisms have been suggested, including polymorphism. In this thesis, we talk about the trade-offs of using static versus dynamic polymorphism through a case study of IBM's open-source Eclipse OMR project.

Eclipse OMR is an open-source *C++* framework for building robust language runtimes. To support the diverse languages and architectures targeted by the framework, OMR's variability implementation uses a combination of build-system variability and static polymorphism. OMR developers now realize that their current static polymorphism implementation has its drawbacks and are considering using dynamic polymorphism instead.

In order to study the trade-offs of using different kinds of polymorphism in OMR, it is crucial to collect function information and overload/override statistics about the current code base. Hence, we create OMRStatistics, a static analysis tool that collects such information about OMR's source code. Using the information provided by OMRStatistics, OMR developers can make better design decisions on which variability extension points should be switched from static polymorphism to dynamic polymorphism.

In addition, we report on our first hand experience of changing the polymorphism used in OMR's variability implementation mechanism from static to dynamic, the challenges we faced in the process, and how we overcame them.

Preface

Parts of this thesis have been previously submitted in the *28th Annual International Conference on Computer Science and Software Engineering* in 2017 and *The 22nd International Systems and Software Product Line Conference* in 2018.

Eclipse OMR is the inspiration of this work since the work. Hence, under my supervisor's, Sarah Nadi, guidance and supervision, I was frequently delivering my work to our IBM CAS collaborators (Xiaoli Liang, Robert Young, and Matthew Gaudet) who constantly gave me feedback and guidance on how to proceed with the project.

It is also important to note here that this work is based on a hypothesis originally suggested by Matthew Gaudet.

In addition, Nazim Bhuiyan, who is currently interning at IBM, helped me run the benchmarks on the server since we did not have access to the benchmark software.

Acknowledgements

First, I would like to thank my supervisor Professor Sarah Nadi for her guidance and leadership through out this research. At some points during our research, the challenges we faced were more complex than I've expected and it seemed to me like there is no practical solution for the obstacles we're facing. At these moments mostly, I valued her support and unshakened determination to find solutions. Compared to when I started working with Sarah, I've developed in multiple aspects, including building up more determination in face of hard challenges and being able to shuffle multiple tasks efficienctly at times of pressure. I am happy to say that Sarah helped providing the perfect environment for such development.

I would also like to acknowledge the Nazim, Matthew, Xiaoli, and Robert for offering me advice and guidance throughout this research. Co-authoring and publishing papers together was a great experience.

On another note, I would like to express my gratefulness for my parents and friends who supported me throughout my research. I've always found motivation in being around them. They were one of the main reasons I get through my days.

Finally, I would like to thank the IBM CAS program for funding this project.

Contents

1 Introduction **1**
1.1 Variability Implementation Mechanisms 2
1.2 Theses Focus 3
1.3 Eclipse OMR and its Variability Implementation 4
1.4 Objective 5
1.5 Contributions 5
1.5.1 Contribution 1: Documenting OMR's Variability Implementation Mechanism 5
1.5.2 Contribution 2: Comparing Static vs Dynamic Polymorphism 6
1.5.3 Contribution 3: Changing OMR's Variability Implementation to Dynamic Polymorphism 7
1.5.4 Recap 7
1.6 List of Publications 8

2 Background **9**
2.1 Basic Definitions 9
2.2 Variability Binding Time 11
2.3 Static Polymorphism vs Dynamic Polymorphism in *C++* . . . 12
2.4 LLVM and Clang's Relation to OMRStatistics 13
2.5 Eclipse OMR 14
2.5.1 Project History 15
2.5.2 Project Structure and Size 16

3 Eclipse OMR Variability Implementation Mechanism **17**
3.1 Directory Structure 18
3.2 Inheritance 20
3.3 Connector Classes 20
3.3.1 Motivation for Creating Connector Classes 20
3.3.2 Connectors Classes Description 21
3.3.3 Connector Classes Implementation 22
3.4 Include Paths 23
3.5 `#ifdef` Directives 23

4 Static Vs Dynamic Polymorphism **25**
4.1 The `self()` Function 25
4.2 Implications of `self()` 28
4.2.1 Advantage: Less Runtime Cost 28
4.2.2 Consequence: Irregularities 29
4.2.3 Consequence: *OMRChecker* 29
4.2.4 Consequence: No Clear API 29
4.3 Recap of the Consequences of OMR's Current Variability Implementation Mechanism 30

4.4 Moving to Dynamic Polymorphism 30

5 OMRStatistics 32
5.1 Features 32
5.2 Setup 33
5.3 OMRStatistics Implementation Mechanism 34
5.3.1 Mechanism Overview - The Algorithm 34
5.3.2 Phase 1: Collecting Information 35
5.3.3 Phase 2: Creation of Class Hierarchies 35
5.3.4 Phase 3: Method Information Collection 38
5.4 OMRStatistics System Implementation 40
5.4.1 Step 1: Find the Source Files 41
5.4.2 Step 2: Run OMRStatistics on the Source Files and Produce Visualizations 41
5.4.3 Step 3: Creating Database 44
5.4.4 Recap and Implications 45
5.5 Use Cases 45
5.5.1 Use Case 1: Find All Overridden Functions 46
5.5.2 Use Case 2: Find All Functions in a Class 46
5.5.3 Use Case 3: Find a Location of the Definition of a Function Signature 47
5.5.4 Use Case 4: View the Class Hierarchy 48

6 Assessing Dynamic Polymorphism 49
6.1 The Queries 50
6.2 Data About Extensible Classes 52
6.3 Data About Overridden Functions 52
6.4 Data About Extensible Class Hierarchies 53

7 The Road to Virtualization in OMR 56
7.1 Convincing the OMR Team 56
7.2 The Process of Virtualization 58
7.3 Manual Virtualization Helper (MVH) 59
7.3.1 Overview 60
7.3.2 Process 60
7.3.3 Configuration 61
7.4 Challenges 61
7.4.1 Picking the Right Commit 62
7.4.2 Finding the problematic functions 62
7.4.3 Problem 1: Functions with no Implementations 63
7.4.4 Problem 2: Functions called in constructors 64
7.4.5 Problem 3: Submitting PRs 67
7.4.6 Problem 6: Infrastructure Issues 67
7.5 Benchmarking 68
7.5.1 Setup 68
7.5.2 Results 69

8 Discussion 75
8.1 Answering the Research Questions 76
8.2 Implementation that Best Fits Eclipse OMR 78
8.3 Open Questions 80
8.4 Threats to Validity 81
8.4.1 Threats to External Validity 81
8.4.2 Threats to Construct Validity 82

9 Related Works **85**
9.1 Variability Implementation Mechanisms Discussed in the Literature . . . 85
9.2 Variability Implementation Mechanisms in Practice . . . 86
9.3 Tools Supporting Variability . . . 87
9.3.1 Variability Aware Analysis . . . 87
9.3.2 Other Tools Supporting Variability . . . 88
9.4 Variability Metrics . . . 89
9.5 Software Product Line Management . . . 89
9.5.1 Recap . . . 90

10 Conclusion **92**

References **95**

List of Figures

2.1 Comparison between dynamic polymorphism and CRTP. Figure (a) shows a UML example of static polymorphism. Figure (b) shows a UML example of dynamic polymorphism. Figures (c) and (d) show the a UML and its implementation respectively of a CRTP example 12

3.1 Directory structure of the compiler component 18
3.2 The `CodeGenerator` hierarchy. Classes in italic represent abstract classes. 19

4.1 Example explaining the need for the `self()` function 26
4.2 Example of an irregular case that becomes legal when using `self` 28

5.1 Example UML explaining `getLinearHierarchy` function . . . 39
5.2 Hierarchy visualization file 43
5.3 Part of the overrides visualizations web page. Originally, all nodes had the same font-size and underlined. Green nodes represent virtual functions and red nodes represent implicit functions. However, in order to make this figure compatible with black-and-white printing, we made only virtual functions underlined, and implicit functions are italicized and have larger fonts. Pressing the button on top hides the implicit functions. 43
5.4 OMRStatistics relational database schema 44
5.5 The results from the database when running the query in 5.4 to find all overridden functions 46
5.6 The results from the database when running the query in 5.5 to find all functions in a class 47
5.7 The results from the database when running the query in 5.6 to find a location of the definition of a function signature 47
5.8 Part of the *graph.pdf* when viewing the hierarchy visualization in the fourth use case 48

7.1 Visualization of `buildRegisterMapForInstruction` function that is declared but has no implementation in `OMR::CodeGenerator` 63
7.2 UML of an example of a call of a virtual function in a `CodeGenerator` class constructor 65
7.3 The Effects of Changing the Variability Implementation Mechanism on the Throughput in the DayTader3 Benchmark - LargeThreadPool Configuration 70
7.4 The Effects of Changing the Variability Implementation Mechanism on the Throughput in the DayTader3 Benchmark - LargeThreadPoolWarm Configuration 71

7.5 The Effects of Changing the Variability Implementation Mechanism on the Throughput in the ODM Benchmark - Seg300 Ruleset 72
7.6 The Effects of Changing the Variability Implementation Mechanism on the Throughput in the ODM Benchmark - Seg5 Ruleset 73

Chapter 1

Introduction

As automation becomes more pervasive, our dependency on software is rapidly increasing which is causing an abundance of customers for software companies. Customers have a diverse spectrum of problems and expect companies to provide software solutions. To accommodate various requirements, software developers started leveraging the similarity in solutions between different problems to rapidly develop corresponding software products. Since they are created to solve similar problems, these products would act and behave very similarly; their behavior would only differ in a few circumstances to accommodate the slight difference between the problems being solved. For example, device drivers nowadays are made to support multiple operating systems and multiple architectures. Hence, a device driver would behave slightly differently to accommodate the difference imposed by the operating system application user interface (API) or the hardware architecture instructions. In that sense, a unique product is installed on each combination of operating system and hardware architecture. However, these products are very similar and categorized under the same software family name. When a software is developed to be configured and act differently, it is called a software product line. A *Software Product Line* (SPL) is a set of software products, developed from a common source base, that share multiple features but also differ in some aiming to satisfy a market segment [4]. An SPL is built while prioritizing flexibility and the source code is developed to be reused when building different products. A software feature in an SPL is defined as a particular functionality in a software

system [61].

The advantage of SPLs is that developers can use one code base for multiple features. Hence, multiple customers only have to download one code base (or installation executable) to build (or install) different products that solve different problems. For example, when installing *TeamViewer* [94], the user is asked to select the features to be installed; features such as *remote print* to *TeamViewer VPN*. Based on the selected features, the same executable resource is used to install a different product of the TeamViewer software family. However, this advantage to customers and a company as a whole can be a disadvantage for developers. When adding features to a product line, testing the product line becomes harder. Having n features in a product line implies that there are 2^n possible combinations of features, or product outcomes; hence adding features to the product line means an exponential increase in the job of the testing and other analysis teams.

These difficulties inspired researchers to invent various tools that support variability [18]. Some focused on modeling the features [7], [84], others focused on supporting development in a multi-feature environment [11], [28]. Some tools helped to fix the problem mentioned above by aiming to analyze SPLs in a more efficient way. An example of such a tool is Typechef [55]. Typechef aims to analyze all variants of a configurable software implemented using *C* and the *C* preprocessor, which implies that Typechef only works on a specific variability implementation mechanism. This brings up a broader question: how is variability implemented? And how does the implementation affect the way it's analyzed.

1.1 Variability Implementation Mechanisms

Developers implement variability in various ways. Arguably, the most popular variability implementation is using the *C* preprocessors's `#ifdef` directive [63], which is, for example, used extensively in the Linux kernel [75]. Another approach is parametrization, where parameters control built features; *MADAM* project is a practical example of a parametrization variability implementa-

tion [39]. *Mozilla Firefox* uses frameworks to implement variability; that implementation is used in Firefox to support plugins, where users can create a plugin software independently without having to understand the firefox code base [45]. We elaborate and discuss more implementation mechanisms in the Related Works chapter (Chapter 9). While there are many variability implementations, most of the literature that studied variability in practice focused on the approach that uses preprocessors for conditional compilation and tools that support that approach (`#ifdef`) [52], [53], [66], [74], [81], [88], [92], [95], [104]. Exploring other variability implementation mechanisms that are used in practice and understanding the challenges they impose on family-based analyses is important to drive SPL research forward and ensure technology transfer.

1.2 Theses Focus

One interesting variability implementation that we study throughout this thesis is polymorphism. Polymorphism means creating many forms of something; which matches the idea of variability in itself [25]. Hence, it makes sense for polymorphism to be commonly utilized when implementing variability in software. The implementation has been discussed in the literature under different categories, such as template method polymorphism [8] and object-oriented frameworks [54]. However, in practice, developers may be using polymorphism in variability implementations in ways that do not perfectly align with the mechanisms discussed in the literature. In *C++*, polymorphism can be implemented through two types: static polymorphism, where function calls are resolved at compile time by the compiler or (2) at runtime where calls are resolved at runtime with the help of a data structure called *virtual method table*(vtable). We discuss both types of polymorphism more later in our thesis (Section 2.3). In this thesis, we study the trade-offs of using different types of polymorphism in variability by looking at a case study: *Eclipse OMR*.

1.3 Eclipse OMR and its Variability Implementation

Eclipse OMR is an industrial case study of an SPL that uses an uncommon combination of variability implementation mechanisms which rely heavily on static polymorphism. OMR is an open-source framework for building language runtimes [33]. It provides the building blocks for just-in-time compilers, garbage collectors and more, each of which can be customized to a target language. The cross-platform components also support multiple operating systems and target architectures: *X86* (*AMD64* and *I386*), *Power*, *Z*, and *ARM* (under development). OMR has already been used in language runtimes for *Java* (in production), as well as with *Ruby*, *Python*, and *Lua* experimentally. As a result, all consumers of the framework can be described as products or variants of the OMR software product line. Product variability can be the result of changing the target language for which OMR components are used, changing the target architecture of the resulting language runtime, or both. It is important to note that we refer to OMR as a library of language runtime components however some of its components can be used outside the scope of a product runtime. OMR's compiler component, for example, can be used as a just-in-time compiler, which is considered part of the language runtime. However, it can also be used to compile code before running it, which is outside the scope of runtime.

The OMR framework's extension model is based on build system variability and static polymorphism. The OMR compiler, a core component of OMR and the main focuses in this thesis, is built in an object-oriented manner where variability is injected through the class hierarchy. A high-level component may have specializations both for the target architecture and language, and the specializations are contained in named directories. OMR creates a class hierarchy for each high-level component and uses static polymorphism where all type resolution for the objects that vary according to language or architecture happens at compile time. The resolution is guided by the selection of directories in the build system to compose a given variant of the product line.

1.4 Objective

The objective of our research is to answer the following questions: What are the practical implications of using static polymorphism versus dynamic polymorphism? Are there any consequences for changing the variability implementation mechanism that use static polymorphism to use dynamic polymorphism?

In order to answer that, we consider changing a relatively large *C++* project's (OMR) variability implementation mechanism from using static polymorphism to dynamic polymorphism, in three milestones:

1. Explore and understand OMR's variability implementation mechanism.
2. Study the trade-offs of using static vs dynamic polymorphism in OMR's variability implementation mechanism and judge which is a better fit for the project.
3. If dynamic polymorphism was deemed fit, upgrade OMR's source code to use dynamic polymorphism in its variability implementation mechanism and document our first-hand experience on that.

The above milestones will be referred to in the rest of the thesis to help the reader understand how they were accomplished.

1.5 Contributions

While conducting our research to achieve the stated objective

1.5.1 Contribution 1: Documenting OMR's Variability Implementation Mechanism

In order to achieve our first milestone, we dug into the OMR source code to understand its variability implementation mechanism. Since reading and understanding the source code was challenging, we documented our first-hand experience in understanding OMR's variability implementation mechanisim in a paper publication [69]. OMR is intended to be used by foreign developers

to complement their language development which makes readability and simplicity of the source base a priority. As the project has a shortage in that area, our documentation would be helpful for (1) OMR developers to have a big picture of the variability implementation mechanism in their project, (2) new developers who plan to contribute to the OMR source base, and (3) researchers who would like to conduct case studies on *C++* projects that use related variability implementation mechanisms.

1.5.2 Contribution 2: Comparing Static vs Dynamic Polymorphism

When considering the second milestone, we first have to define the meaning of "better fit for the project". As discussed in Section 1.5.1, readability of the code is a priority. However, when looking at the project in the context of being a language runtime library, the runtime performance is also a priority. Hence, our meaning of "better fit for the project" is the solution that provides the best trade-off between readability (or simplicity of the code) and runtime performance.

In order to decide on the variability implementation mechanism that better fits the project, we create a static analysis tool, OMRStatistics, that is tailored to work on the variability implementation mechanism in OMR. It collects statistics about the code (such as class names, functions in each class, overridden functions...) and stores the information in a database. After that, we used the outputted database to identify overridden functions and gather relevant statistics for developers.

OMRStatistics has two main implications on OMR developers and contributors:

1. It identifies the overridden functions that should be virtualized when switching to dynamic polymorphism

2. It helps developers understand the OMR architecture better and identify strengths and weaknesses

It is important to note that although OMRStatistics is tailored to analyze OMR's variability implementation mechanism, it can be used on any other *C++* project that uses a similar variability implementation mechanism to collect information about the code and store them in a database.

1.5.3 Contribution 3: Changing OMR's Variability Implementation to Dynamic Polymorphism

After collecting information and judging that it is better fit for the project to change its variability implementation mechanism to use dynamic polymorphism, we virtualized the overridden functions to use dynamic polymorphism (adding the `virtual` keyword to the function declaration) and benchmarked the project to make sure there was minimal degradation in performance. While virtualizing the functions, we ran into multiple problems that revealed practical pros and cons of using static polymorphism. We communicated such challenges to OMR developers and elaborate on them in this thesis.

1.5.4 Recap

To recap, the contributions of this thesis:

- Documented OMR's variability implementation mechanism
- Created a tool that analyzes variability in OMR and projects that have similar variability implementation mechanisms (use static polymorphism).
- Partially changed a project with a large code base (Eclipse OMR) from using static polymorphism to dynamic polymorphism, submitted pull requests for the suggested changes, and benchmarked the project with the new changes to measure the impact on runtime performance.
- Documented the benefits and costs of changing OMR's variability implementation mechanism to use dynamic polymorphism instead of static polymorphism

The above contributions will be referred to in the rest of the thesis to help the reader understand how they were achieved.

1.6 List of Publications

Some chapters of this thesis are based on the following two publications:

- Samer AL Masri, Nazim Bhuiyan, Sarah Nadi, and Matthew Gaudet. "Software Variability Through *C++* Static Polymorphism: A Case Study of Challenges and Open Problems in Eclipse OMR." In Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering (CASCON 2017). IBM Corp., Riverton, NJ, USA, 285–291. http://dl.acm.org/citation.cfm?id=3172795.3172831

- Samer AL Masri, Sarah Nadi, Xiaoli Liang, Robert W. Young, and Matthew Gaudet. Using Static Analysis to Support Variability Implementation Decisions in *C++*. In Proceedings of the 22nd International Conference on Systems and Software Product Line - Volume 1. ACM, New York, NY, USA, p. 236-245. https://dl.acm.org/citation.cfm?id=3233043

Chapter 2

Background

In this chapter, we present multiple concepts and ideas that are necessary to understand the rest of this thesis. More specifically, we talk about (1) the different feature binding times of different variability implementations, (2) how different types of polymorphism are implemented under the hood, (3) the clang front-end compiler modularity in order to implement OMRStatistics, and (4) the background and history from where the Eclipse OMR project is coming from to provide the reader with more context. We now summarize what we believe are the fundamental concepts on which our research is built.

2.1 Basic Definitions

While describing the research in this thesis, we use multiple scientific terms related to software engineering in general and software variability in specific. In the following, we explicitly define all terms that we use in this thesis:

1. *Software variability*: According to Bosch, software variability is the capability of a software system to change its behavior in a particular context [19]. In that sense, a software system is considered to be variable if it can alter its behavior when having different configurations or put in different environments.

2. *Software product line (SPL)*: An abstract product that is used to produce multiple other products depending on the customer needs [79]. It is important to note here that all SPLs are variable software but not all

variable software are SPLs. Variability can appear in any software that changes its behavior in different contexts. For instance, a variable software can consist of multiple products that are not related in definition or functionality, but they share part of their codebase. On the other hand, SPL products must be related in terms of functionality

3. *Variability implementation mechanism*: For a software to be variable, it must have some mechanism or framework that specifies how the different products share the code base and its resources. Such mechanisms are called *variability implementation mechanisms*. Examples of variability implementation mechanisms have been discussed in the introduction (*C* preprocessors's *#ifdef*, parametrization...). We provide a literature review of the most common variability implementation mechanisms in Chapter 9.

4. *Feature*: describes a particular behavior of a system. A feature can be functional or non-functional [17]. For example, requiring a system to display the number of empty classrooms on a campus is a functional feature. Whereas requiring a system to display that number quickly is a non-functional requirement. In our research, however, we mainly focus on functional features in Eclipse OMR.

5. *Abstract Syntax Tree (AST)*: it is the output of the semantic analysis phase during compilation. An AST can be considered a way of communication between the front-end compiler and the optimizer and backend compiler [26]. The three phases are discussed more later in this chapter (Section 2.4).

6. *Application Programming Interface (API)*: An API is a way for software to communicate with its clients. As described by Brail et al. [103], it is a *contract* between a software and its client developers; client developers rely on such contracts to guarantee ways to access the services of the provider software. When taken into Eclipse OMR's context, since OMR uses static polymorphism, the API is the polymorphic functions that

client developers need to use or extend to connect their language to the runtime library.

2.2 Variability Binding Time

Determining which features are included in a product variant can be implemented at different stages of the program life cycle. According to Apel et al., feature selection can happen when the program is built (*build-time variability*), at the time when the program starts and is loaded to the memory (*load-time variability*), or while the program is being executed (*run-time variability*) [8]. As pointed out by Apel et al., some researches distinguish even more binding times [86]. However, for our discussion, the above binding times suffice. An example of build-time variability is using the conditional *C* preprocessor directives (`ifdef`) to include different features based on parameters passed to the compiler [31]. The Linux kernel mainly uses build-time variability, and currently has more than 14,000 configuration options implemented as macros [38]. With load-time variability, a software system is composed of different parts that are compiled and ready to be loaded to memory; based on the defined configurations, the right binaries would be loaded to memory and the program executes [8]. *NanoMail* is an email client SPL that uses load-time variability. The client integrates different email functionalities. When the program starts, depending on the functionalities specified in the configuration by the user, the right binaries are loaded to memory and the software executes accordingly [87]. Run-time variability is when the software can instantly change its behavior at runtime as a reaction to a change in its configuration or surrounding environment. An example of such variability is Mozilla Firefox's Add-ons. It is possible to install an add-on to the browser and the functionality will be added instantaneously (no restart needed). Variability implementation mechanisms bind the components of the software at different times, and each binding time has its pros and cons. More details about the trade-offs of different binding times are described in Coplien's book [24]. When implementing variability using static polymorphism in *C++*, the right functionality or im-

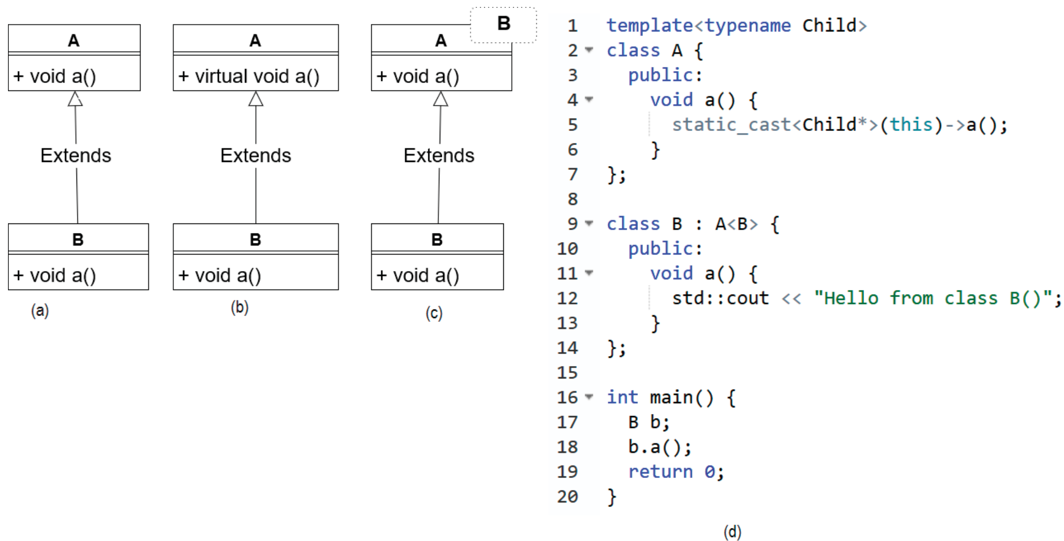

Figure 2.1: Comparison between dynamic polymorphism and CRTP. Figure (a) shows a UML example of static polymorphism. Figure (b) shows a UML example of dynamic polymorphism. Figures (c) and (d) show the a UML and its implementation respectively of a CRTP example

plementation of a function is linked to the definition of a function at compile time. On the other hand, *C++*'s implementation of dynamic polymorphism links the implementation of a function to the call at run-time. Hence, deciding on using static or dynamic polymorphism in OMR's variability implementation mechanism affects the binding time of features in the project.

2.3 Static Polymorphism vs Dynamic Polymorphism in *C++*

Polymorphism is commonly used in object-oriented languages to enable access to multiple related behaviors, and can be used to implement variability [8]. Polymorphism can be viewed as a variability implementation mechanism and is therefore useful to use in the context of SPLs [8].

In *C++*, polymorphism is by default implemented statically where function calls are resolved and bounded to their implementation at compile time. This is referred to as *Static Polymorphism*. Static polymorphism is usually associated with the Curious Recurring Template Pattern (CRTP) [16]. CRTP exploits template classes and static casting to implement inheritance.

On the other hand, it is also possible to bind functions to their implementation dynamically using the *virtual* keyword in *C++*; this is called *dynamic polymorphism*. Since dynamic polymorphism one of the main features used by object-oriented language programmers, its use is increasing and expected to increase over time [10]. Dynamic polymorphism is achieved by having pointers to lead the function call to the right function implementation at run-time [16]. While this is more flexible than static polymorphism, it adds a run-time overhead involved with resolving virtual calls [32]. This downside may drive some projects, including Eclipse OMR, to use static polymorphism instead.

In order to explain the difference between static and dynamic polymorphism, we provide an example of using virtual functions, and compare it to the alternate implementation using CRTP. Consider Figure 2.1(a), class `B` inherits from class `A` and overloads its function `a()`. In a dynamic implementation, `a()` in `A` is declared as *virtual*, and `B` extends `A` as shown in Figure 2.1(b). Whereas in CRTP, `B` extends `A` such that `A` has a template class parameter, which is `B` itself, as shown in Figure 2.1(c). In the implementation of `a()`, `A` casts itself into the template class (which is its child) and calls the appropriate function as shown in Figure 2.1(d) (adapted from Bendersky's article [16]). Both implementations would end up with the program behaving the same however each has its advantages and drawbacks. Using `virtual` (or dynamic polymorphism) implies that the program would take more time to run every time. In addition, the syntax is more readable than having to cast every time a member function is called. On the other hand, using CRTP implies that the program would take more time to compile but run faster.

2.4 LLVM and Clang's Relation to OMRStatistics

Since creating a static analysis tool from scratch is time-consuming, we decided to take the same path taken by OMR developers and build our static analysis tool, OMRStatistics, as a plugin for the clang front-end compiler. OMRStatistics is explained in details in Chapter 5. We now discuss some

background about compilers and where are clang and OMRSTATISTICS triggered in the compilation process.

Generally, compilers have three components or phases [59]:

1. A *front-end* that parses the source code to create an *Abstract Syntax Tree* (AST)

2. An *optimizer* that makes optimizations on the AST; usually that AST is translated to an intermediate language which would be optimized by the optimizer

3. A *backend* that translates the AST or intermediate language to machine language

LLVM [93] is a generic compiler that uses a modular design and an *intermediate representation* (IR) to separate concerns between front-end and back-end developers [59], [60]. The IR is language independent which allows front-end developers to write front-end compilers for any language, which translates any language into the IR. On the other hand, optimization and back-end developers can write their algorithms and tools to deal with the IR. Clang [1] is a *C* language front-end for LLVM. Clang also allows writing plugins to use its resources and statically analyze the code. Our clang plugin, OMRSTATISTICS, uses clang's resources to collect information about the code.

2.5 Eclipse OMR

Java runtime technology has benefited from hundreds of person-years of development investment over the last two decades, resulting in a highly capable and scalable dynamic language that delivers powerful performance and has a vibrant developer ecosystem. The Eclipse OMR project aims to expand access to high-quality runtime technologies for other dynamic languages through an ongoing effort to restructure the core components of IBM's J9 Java Virtual Machine (JVM).

Eclipse OMR [33] is an open-source *C++* library that consists of multiple components for building language run-times, such as a compiler, garbage

collector, and a diagnostic engine, that are equipped for multiple architectures. These components are not created for a specific language. Instead, programming-language developers can leverage the designed software variability to add functionality to OMR to support their specific language.

2.5.1 Project History

OMR traces back to the IBM Java Virtual Machine, J9, as well as its Just in Time (JIT) compiler, Testarossa [90]. Testarossa is a multi-target compiler technology that translates Java bytecode to machine code, in order to accelerate program execution.

After successfully implementing Testarossa for Java, IBM adapted the compiler to other languages, including COBOL and other proprietary languages and runtime systems. This created the Testarossa SPL, which used dynamic polymorphism and build-time selection to achieve software variability. A changing industry suggested that it may be time for an entire language-runtime SPL. This hypothesis was the genesis of the OMR project, which refactored both the compiler component and the rest of the J9 JVM system to extract the core into a set of code called *OMR*. As a result, OMR was created to unlock the inner workings of the JVM without imposing Java semantics to create a common platform for building language runtimes [34]. Later, in 2016, it was open-sourced. Today, OMR supports *X86*, *Power*, *Z*, and *ARM* (under development) platforms.

The integral idea of OMR was that many SPLs can be built from the core components of OMR, which is intended to be itself largely language independent. While SPL concepts lend themselves well to compiler design, Eclipse OMR introduces SPLs for other run-time components as a bet that SPLs can be applied to language run-times.

Subsequently, the compiler component was refactored to use static polymorphism to express variability, moving language-specific code into subclasses. The garbage collector also used subclasses; however, it has so far used dynamic dispatch to reflect polymorphism. In one of our IBM collaborator's previous work [40], the refactoring process of the compiler component is described, as

well as the lessons learned from it, but without focusing on the SPL and variability perspective.

2.5.2 Project Structure and Size

OMR has 735,733 lines of code (LOC) according `SLOCCount` [2] on October 2018, and 111 contributors according to GitHub's statistics [33]. Most of the source files (85% [33]) are cpp files, and the source code is divided into individual component directories: Compiler in *compiler*, Garbage Collector in *gc*, etc. This allows the flexibility of having different variability implementation mechanisms in different components. For example, the garbage collector heavily uses `#ifdef` directives whereas the Compiler mostly uses static polymorphism. Our research focuses on the variability implementation in the Compiler component.

Chapter 3

Eclipse OMR Variability Implementation Mechanism

In this chapter, we explain how variability in Eclipse OMR is implemented based on our first-hand experience in diving into the source code. This chapter is based on our published paper [69]. We hereby present our first contribution and describe the path of how we achieved the first objective of this thesis (see Sections 1.4, 1.5).

Eclipse OMR supports three dimensions for product variability: (1) *language variability*, (2) *platform variability*, and (3) *feature variability* in some components. Given n features, the total number of unique OMR products is $|\text{languages}| * |\text{platforms}| * 2^n$. OMR currently supports five architectures: *X86/i386*, *X86/AMD64*, *Power*, *ARM*, and *Z*, and is used to develop run-time components for Java, Ruby, and Python. The number of supported programming languages is currently small, but is expected to increase since OMR's goal is to enable the quick development of language runtimes.

As previously stated, we focus on the compiler component in this thesis. In general, the variability implementation mechanism used in the compiler component is *static polymorphism*. However, it is different from the CRTP design we discussed earlier (in Section 2.3). In terms of implementation, OMR developers initially wanted to use the typical CRTP used to implement static polymorphism. However, due to the concern of introducing large amounts of template code that may bloat the system, a customized notion of static polymorphism was created based on the idea of extensible classes [67]. *Extensible*

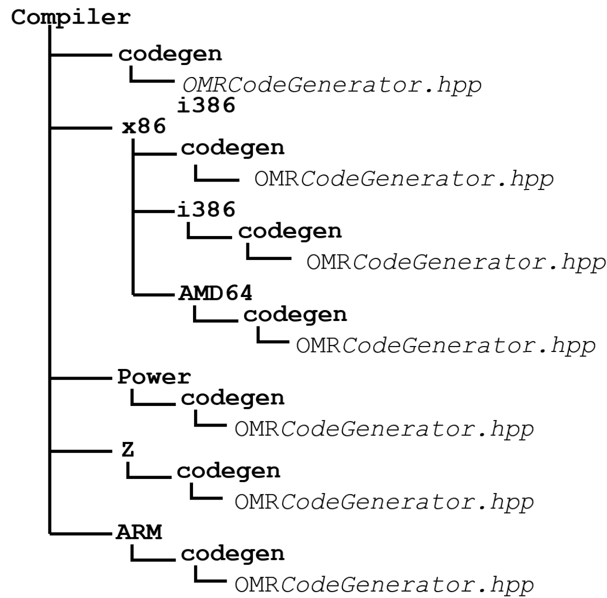

Figure 3.1: Directory structure of the compiler component

classes are simply a hierarchy of normal *C++* classes that are organized in a special way to allow the compiler to find the *most extended implementation* of a member function, i.e., the member function in the most derived class, no matter where this function is called from. Such classes are tagged with a keyword, `OMR_EXTENSIBLE`, when they are declared and are meant to be extended by users. Note that OMR consists of more than 1,000 classes that build up its various functionality across multiple architectures. However, only certain classes are extensible. This special organization in extensible classes depends on several building blocks that we now discuss.

3.1 Directory Structure

Common source code across all architectures is placed in the main directory of the compiler component, and the architecture-specific code for each platform is in a nested directory named after each platform. For example `CodeGenerator` is part of the compiler's implementation, responsible for generating intermediate-language code. Figure 3.1 shows the directory structure of the compiler component and how the header files of the `CodeGenerator` fit in. `CodeGenerator` is declared in *OMRCodeGenerator.hpp*; it is one of the

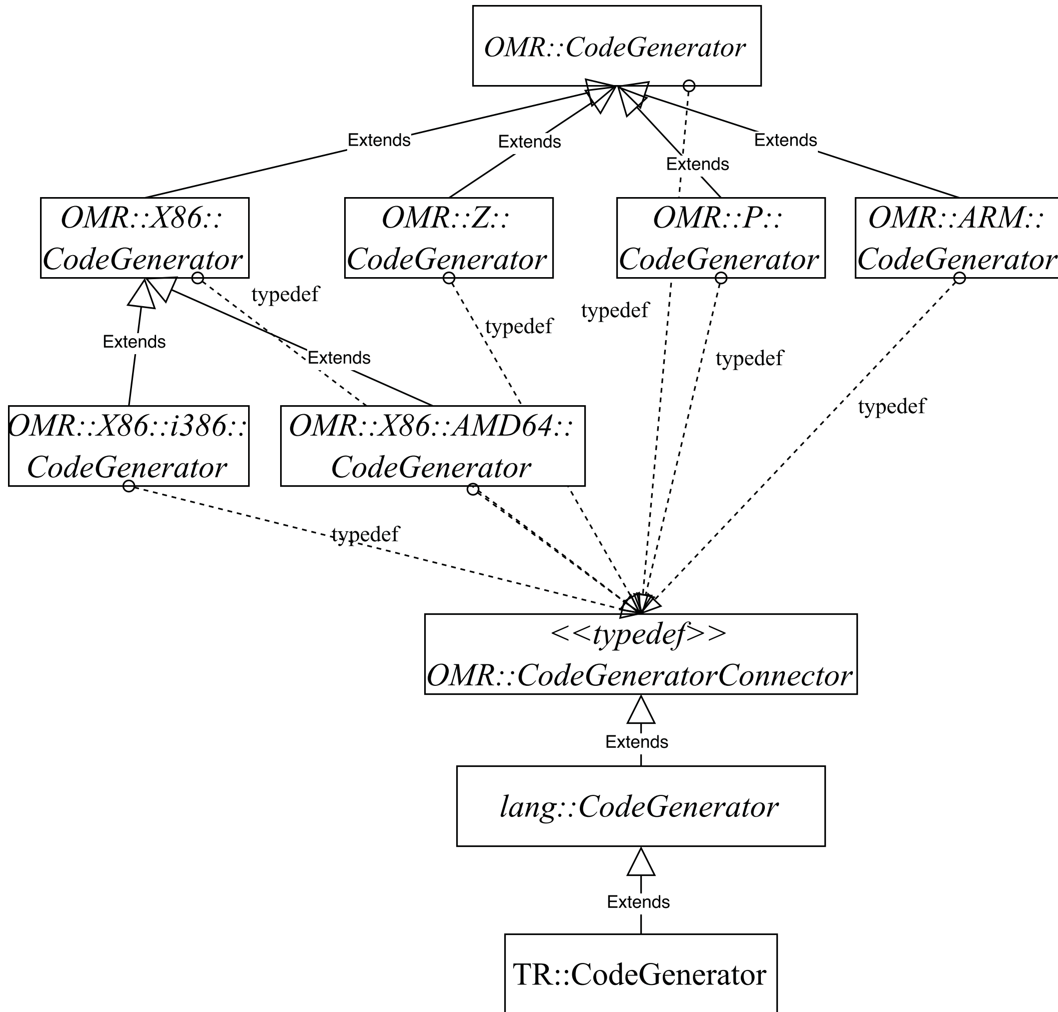

Figure 3.2: The `CodeGenerator` hierarchy. Classes in italic represent abstract classes.

classes that have different implementations according to the target programming language and architecture. Since `CodeGenerator` is part of the compiler component, its source files are under the `compiler` directory. The common header file and implementation code across all products are placed in a directory directly under `compiler`, whereas the specific header information and implementation code that customize the `CodeGenerator` for *Z* architecture, for example, are inside the `z` directory. Notice that different header files all share the same name; this is a critical fact that allows the include path variability (explained in Section 3.4) to function correctly.

3.2 Inheritance

To support architecture-specific functionality, OMR has an *extensible class hierarchy* that mimics the directory structure. To elaborate, we take the `CodeGenerator` extensible class hierarchy as an example. There is a `CodeGenerator` class in each architecture-specific directory as shown in Figure 3.1. Each `CodeGenerator` class extends its less specific counterpart in the parent directory as shown in Figure 3.2. For example, `CodeGenerator` inside *ARM* extends the main `CodeGenerator`. The `CodeGenerator` inside *i386* extends the *x86* one, which in turn extends the main `CodeGenerator`. Since *C++* does not allow multiple declarations with the same class name, OMR developers created a unique namespace for each architecture, which mostly corresponds to the directory structure. The base class in any extensible hierarchy, which would be in the file `compiler/codegen/OMRCodeGenerator.hpp` in Figure 3.1, is declared in the `OMR` namespace, while the rest of the classes have nested namespaces according to the corresponding architecture. Figure 3.2 shows the full extensible class hierarchy for `CodeGenerator`.

3.3 Connector Classes

3.3.1 Motivation for Creating Connector Classes

Note how all classes in Figure 3.2 apart from `TR::CodeConnector` are abstract classes. This is because developers who use OMR to implement a language runtime for a new programming language need to add *concrete classes* to the bottom of the class hierarchy. The intuitive way to extend the class hierarchy for all the supported architectures is to create a concrete class that extends the most specific class in that hierarchy for every supported architecture. For example, to extend `CodeGenerator` to work for a new language, *lang*, one would create a `lang::CodeGenerator` class that extends `X86::CodeGenerator` if targeting *x86*, `P::CodeGenerator` if targeting *power*, and so on. Although *C++* supports multiple inheritance, it is not possible to make use of it when the parent class is still uncertain due to architecture variability. For exam-

```
1  #ifndef OMR_CODEGENERATOR_CONNECTOR
2  #define OMR_CODEGENERATOR_CONNECTOR
3  #else
4  #error multiple definition of OMR::X86::i386::CodeGenerator
5  #endif
6  namespace OMR {typedef
7              OMR::X86::i386::CodeGenerator CodeGeneratorConnector;}
8  namespace OMR {
9    namespace X86 {
10     namespace i386 {
11       class OMR\_EXTENSIBLE CodeGenerator :
12                     public OMR::X86::CodeGenerator {
13         ...
14       }
15 }}}
```

Listing 3.1: Using typdef to connect `OMR::X86::i386::CodeGenerator` to `OMR::CodeGeneratorConnector`

ple, assume a function `f()` is implemented in the `CodeGenerator` classes in all architecture namespaces. If the language-specific `CodeGenerator` class (`lang::CodeGenerator`) tries to call `f()`, the compiler would complain that `f()` is ambiguous since it is implemented in more than one parent class.

3.3.2 Connectors Classes Description

With static polymorphism, all variability in the inheritance hierarchy must be resolved at compile time to gain efficiency over dynamic polymorphism. This means that at compile time, a single *linear* hierarchy for a particular architecture must be present. Hence, OMR developers had to provide a way for the language-runtime developer to extend from a single class, which is the most specific class of the target architecture. They created a new *connector* class for each existing extensible class hierarchy. For example, `CodeGenerator` hierarchy has a corresponding `CodeGeneratorConnector` class residing in the `OMR` namespace; another extensible class hierarchy, `Machine`, has a `Machine-Connector` class in the `OMR` namespace, and so on. The connector acts as a liaison between the class hierarchy it is representing in *OMR* namespace and external classes that aim to extend this hierarchy from outside the `OMR` namespace. For example, `lang::Machine` would extend `OMR::MachineConnector` to use its functionality.

3.3.3 Connector Classes Implementation

We use the `CodeGenerator` hierarchy to explain how connectors bridge the gap between the correct most-derived class of the extensible class hierarchy and other `OMR` classes in *C++*. Every time a `CodeGenerator` class along the extensible class hierarchy is declared, a `typedef` from that class to `CodeGeneratorConnector` is created as shown in Listing 3.1. Note how the `typedef` has an `#ifdef` guard similar to traditional `#include` guards. Since all classes along the same extensible class hierarchy will have the `typedef` statement, the guard ensures that only one `typedef` is defined at a time. The general goal is to ensure that when we compile for `X86/i386`, the only compiled `typedef` statement is the one that connects `OMR::X86::i386::CodeGenerator` to `OMR::CodeGeneratorConnector`, whereas if we compile for ARM architecture, the compiled `typedef` is the one connecting the `OMR::ARM::CodeGenerator` to the `OMR::CodeGenerator-Connector`. Looking at the big picture, `OMRCodeGeneratorConnector` is connected by a `typedef` to each class along the extensible class hierarchy in each architecture as shown in Figure 3.2. Hence, as long as there is a way to guide the compiler to detect the `typedef` in the most derived class first, `Code-GeneratorConnector` will represent the correct most-specialized class in that architecture. The order in which files get compiled can be controlled via the include paths passed to the compiler; more on this in the next section.

Now that the right class to extend from is identified, language developers can create their own customizations by extending the connector class as shown in Figure 3.2. To provide a generic way to always use architecture and language extensions that are only determined at compile time, OMR developers created a namespace `TR`, short for *Testarossa*, that contains the final implementation of the current combination of language and architecture extensions, and which will be used by the runtime-environment components. For example, assume we use OMR for language `lang` on an *ARM* host, and we need to use `CodeGenerator`, we would then directly use `TR::CodeGenerator` since it is guaranteed to have all the necessary extensions: `OMR::CodeGenerator`,

OMR::ARM::CodeGenerator, and lang::CodeGenerator (the language's adaptation of the CodeGenerator).

3.4 Include Paths

One last problem when using connectors is how to connect the right class to the connector class. Going back to the CodeGenerator class hierarchy, when we compile for the *Power* architecture, we actually compile two CodeGenerator classes, OMR::CodeGenerator and OMR::P::CodeGenerator, and each of them has a typedef for OMR::CodeGeneratorConnector. Hence, the challenge is how to connect the connector to the most specific CodeGenerator. This is solved by exploiting the compiler's prioritization of include paths.

When compiling a class that implements the compiler component on the *i386* architecture, the following includes are passed to the preprocessor: -Icompiler-/x/i386
- -Icompiler/x -Icompiler. Since the preprocessor searches for files in the order of the passed includes, it will search for the class in *i386*-specific classes first, then in *x86*-specific classes, and lastly in the base classes common for all architectures. For example, CodeGenerator is found in the *i386* directory since a specialized implementation is present. On the other hand, a class that has a single common implementation for all architectures will be found in the compiler main directory. Based on the first file found and processed, the CodeGeneratorConnector will be associated with a different class.

3.5 #ifdef Directives

OMR also uses #ifdef directives to implement variability, especially in the Garbage Collector component. #ifdef directives can be used to include or omit blocks of code by passing -D arguments to the preprocessor. Some of the present macros control architecture-specific functionality, such as TR_TARGET-_X86, while others are used for debugging, such as -DDEBUG_ARM_LINKAGE. Finally, there are macros used to select specific features or functionality in the code. For example, OMR_GC_MODRON_SCAVENGER is an optional feature

of the garbage collection component. In general, there are many ways the `#ifdef` directives are exploited in OMR, from being able to enable certain optimizations to being able to specify the endianness of the build. However, since the focus of our research was on the Compiler component, we do not discuss `#ifdef` directives any further in this thesis.

Chapter 4

Static Vs Dynamic Polymorphism

OMR's variability implementation described in Chapter 3 guarantees to provide an optimal runtime performance due to the elimination of indirections that would be introduced in function calls by dynamic polymorphism. However, static polymorphism comes at multiple costs. We now discuss these costs which are the main motivation behind revisiting the variability implementation mechanism in OMR and considering dynamic polymorphism as a replacement for static polymorphism. We first explain the *self()* function, then discuss its pros and cons. Lastly, we describe the developers' opinions about the current implementation. This chapter describes part of our work towards the second objective of this thesis (see Section 1.4).

4.1 The `self()` Function

One of the main characteristics of OMR's variability implementation is that the most derived class in a given extensible class hierarchy is always the one that is used for all functionality. Consider the scenario in Figure 4.1, where class `A` is the base of the class hierarchy (similar to `OMR::CodeGenerator`) and class `C` is the class with the most-specific implementation in the hierarchy (similar to `OMR::X86::AMD64::CodeGenerator` when building OMR for *X86/AMD64* architecture). Note that function `a()` in class `A` calls function `b()`. Based on the desired inheritance behavior in OMR, it is expected that whenever function `b()` is called, even when the call is from inside class `A`, the most specific implementation of `b()`, which is in class `B` in this case, is executed.

```
 1  class A {
 2  public:
 3      A() {}
 4      void a() {
 5          printf("function a from class A\n");
 6          b();                                        ");
 7      }
 8      void b() {
 9          printf("function b from class A\n");
10      }
11  };
12
13  class B : public A {
14  public:
15      B() {}
16      void b() {
17          printf("function b from class B\n");
18      }
19  };
20
21  class C : public B {
22  public:
23      C() {}
24  };
25
26  int main() {
27      C instance;
28      instance.a();
29  }
```

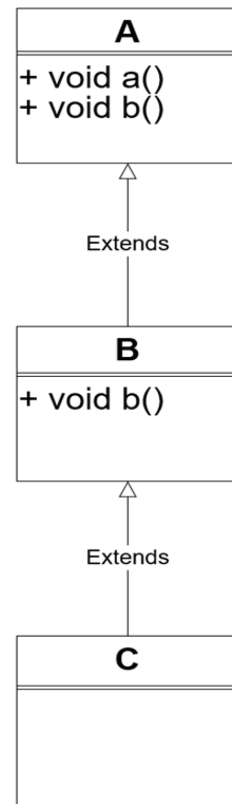

Figure 4.1: Example explaining the need for the `self()` function

Hence, the following is the desired output from the code in Figure 4.1:

```
function a from class A
function b from class B
```

However, running the example in Figure 4.1 prints:

```
function a from class A
function b from class A
```

This means that `A::b()` is executed instead of `B::b()`. This is due to the value of the self-pointer (accessed by `this` keyword) after resolving the call for `a()` statically. When `a()` is called from an instance of `C` on line 28, the compiler resolves the call to the implementation of `A::a()` on line 4 in Figure 4.1 since class `C` does not have the function and class `A` is the nearest parent class of `C` that has function `a()` defined. Calling `b()` from inside `A::a()` in line 9 is implicitly calling `this->b()`. Since the self-pointer here is a class `A` pointer, the call resolves to `A::a()`.

However, OMR developers want to force the program to start searching for `b()` from the bottom of the class hierarchy again, which is a functionality that is not supported by static polymorphism. To solve this, OMR developers created a function `self()` that always returns a fresh pointer of the most concrete class in an extensible hierarchy by downcasting the self pointer (`this`). Hence, instead of calling `b()` from function `a()`, developers have to call `self()->b()`. `self()` will return a pointer to class `C`, which forces the program to look from the bottom of the hierarchy again and get the most specific implementation available. Given that the `TR` namespace contains most of the concrete classes of the different extensible hierarchies, `self()` is implemented to always return a pointer of the concrete class in `TR`.

It is important to note here that this problem occurs because the function call to `b()` is being resolved statically at compile time (static polymorphism). If the call on line 28 was resolved at runtime, as in dynamic polymorphism, the implementation of `A::a()` will be called at runtime. Hence, `this` pointer in line 6 would be a pointer to class `C` and the call on that line would be a call on a pointer to class `C`. So line 6 would trigger the program to use the implementation of function `b()` pointed to in the virtual method table of the

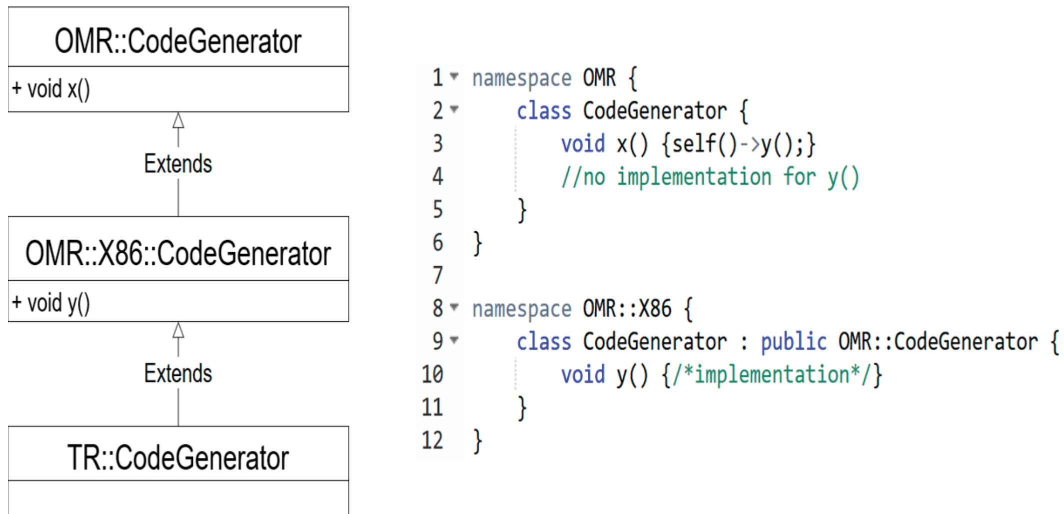

Figure 4.2: Example of an irregular case that becomes legal when using `self`

`instance` object (which is of type `C`) and hence the implementation `B::b()` would be executed.

4.2 Implications of `self()`

The `self()` function can be considered a hack for static polymorphism design to force the compiler to get the most specific function implementation when resolving the calls. This hack has multiple implications. We now discuss one advantage and three consequences of the `self()` function.

4.2.1 Advantage: Less Runtime Cost

Developers use `self()` to trigger static polymorphism in *C++*, which allows the compiler to resolve the function calls at compile time. Hence, when using `self()`, there is no added runtime cost of calling polymorphic functions when compared to running non-polymorphic functions. On the other hand, using dynamic polymorphism delegates part of the task of resolving function calls to the program at runtime, which is an added cost.

4.2.2 Consequence: Irregularities

The introduction of `self()` function allows usage of *C++* in a way that is not allowed in the language by default. Figure 4.2 demonstrates that way, where the `self()` function allows developers to call a function that is defined in a derived classes (`y()`) without defining it in the base class. Allowing such a case adds to the complexity of reading and understanding the OMR code base.

4.2.3 Consequence: *OMRChecker*

Given that the use of `self()` is a convention created by OMR developers, silent failures due to missing downcasts when using `this` instead of `self()` can occur. In order to ensure the conventions are respected, OMR developers created `OMRChecker`, a static linter implemented as a `Clang` [1] plugin that checks for the use of `self()` instead of `this`. To signify for the linter that a specific class hierarchy is using static polymorphism and should be using *self()*, OMR developers tag all the class declarations in such a hierarchy with `OMR_EXTENSIBLE` tag. For example, all class declaration of the `CodeGenerator` hierarchy would look like this: `class OMR_EXTENSIBLE CodeGenerator{....`

In more details, the linter checks: (1) that concrete classes in class hierarchies are in the correct namespace (usually the `TR` namespace), (2) that `self()` replaces `this` in the appropriate places [67], and (3) if one class declaration contains the *OMR_EXTENSIBLE* tag, all classes that belong to the same hierarchy are declared with that tag.

4.2.4 Consequence: No Clear API

In order to add a feature to an SPL, developers have to find the extension points of the project use them to extend the project's support for the new feature. In OMR, client developers have to extend the right classes and override the right polymorphic functions to connect their language to the library. Since there are multiple reasons for extending and overriding classes and functions respectively in OMR, not all extended and overridden classes and functions respectively are meant to be extension points for the project. In other words,

some classes are extended but client developers are not expected to use them to connect their language to the library. Similarly, some functions are overridden but are not expected to be further overridden by client developers. This causes confusion for new client developers as to which functions and classes should be considered as extension points and which ones to keep intact. In other words, new client developers cannot identify Eclipse OMR's API clearly. Part of this research's future work is to initiate a new code convention to mark API functions and classes. Although created for a different reason, the *OMR_EXTENSIBLE* tag already marks API classes. However, marking API functions also is necessary.

4.3 Recap of the Consequences of OMR's Current Variability Implementation Mechanism

To recap, static polymorphism, is adding conventions that the community contributors are obliged to follow, making the code less approachable and harder to read. In spite of that, OMR developers decided to go with static polymorphism since it is expected to have better runtime performance which is worth the consequences. Hence, we decided to study the trade-offs between the two types of polymorphism implementations and raise the question whether it is worth switching to dynamic polymorphism or not. For example, if the extension points are switched to dynamic polymorphism, that is having *virtual* in the definition of functions that are expected to be overridden later, language developers who are extending the language do not have to deal with the *self()* convention.

4.4 Moving to Dynamic Polymorphism

Given the above downsides, some of the OMR developers were advocating the change to dynamic polymorphism. The hypothesis was that the vast majority of specialization exists in a small number of methods that are not sufficient to substantially impact the run-time speed and that furthermore, switching to dynamic polymorphism may, in fact, *improve* the compilation performance

by allowing the compiler to do a better job building the source code. This improvement would be due to the ability to declare functions in header files, allowing the build compiler to inline more functions. In order to test this hypothesis and collect more information to help OMR developers reach a decision about the implementation that better fits their project, we created a tool, OMRStatistics, that analyzes the methods and classes of OMR's source code.

Chapter 5

OMRStatistics

In this chapter, we introduce OMRStatistics, our open-source static analysis tool, which we built as a Clang plugin [78]. It records the parent-child class relationships in the source code and collects information about the methods in these classes. This information includes their source location, where they have been overridden, and whether they are virtual or implicit. OMRStatistics records all this information in a MySQL database to make it easier for developers to query. Additionally, it provides visualizations of the information in the form of diagrams and HTML pages.

In this chapter, we continue describing our work towards the second objective of this thesis (see Section 1.4). We also present our second contribution: OMRStatistics (see Section 1.5). We describe in details the process of developing OMRStatistics, including algorithms of how it works and what test case we created to ensure its correctness. After that, we show how we utilized OMRStatistics to get useful information for assessing which variability implementation decision better fits the project.

5.1 Features

As an overview, OMRStatistics has the following features:

- Prints class hierarchy information in a CSV file
- Prints class and function information in a CSV file

- Creates a MySQL database from the CSV files that are easily queried for useful information
- Visualizes the class hierarchies in a PDF file
- Visualizes the function override information in an HTML file
- Has a Makefile system that connects all the components together so that one `make` command can run the tool over OMR

5.2 Setup

Since we implement OMRStatistics as a Clang plugin, it runs its analysis while compiling a given source file with Clang. In order to run OMRStatistics on the whole OMR source code, the whole source code has to be compiled by Clang. For a given compilation, OMR would need to already be configured with the combination of architecture and programming language to build for. This means that only a subset of the source files would be analyzed in any given build. Analyzing all variants of the source code means compiling OMR multiple times with all possible combinations of architectures and languages. At the moment, OMR has a relatively small number of variants due to the handful number of languages that use OMR. However, we expect that the number of variants to rapidly increase as more languages start using the OMR technology. For now, we individually run OMRStatistics on all variants and then aggregate the results; it is part of our future work to edit Clang such that we can leverage the similarities between variants and avoid redundant runs on the same file [69].

In order to setup OMRStatistics in an OMR project, `make produceVisualizations` should be run. After that, the instructions inside *tools/compiler/OMRStatistics/database/all.sql* need to be executed on a *MySQL* server to build the database.

We ran OMRStatistics on OMR and OpenJ9 [35], IBM's Java Virtual Machine (JVM) implementation. OpenJ9 is the biggest consumer of the OMR

```
1  // Data Structure declarations
2  Declare Class MethodTracker // keeps track of method information.
3  Declare Data Structure Node having
4    name // Name of class represented
5    parent // Pointer to node representing the parent of the represented class
6  Declare a Data structure Hierarchy having
7    base // Node representing the base class of the hierarchy
8    methodName2MethodTracker // Maps a function name to a set of MethodTracker objects
9
10 // Phase 1: Collecting information
11 classHierarchy = {class —> []} // maps relationship between each class and its array of parents
12 class2Methods = {class name —> method signature} // maps each class to its member methods
13
14 def recordParent(ClassDecl decl):
15   foreach (parent in decl.parents) do
16     if (decl in classHierarchy.keys() and parent in classHierarchy[decl]) then continue
17     classHierarchy[decl].add(decl.parent)
18     recordParent(parent)
19
20 foreach (ClassDecl decl in the source code) do
21   recordParent(decl)
22   foreach (MethodSignature sig in decl) do
23     class2Methods.add(decl, sig)
```

Listing 5.1: Pseudocode for definitions and phase 1 of the algorithm used in OMRSTATISTICS to collect class hierarchy and method information

technology today. Analyzing it gives us an insight into how language developers already use OMR and how future language developers are likely to use the OMR library.

5.3 OMRSTATISTICS Implementation Mechanism

5.3.1 Mechanism Overview - The Algorithm

As a Clang plugin, the tool is limited by the API provided by Clang, which is to traverse the Abstract Syntax Tree (AST) after Clang has already parsed the source code and created the AST. We hereby present our algorithm in pseudocode in Listings 5.1, 5.2, and 5.3 that uses Clang's `RecursiveASTVisitor` class which calls its member function `VisitCXXRecordDecl(const CXXRecordDecl *decl)` to traverse *C++* class declarations. After that we use the `ASTConsumer` class's `HandleTranslationUnit(ASTContext & Context)` function to process the collected data. `HandleTranslationUnit(ASTContext & Context)` runs after all calls to `VisitCXXRecordDecl(const CXXRecordDecl *decl)` are done, hence when executing the code of that function, Clang guarantees that all classes are visited and all our data is collected.

Our algorithm is divided into three phases. First, we collect the information from the source code and store them in data structures. Then, we create the class hierarchies. After that, we process the function information to get links between each class and its functions in addition to override information. We now explain, in more details, each phase of the algorithm, while referring to Listings 5.1, 5.2, and 5.3.

5.3.2 Phase 1: Collecting Information

In this phase, described in Listing 5.1, we first define a function that recursively visits every parent of a class declaration C (i.e., a class from which C inherits), and records the relationships between every class and its parent, then we collect the method information about the original class declaration visited. More specifically, OMRStatistics does the following:

1. Visit every class declaration in the source code (line 20)
2. Recursively iterate through all the parents of that declaration (lines 15 and 18).
3. When a parent that has not been processed earlier is found, record the relationship between this class and its parent in a binary map (called *classHierarchy*) (line 16-17).
4. After processing all parents of the initial class declaration (where `VisitCXXRecordDecl` was called), go back to the initial declaration and record all its methods in a map (called *class2Methods*) (lines 22-23).

5.3.3 Phase 2: Creation of Class Hierarchies

For this part of the algorithm, we need to define two data structures: `Node` (line 3 in Listing 5.1) and `Hierarchy` (line 6 in Listing 5.1). A `Node` basically represents a *C++* class. Each node has a string name and a pointer to its parent. On the other hand, `Hierarchy` represents a class hierarchy; i.e., represents a collection of `Node`s. Each hierarchy has a base representing its base class and a `methodName2MethodTracker` which is used to collect information about

```
1  // Phase 2: Creation of the Class Hierarchies
2  class2Address = {class name —> address} // Links each class to the address
3                                          // in memory of the node representing it
4  hierarchies = [] // Array of Hierarchy structures
5  for (child, parent) in classHierarchy do
6    if (child in class2Address && parent not in class2Address) do
7      Define Node newNode
8      newNode.name = parent
9      class2Address.add(newNode, &newNode)
10     child = node found in class2Address // representing the child class
11     child.parent = newNode
12   else if (parent in class2Address && child not in class2Address) do
13     Define Node newNode
14     newNode.name = child
15     class2Address.add(newNode, address of newNode)
16     parent = node found in class2Address // representing the parent class
17     newNode.parent = parent
18     if (Hierarchy h | h.base == parent) do
19       h—>base = newNode
20   else if (child not in class2Address && parent not in class2Address) do
21     Define Nodes newNode1 and newNode2
22     newNode1.name = child
23     newNode2.name = parent
24     class2Address.add(newNode1, address of newNode1)
25     class2Address.add(newNode2, address of newNode2)
26     newNode1.parent = newNode2
27     Define Hierarchy h
28     h.base = newNode1
29     hierarchies.add(h)
30   else
31     // if child and parent are found in class2Address
32     child = node found when searching class2Address for child class
33     parent = node found when searching class2Address for parent class
34     child.parent = parent
35     if (Hierarchy h | h.base == parent) do
36       delete h
```

Listing 5.2: Pseudocode for phase 2 of the algorithm used in OMRStatistics to collect class hierarchy and method information

functions in that hierarchy; we explain and use this map in the third phase of the algorithm. An array to store all hierarchy structures, `hierarchies`, is also used. A map, `class2Address`, that maps every class name to its created `Node` address is also used. In this phase, described in Listing 5.2, OMRStatistics does the following:

1. Iterate through every record in *classHierarchy*. The key represents the child class name and the value represents the parent class name (line 5).

2. Search for the class name in the `class2Address` map, finding the class means that there already exists a node with this class's name.

3. If only the child class was found in the map (line 6):

   - Create new node for the parent class (lines 7-8)
   - Store the new node in `class2Address` (line 9)
   - Link the child's parent pointer to the new node (lines 10-11)

4. If only the parent class was found in the map (line 12):

   - Create new node for the child class (lines 13-14).
   - Store the new node in `class2Address` (line 15).
   - Link the new node's parent pointer to the parent (lines 16-17).
   - Search all hierarchies (in the `hierarchies` array) for the hierarchy whose base is the parent. Change the base of that hierarchy to the new node (lines 18-19).

5. If neither child nor parent class were found in the map (line 20):

   - Create new node for both the child and parent classes (lines 21-22).
   - Store the both new nodes in `class2Address` (lines 24-25).
   - Link the new child node's parent pointer to the new parent node (line 26).
   - Create a new Hierarchy that has these two nodes (lines 27-28).

```
1
2  // Phase 3: Collect Method Information
3  subHierarchies = []
4
5  def getLinearHierarchy(Node n, Node[] array):
6    array.add(n)
7    if(n.parents.length == 0) do // Top of hierarchy reached
8      subHierarhcies.add(array)
9    else do
10     foreach(parent in n.parents) do
11       getLinearHierarchy(parent, array)
12
13 for (Hierarchy h in hierarchies) do
14   getLinearHierarchy(h.base, [])
15   foreach(Node[] subHierarchy : subHierarchies)
16     Define Map methodName2MethodTracker // Links every function name to the set of
17                         // trackers that represent functions with that name
18     foreach (Node n : subHierarchy) // Traverse from last node to first node
19       for (function f in class2Methods[n.name]) do
20         if (f.name in methodName2MethodTracker) do
21           m = MethodTracker found in methodName2MethodTracker
22           m.addOccurrence(decl)
23         else do
24           m = Create MethodTracker for f
25           methodName2MethodTracker.add(m)
```

Listing 5.3: Pseudocode for phase 3 of the algorithm used in OMRStatistics to collect class hierarchy and method information

- Add the new Hierarchy to the `hierarchies` array (line 29).

6. If both child and parent class were found in the map (line 30):

    - Link the child node's parent pointer to the parent node (lines 31-33).
    - If a hierarchy exists such that the hierarchy's base is the parent node, delete that hierarchy (lines 34-35).

5.3.4 Phase 3: Method Information Collection

In order to collect method information about each class in a class hierarchy, we create a data structure, called `MethodTracker` for every function signature that contains information about the function. Then, we create a map (`methodName2MethodTracker`) for every hierarchy that connects every method name in a class in that hierarchy to an array of all the trackers that represent functions with that name. Multiple trackers might be found for the same name in cases of overloads. Since in these cases, the function name is the same but different signatures are defined which means different trackers are

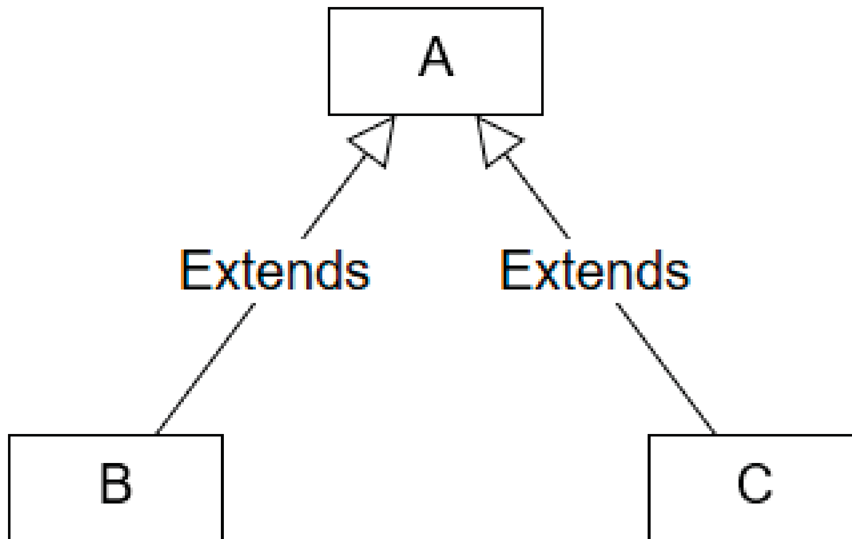

Figure 5.1: Example UML explaining `getLinearHierarchy` function

instantiated. The only reason for this map is to make searching for trackers faster. When searching for a function signature, we extract its name first and search for it in the `methodName2MethodTracker` map. From that search, we get all the trackers that represent functions with that name. Then, we search each of these trackers for the right signature of the function being searched for.

We also define a function `getLinearHierarchy` that transforms the Hierarchy structures we have into Linear ones whose nodes are stored in arrays, each of them called a `subHierarchy`. For example, calling `getLinearHierarchy` on a hierarchy similar to the one in figure 5.1 produces two subhierarchies: the first would be `[B,A]` symbolizing `B->A` and the second would be `[C,A]` symbolizing `C->A`.

In phase 3, described in Listing 5.3, OMRStatistics does the following:

1. For each hierarchy, load the list of subHierarchies in the `subHierarchies` array. This happens as follows:

   (a) Call the `getLinearHierarchy` function passing to it the base of a hierarchy and an empty array of `Node` structures (line 14)

   (b) Add the current node to the current array passed in the parameters (line 6).

(c) If the current node has parents, then this is not the top of the hierarchy. In this case, recursively call the function on each of the node's parents, giving it the same array to allow incrementally adding all the nodes of a linear hierarchy in the same array (lines 9-11).

(d) If the current node has no more parents, then we reached the top of the hierarchy. At this point, the passed array would have all the nodes representing a full subhierarchy. Hence, add this array to the list of completed subhierarchies (lines 7-8).

2. After that, we traverse the subhierarchies inversely, from the last node to the first node. This is because, initially, nodes are stored from the most derived class to the most base class. In other words, the more derived class nodes come before their parents. However, to derive method overrides we need to search the hierarchies from top to bottom (line 18).

3. Use `class2Methods` map to get all functions for the class represented by the current node. For each of function check if the name of that function is already in `methodName2MethodTracker` (lines 19-20).

4. If it is, then this is an override, record the occurrence of the method in that class inside the `MethodTracker` (lines 21-22).

5. If it does not exist in the map, then this is a unique function. Create a new tracker and add it to the map (lines 23-25).

5.4 OMRStatistics System Implementation

In this section, we describe how OMRStatistics works in the project to get the results it needs. As mentioned in the features section (Section 5.1), we have a Makefile system in place that runs the tool automatically. Hence, we now describe this Makefile's execution when running `make produceVisualization` while elaborating on every step of the process.

5.4.1 Step 1: Find the Source Files

Since OMR supports multiple architectures and relies on build path variability, OMR already has a system in place to get the list of files that must be built given a specific architecture. That system is a Makefile that takes four environment variables as parameters and returns the source files in two different environmental variables. More specifically, the Makefile takes the architecture information in the following variable name: `HOST_ARCH`, `HOST_SUBARCH`, `TARGET_ARCH`, and `TARGET_SUBARCH` and loads all the source file names and paths that should be compiled in variables named: `JIT_PRODUCT_BACKEND_SOURCES` and `JIT_PRODUCT_SOURCE_FILES`. This Makefile is located in *omr/fvtest/compilertest/build/files/* and it is called *common.mk*.

In order to run our tool, we: (1) set the four input environmental variables in our own makefile, (2) include the Makefile mentioned above, (3) concatenate the two output variables into a variable called `LIST`, and (4) lastly call clang to compile every file in `LIST`.

Since our tool runs on all architectures, we basically reproduce the steps described above four times (for each architecture: *X86/I386*, *X86/AMD64*, *Power*, and *Z*) where in each instance we define the input variables differently to represent the right architecture.

5.4.2 Step 2: Run OMRStatistics on the Source Files and Produce Visualizations

When OMRStatistics is run on the OMR source code, it creates five CSV output files as follows:

1. *allClasses.csv*: contains a record for each class. The record includes the class name, the namespace it resides in, and whether this class is extensible or not.

2. *allFunctions.csv*: contains a record for every function. The record includes the function name, signature, class which this function belongs to, and whether this function is implicitly declared and/or virtual.

3. *functionLocation.csv*: contains the information that links every function to the source file location where it was declared.

4. *hierarchy.csv*: contains two fields in each record. The first one indicates whether this hierarchy is extensible or not. The second field is a textual serialization of that hierarchy. The hierarchy is represented in the following form: `class -> parent 1 -> ... -> parent n`.

5. *overrides.csv*: contains function override information. Specifically, it holds the qualified class name of the base class, the qualified class name of the derived class where the function was overridden, and the function signature that was overridden.

Since we run the tool four times on different architectures, we get the above outputs four times. Hence, as part of our build system, we have a combination of batch and python commands that does the following: (1) copy all the original files to a directory as a backup, (2) concatenate all outputs together, and (3) remove duplicate records. After doing that, we end up having all the information we need in all four architectures in the project stored in five CSV files in total.

Recall that classes in one hierarchy should have the same name but they would be located in different namespaces (see Chapter 3). Since the OMR library is always under development, we found some hierarchies that defy this rule and are expected to be fixed at a later stage in the OMR development phase. Such hierarchies would have classes of different names but are located in the same namespace. We called such hierarchies *Internamespace Hierarchies* and put them in a separate outfile called: *interNamespaceHierarchies*

After that, we use Python and JavaScript tools (*omr/tools/compiler/-OMRStatistics/visualization/Hierarchy/getDatabaseSQL.py* and *omr/tools/compiler/OMRStatistics/visualization/Overrides/processOverrides.js*) to create a PDF file and populate an HTML file (*overrides.html*) respectively from the data in *hierarchy.csv* and *overrides.csv*. The PDF file (*hierarchy.pdf*) is a PDF file that visualizes the class hierarchies. A screen shot of the visualization is in

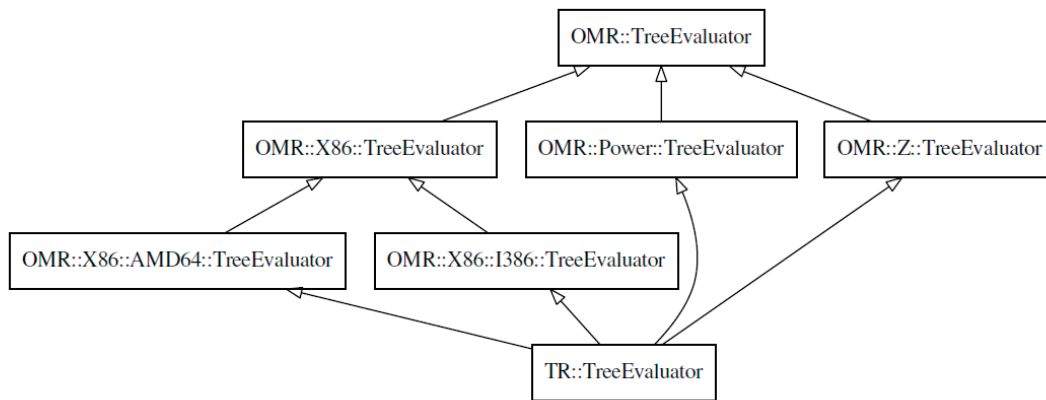

Figure 5.2: Hierarchy visualization file

Hide Implicit Declarations

OMR::StaticSymbol::StaticSymbol(TR::DataType,void *)
OMR::StaticSymbol::StaticSymbol(TR::DataType)
OMR::Symbol::getName()
OMR::StaticSymbol::~StaticSymbol()
OMR::X86::AMD64::Machine::Machine(TR::CodeGenerator *)
OMR::Machine::~Machine()
OMR::X86::Machine
OMR::X86::AMD64::Machine
TR::Machine
OMR::X86::I386::Machine
TR::Machine
OMR::Power::Machine
TR::Machine
OMR::Z::Machine
TR::Machine
OMR::X86::RealRegister::RealRegister(TR::CodeGenerator *)
OMR::Register::Register(enum TR_RegisterKinds)
OMR::Register::Register(enum TR_RegisterKinds,uint16_t)
OMR::Register::Register(uint32_t)
OMR::Register::block()

Figure 5.3: Part of the overrides visualizations web page. Originally, all nodes had the same font-size and underlined. Green nodes represent virtual functions and red nodes represent implicit functions. However, in order to make this figure compatible with black-and-white printing, we made only virtual functions underlined, and implicit functions are italicized and have larger fonts. Pressing the button on top hides the implicit functions.

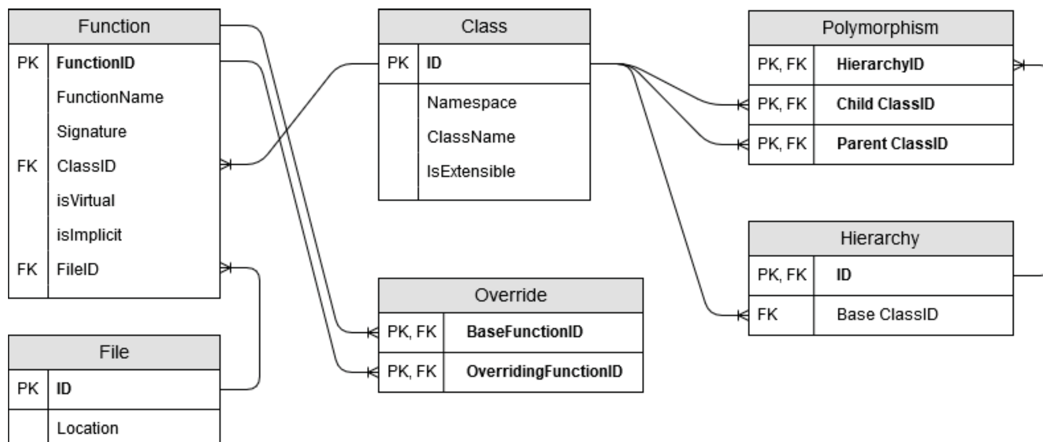

Figure 5.4: OMRStatistics relational database schema

Figure 5.2. The *overrides.html* is a web page that visualizations the overrides present in OMR. Figure 5.3 shows an excerpt of this visualization.

5.4.3 Step 3: Creating Database

After the visualization, the system triggers another Python tool (*omr/tools/-compiler/OMRStatistics/database/getDatabaseSQL.py*) to use the CSV outputs to create MySQL instructions to create the database. The database can be queried to find different information about how the functions are used. Figure 5.4 shows the relational database schema.

The *Function* and *Class* tables contain all the functions and classes found in the project. The *Function* table defines each function by an ID. The record contains the name and signature for each function, whether this function is virtual or implicit, the header file where this function is declared, and to which class this method belongs. The *Class* table contains a record for each class, mentioning its name, namespace, and whether it is extensible. Similarly, the *File* table contains information about the source locations of declarations. For now, we keep track of the file location only; however, it is kept in a separate table in case more information is needed about the source locations. The class relationships are also saved in the *Polymorphism* and *HierarchyBase* tables. Finally, the override relationships are found in the *Override* table. Note that in the schema, there is a table called *FunctionCalls*; this contains information

about function calls in the source code. However, the information in this table is currently not complete; we later realized that there are corner cases that we did not correctly handle and hence did not use this table in our analysis. We only add it here for the completion of the actual database schema in our project.

5.4.4 Recap and Implications

In summary, OMRStatistics and the system around it run in three phases: (1) get source files from the OMR build system, (2) run OMRStatistics on each of these source files and get the outputs, and (3) concatenate and trim those outputs then produce visualizations and a MySQL database.

OMRStatistics can be helpful in multiple ways: first, the information provided helps to check whether the amount of overridden functions in extensible classes is enough to significantly impact performance if virtualized. In addition, OMRStatistics helps OMR developers document the API boundaries and reason about the extension points of OMR on a per-method basis. In summary, OMRStatistics helps developers reason about the variability in their source code.

Although we currently only ran OMRStatistics on OMR and OpenJ9, it is important to note that the nature of the tool as a Clang plugin allows it to run on any source code that can be compiled by Clang, hence it does theoretically work on other *C++* projects.

5.5 Use Cases

In order to show how our tool can be used, we demonstrate here a few use cases that show only a subset of what our tool can do. First, OMRStatistics should be set up as described in Section 5.2. We now describe some use cases, alongside with the instructions and results specific for each use case.

```sql
SELECT DISTINCT bc.Namespace, bc.Classname, of.Signature, oc.Namespace, oc.Classname
FROM Override as o
INNER JOIN Function as bf on bf.id = o.BaseFunctionID
INNER JOIN Function as of on of.id = o.overridingFunctionID
INNER JOIN Class as bc on bc.id = bf.classID
INNER JOIN Class as oc on oc.id = of.classID
WHERE bc.className= 'CodeGenerator';
```

Listing 5.4: The query used to find all overridden functions

OMR	CodeGenerator	CodeGenerator()	OMR::X86	CodeGenerator
OMR	CodeGenerator	CodeGenerator()	OMR::Power	CodeGenerator
OMR	CodeGenerator	CodeGenerator()	OMR::Z	CodeGenerator
OMR	CodeGenerator	IsInMemoryType(TR::DataType)	OMR::Z	CodeGenerator
OMR	CodeGenerator	addProjectSpecializedPairRelocation(uint8 t *.u...	J9	CodeGenerator
OMR	CodeGenerator	addProjectSpecializedRelocation(uint8 t *.uint8...	J9	CodeGenerator
OMR	CodeGenerator	addProjectSpecializedRelocation(TR::Instructio...	J9	CodeGenerator
OMR	CodeGenerator	allocateRecompilationInfo()	J9::X86	CodeGenerator
OMR	CodeGenerator	allocateRecompilationInfo()	J9::Power	CodeGenerator
OMR	CodeGenerator	allocateRecompilationInfo()	J9::Z	CodeGenerator
OMR	CodeGenerator	allowGlobalRegisterAcrossBranch(class TR Regi...	OMR::X86	CodeGenerator
OMR	CodeGenerator	allowGlobalRegisterAcrossBranch(class TR Regi...	OMR::Power	CodeGenerator

Figure 5.5: The results from the database when running the query in 5.4 to find all overridden functions

5.5.1 Use Case 1: Find All Overridden Functions

In order to find all overridden functions in a class `CodeGenerator` in OMR, the query in Listing 5.4 needs to be executed in the database. The query would result in a table similar to the one in Figure 5.5.

5.5.2 Use Case 2: Find All Functions in a Class

```sql
SELECT Namespace, Classname, Signature, FileID
FROM Function as f
INNER JOIN Class as c on c.id = f.class
IDWHERE className='CodeGenerator';
```

Listing 5.5: The query used to find all functions in a class

In order to find all functions in a class `CodeGenerator` in OMR, the query in Listing 5.5 needs to be executed in the database. The query would result in a table similar to the one in Figure 5.6.

namespace	classname	signature
OMR	CodeGenerator	addAOTRelocation(TR::Relocation *,TR::Reloca...
OMR	CodeGenerator	addAOTRelocation(TR::Relocation *,const char ...
OMR	CodeGenerator	addAccumulatedInstructionLengthError(int32_t)
OMR	CodeGenerator	addAllocatedRegister(TR::Register *)
OMR	CodeGenerator	addAllocatedRegisterPair(TR::RegisterPair *)
OMR	CodeGenerator	addBreakPointAddress(uint8_t *)
OMR	CodeGenerator	addCountersToEdges(TR::Block *)
OMR	CodeGenerator	addMonClass(TR::Node *,class TR_OpaqueClas...
OMR	CodeGenerator	addProjectSpecializedPairRelocation(uint8_t *,u...
OMR	CodeGenerator	addProjectSpecializedRelocation(uint8_t *,uint8...
OMR	CodeGenerator	addProjectSpecializedRelocation(TR::Instructio...
OMR	CodeGenerator	addRelocation(TR::Relocation *)
OMR	CodeGenerator	addSnippet(TR::Snippet *)

Figure 5.6: The results from the database when running the query in 5.5 to find all functions in a class

5.5.3 Use Case 3: Find a Location of the Definition of a Function Signature

```
1 SELECT Namespace, Classname, Signature, Location
2 FROM Function as f
3 INNER JOIN Class as c on c.id = f.classID
4 INNER JOIN File as l on l.id = f.fileID
5 WHERE namespace='OMR' and classname='CodeGenerator
```

Listing 5.6: The query used to find a location of the definition of a function signature

namespace	classname	signature	Location
OMR	CodeGenerator	afterRA()	omr/compiler/codegen/OMRCodeGenerator.hpp

Figure 5.7: The results from the database when running the query in 5.6 to find a location of the definition of a function signature

In order to find the location of function `OMR::CodeGenerator::afterRA()` in OMR, the query in Listing 5.6 needs to be executed in the database. The query would result in a table similar to the one in Figure 5.7.

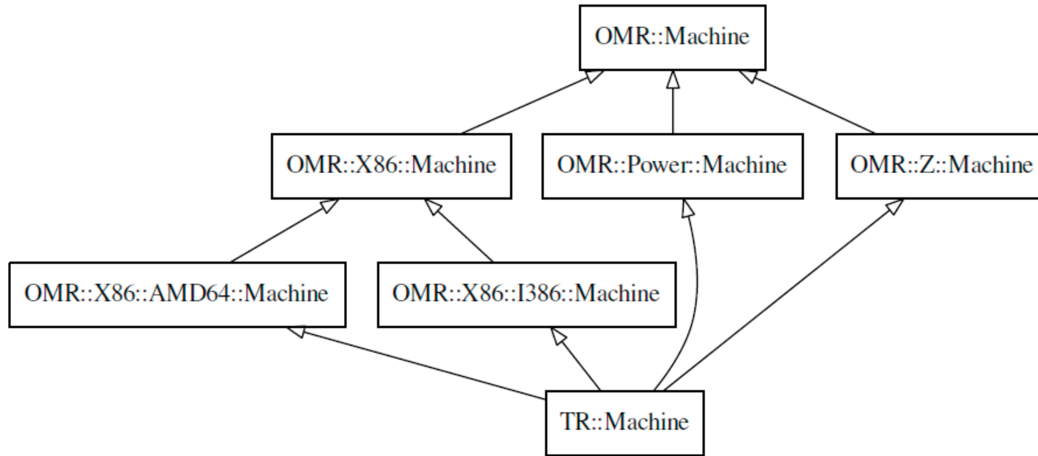

Figure 5.8: Part of the *graph.pdf* when viewing the hierarchy visualization in the fourth use case

5.5.4 Use Case 4: View the Class Hierarchy

The class Hierarchy of the `Machine` class is visualized in *tools/compiler/OMRStatistics/visualization/Hierarchy/graph.pdf*. To demonstrate that, the `Machine` hierarchy visualization in that PDF is shown in Figure 5.8.

Chapter 6

Assessing Dynamic Polymorphism

As mentioned in the use cases in the previous chapter, lots of information can be queried from the database to help OMR developers answer their questions. In order to decide which kind of polymorphism is best for this project, we query the database to answer the following questions that OMR developers were interested in:

- Q1: How many classes are in OMR altogether and how many of them are made into extensible classes?
- Q2: How many methods are there in all OMR extensible classes and how many of them are overridden in client/extension code?
- Q3: Is most functionality added through static polymorphism or through the addition of new functions in derived classes?

The aim of the first question is to understand the number of classes that have to be searched by client developers when extending OMR to find the extension points of the project. If that number is large, dynamic polymorphism can help (with its `virtual` keyword) highlight overridden functions and make it easier for developers to find functions that they would need to override. The second question aims to show us the percentage of overridden functions in OMR. This helps us determine the likelihood of performance degradation. If the number of overridden functions is low then it is less likely that considerable performance degradation will happen if functions are virtualized and vice

versa. Note that this is not a factor that can strictly determine the performance impact but it does help to make a more educated decision about the change. Additionally, this gives us an insight into the amount of work needed to virtualize all the functions in OMR. The third question is designed to give us insight about how OMR is expected to be used by client developers. We use OpenJ9, as the biggest consumer of OMR today, to determine how OMR is likely to be used by client developers.

We now discuss each of these questions more thoroughly, how we used OMRStatistics to answer it, and how answering that question helped OMR developers move one step closer to the best variability implementation decision for the project.

6.1 The Queries

We hereby present the queries used in our collection and processing of results. We refer to these in later sections of this chapter to show how we got our results. Some of the following queries were run on the database generated when running OMRStatistics on all architecture variants of OMR and others were run on the database generated when running OMRStatistics on all architecture variants of OpenJ9 and OMR together. We will elaborate on which database each query ran on later in this section. The databases were created in May 2018.

```
1 SELECT count(*)
2 FROM class
```

Listing 6.1: Query used to get the total number of classes in the source code.

```
1 SELECT count(*)
2 FROM class
3 WHERE isExtensible=1;
```

Listing 6.2: Query used to get the total number of *extensible* classes in the source code.

```sql
SELECT COUNT(DISTINCT oc.Namespace, oc.Classname)
FROM Override as o
INNER JOIN Function as bf on bf.id = o.BaseFunctionID
INNER JOIN Function as of on of.id = o.overridingFunctionID
INNER JOIN Class as bc on bc.id = bf.classID
INNER JOIN Class as oc on oc.id = of.classID
WHERE bc.isExtensible = 1 and oc.isExtensible = 1;
```

Listing 6.3: Query used to get the number of extensible classes that contain overridden functions

```sql
SELECT Namespace, Classname, Signature
FROM function as f
INNER JOIN class as c on c.id = f.classID
WHERE c.isExtensible = 1;
```

Listing 6.4: Query used to get the function signatures of all the functions inside extensible classes. It returns records where each record contains the function signature and the class name where this function belongs

```sql
SELECT DISTINCT bc.Namespace, bc.Classname, bf.Signature
FROM Override as o
INNER JOIN Function as bf on bf.id = o.BaseFunctionID
INNER JOIN Function as of on of.id = o.overridingFunctionID
INNER JOIN Class as bc on bc.id = bf.classID
INNER JOIN Class as oc on oc.id = of.classID
WHERE bc.isExtensible = 1 and oc.isExtensible = 1;
```

Listing 6.5: Query used to get the number of extensible classes that contain overridden functions

6.2 Data About Extensible Classes

Question 1: How many classes are in OMR altogether and how many of them are made into extensible classes? Running the queries in Listings 6.1 and Listings 6.2 on the database generated by running OMRStatistics on OMR show that 149 of the 1365 classes in OMR (~10.91%) are marked as extensible. Considering only the extensible classes, we ran the query in Listing 6.3 which showed that the functions which OMR downstream projects are expected to extend are spread in 104 classes (~7.62% of total classes). This means that OMR downstream language developers have to look through all of these classes and decide which functions they need to override to provide the desired behavior for their project while leveraging the rest of the OMR code. Searching through such a large number of classes for extension points is not ideal. With the switch to dynamic polymorphism, only the functions that are overridden will be made into `virtual` functions. This will make all possible extension points easier to find for OMR downstream project developers.

Finding 1

149 of the 1365 classes in OMR (~7.75%) are marked as extensible. Considering only the extensible classes, the functions which OMR downstream projects are expected to extend are spread in 161 classes (~8.37% of total classes).

6.3 Data About Overridden Functions

Question 2: How many methods are there in all OMR extensible classes and how many of them are overridden in client/extension code?

Running the query in Listing 6.4 on the database generated when running OMRStatistics on OMR results in a table with functions and what class each function belongs to. However, some functions might be overridden and hence would have multiple records in our results. Hence, we ran a python script that processes the results to get the total number of unique functions

in extensible classes, 8450. Similarly to the query in Listing 6.4, the one in Listing 6.5 produces a table with unique overridden functions and what class each function belongs to. Note that this is different from the previous query since it focuses on overridden functions and not the total number of functions. The python script also processes the results due to duplicate records for the same function to get the number of unique overridden functions in extensible classes, 855. The python script full path is: *omr/compiler/OMRStatistics/-sourceCodeProcessrs/statistics.py*; this script will be referred to till the end of this chapter since it is responsible for parsing all results discussed here.

Executing the above queries and processing the results with the python file reveals that OMR has 8,450 methods in extensible classes and only 855 of these methods, roughly 10.11%, are overridden. After presenting our results to OMR developers, they were encouraged by the small percentage of methods that would need to be virtualized, but further run-time profiling is needed to determine the possible run-time overhead since it depends on how often these function will be called.

Finding 2

OMR has 8,450 methods in extensible classes and only 855 of these methods, roughly 10.11%, are overridden.

6.4 Data About Extensible Class Hierarchies

Question 3: Is most functionality added through static polymorphism or through the addition of new functions in derived classes?

New functionality in OMR can be added either by adding new methods in derived classes or by overriding existing methods and altering their behavior. One of the tasks our python script that we referred to earlier does is get the number of extensible hierarchies and their average depth. The script basically counts the number of hierarchies from OMRStatistics *hierarchy.csv* output. It also uses the same output to get the average length of a hierarchy. Running the script on the *hierarchy.csv* showed that the OMR and OpenJ9 source codes

combined have 80 extensible hierarchies with average class hierarchy depth of 3.25 classes/hierarchy.

We also ran the queries in Listings 6.4 and 6.5 on the database obtained by running OMRStatistics on OMR and OpenJ9. After that, we processed the results with the python script to find the average percentage of overridden functions in an extensible hierarchy. More specifically, for every hierarchy, we get the number of unique overridden functions from 6.4 and the total number of unique functions from 6.5 and divide the first by the second to get the percentage of overridden function in that hierarchy. After that, we average out all the percentages to get our result. We found that on average, only 19.89% of an extensible hierarchy's functions are overridden. This implies that the majority of the variability points in OMR are not in the form of method overrides, but are instead in the form of adding new functionality in the derived classes. In other words, the low percentage of overridden functions suggests that existing client developers extended OMR mostly by adding new functionality in derived classes instead of overriding existing functions. The last also implies that new language developers are more likely to extend OMR by adding new functions. Hence, moving from static polymorphism to dynamic polymorphism will only affect a low percentage of functions and variability points in OMR and its current client (OpenJ9) and is likely to affect a low percentage of functions and variability points in new client languages. Hence, the change is likely to have a minimal impact on the performance of OMR.

Finding 3

> We find 80 extensible hierarchies and calculated the average class hierarchy depth to be 3.25 classes. We also find that on average, only 19.89% of an extensible hierarchy's functions are overridden.

Note that while we only answer these three questions here, the data gathered by OMRStatistics in the database allows OMR developers to query for additional information about the class hierarchies and function overloading/overriding in OMR and OpenJ9.

The facts and data in this chapter suggest that there are no data that

strongly discourages the decision to change from static to dynamic polymorphism.

Chapter 7

The Road to Virtualization in OMR

Altering a big project's variability implementation mechanism is not a straightforward task; it includes lots of unexpected challenges which require straying from or even changing the plan to solve them. In this chapter, we discuss our work towards the third objective of this thesis (see Section 1.4). This chapter also presents the process used to achieve the third contribution of this thesis (see Section 1.5).

More specifically, we first describe the discussions we had with the people that would be affected by that change and how these discussions changed the plans we had at the time. We later formalize the process used to virtualize the functions and describe it here. Then, we talk about a Python tool that we created to help us virtualize the functions faster. After that we discuss the challenges that we faced when virtualizing and pushing the code to the OMR code base and how are we overcoming them. Finally, we show the results that we got after virtualizing a significant part of the project.

7.1 Convincing the OMR Team

In our research, we worked directly with part of the IBM compiler's team, precisely, two people from that team. However, our research aimed to change the variability implementation mechanism of the whole OMR project. Hence, the change would affect the whole IBM compiler's team and the rest of the OMR

community. Therefore, before changing OMR's variability implementation, we needed to discuss our change with the stakeholders and get their approval. We here talk about how discussing our proposal with them changed the track of our research.

As a first step, we participated in the OMR Compiler Architecture Meeting [77] (we participated in the meeting on June 6 [3]). The meeting result changed our research track. Our initial plan for virtualization was to only virtualize overridden functions that are intended to be used by client code. For example, if a function was overridden in the `CodeGenerator` hierarchy but was not intended to be used by `lang::CodeGenerator`, then such function will keep using static polymorphism. The idea behind this was to have *Hybrid polymorphism* implementation in OMR (the use of both static and dynamic polymorphism) and to use the `virtual` keyword (also known as dynamic polymorphism) to highlight what functions are supposed to be overridden for language developers. However, after discussing the idea in the OMR Compiler Architecture Meeting, the plan changed to virtualizing *all* the overridden functions in OMR to keep the consistency in the project. After that, the plan would be to benchmark OMR by the IBM compilers team. Stakeholders can then evaluate whether any runtime performance degradation is an acceptable trade-off for the convenience that is obtained from dynamic polymorphism. Hence, the approval of the change to dynamic polymorphism would depend on how much degradation in performance would happen.

To put it in concrete steps, the plan was as follows: (1) virtualize the functions that are supposed to be overridden by client developers in the `CodeGenerator` hierarchy, (2) submit a pull request(PR) with the change, (3) the team benchmarks OMR with our changes, (4) if no significant change in runtime performance happens, the change is accepted, (5) part of the future work is to do the same for other classes. We picked `CodeGenerator` hierarchy here as the class hierarchy with the most amount of overridden functions in OMR.

Whereas the OMR Compiler Architecture Meeting changed the plan to become as follows: (1) virtualize ***all*** overridden functions in the first ***twenty*** hierarchies with the most amount of overridden functions, (2) submit PRs

with the change for each hierarchy, (3) the team benchmarks OMR with our changes, (4) if no significant change in runtime performance happens, our changes are accepted. However, due to challenges that we discuss later in this chapter, we virtualize almost all of the `CodeGenerator` class in our work, in addition to a simpler class called `ELFRelocationResolver`, and leave virtualizing the last two functions of the `CodeGenerator` class and other hierarchies for future work.

7.2 The Process of Virtualization

The `CodeGenerator` hierarchy has more than 180 overridden functions that were candidates of virtualizations. In order to virtualize all functions systematically and minimize errors, we formalized a process that we follow; the process is described below:

1. Go to the headers of all the classes in the target hierarchy and remove the `OMR_EXTENSIBLE` tag. The reason behind this is that when virtualizing the functions, we remove the `self()` keyword. Hence, the linter, *OMRChecker*, would complain when running on that code. Therefore, we remove `OMR_EXTENSIBLE` from all the classes in the hierarchy to signify to the linter not to check this hierarchy for `self()` calls.

2. Add the `virtual` keyword to the first declaration of every overridden function. We say the first declaration since overridden functions are declared many times through the hierarchy. In *C++*, we only need to virtualize the first occurrence of that function when traversing the hierarchy from the base class to the most derived class in that hierarchy.

3. Remove the *self()* from all calls to virtualized member functions. The reason behind this is that calls for virtualized functions are resolved at runtime, which eliminates the problem that was being solved by the `self()` keyword. Hence, we can safely remove the keyword from all member function calls.

```
1 namespace OMR {
2    class OMR_EXTENSIBLE ELFRelocationResolver {
3    public:
4       uint32_t resolveRelocationType(const TR::StaticRelocation &relocation);
5    };
6 }
```

Listing 7.1: `OMR::ElfRelocationResolver`'s original header file.

```
1 namespace OMR {
2    class OMR_EXTENSIBLE ELFRelocationResolver {
3    public:
4       virtual uint32_t resolveRelocationType(const TR::StaticRelocation &relocation);
5    };
6 }
```

Listing 7.2: `OMR::ElfRelocationResolver`'s header with our changes.

4. Rebuild the project with the changes and make sure that our changes did not break the build.

5. Run the resultant Java binary using the command: `java -version` to make sure that the executable runs without crashing.

6. Submit a PR with the changes.

In order to test the process, we virtualized a simple class hierarchy in OMR called `ELFRelocationResolved` and submitted a PR [102] with our changes. The class header before and after our virtualization is shown in Listings 7.1 and 7.2 respectively. We basically just add the `virtual` keyword in front of the only function in the class since this function is overridden in classes derived from `OMR::ElfRelocationResolver` and there were no calls for that function that use `self()`.

7.3 Manual Virtualization Helper (MVH)

To virtualize classes easily and more efficiently, we created a Python tool that helps virtualizing the classes and removing the `self() calls`. The tool resides in *omr/tools/compilers/OMRStatistics/sourceCodeProcessors/ManualVirtualization-Helper* directory. This tool is surrounded by helper Python scripts, all connected by a Makefile system; we call the whole system *Manual Virtualization Helper* (MVH).

```
1 FROM Override as o
2 SELECT DISTINCT bc.namespace, bc.classname, of.signature, oc.namespace, oc.classname
3 INNER JOIN Function as bf on bf.id = o.BaseFunctionID
4 INNER JOIN Function as of on of.id = o.overridingFunctionID
5 INNER JOIN Class as bc on bc.id = bf.classID
6 INNER JOIN Class as oc on oc.id = of.classID
7 WHERE bc.isExtensible = 1 and oc.isExtensible = 1 and bf.isVirtual = 0
8 and of.isVirtual = 0 and bf.isImplicit = 0 and of.isImplicit = 0
9 and bc.className=CodeGenerator;
```

Listing 7.3: SQL Query used for MVH.

7.3.1 Overview

The system aims to automate the virtualization of classes in OMR. The system does the following:

- Virtualizes function definitions (adds `virtual` to the beginning of function definitions)
- Removes `self()->` from all calls for the target functions
- Detects functions that are defined in the header file of a class but never implemented in that class and warns the user about them.

7.3.2 Process

More specifically, MVH follows the process below to achieve its objectives.

- Extract the list of functions that need to be virtualized from the output of the database query in Listing 7.3.
- Search the header files of class hierarchies from base to the most derived class and virtualize the first occurrence of the function definition. If multiple definitions are found in the same header (i.e., overloads), the function is ignored and a warning for manual virtualization is issued.
- Before the tool runs, a Makefile greps for the function calls in the source code. MVH uses the results to remove any `self()` call of these functions.
- Before the tool runs, a Makefile greps for the function implementations in the source code. MVH uses the results, in addition to searching the

header files for function implementations, to find declared but unimplemented functions in a class. Then it warns the user about unimplemented functions. Declared but unimplemented functions are dangerous when virtualizing overridden functions; they are the reason behind a challenge we faced and we discuss in Section 7.4.3.

7.3.3 Configuration

The system needs to be configured as follows to work correctly:

- In the Makefile, the following configuration should be set:
  - `OMR_PATH` and `OPENJ9_PATH`: Location of OMR and OpenJ9 directories to read and virtualize functions.
  - `PATH_TO_DB_QUERY`: Location of file containing the query mentioned in 7.3 when run in the database.
  - `TARGET_CLASS`: Class to be virtualized.
- In *processSearchResults.py*, the following verbosity configuration should be set:
  - `PRINT_OVERLOADS`: Triggers MVH to print the cases where an overload is detected.
  - `PRINT_NO_IMPLEMENTATION`: Triggers MVH to print the cases where no implementation for the function is detected.

Assumptions, supported cases, and unsupported case for MVH are documented in our corresponding Github repository [68].

7.4 Challenges

While virtualizing the functions in `CodeGenerator` we ran into multiple problems. Some issues were handled manually; other issues motivated us to go back and upgrade MVH to handle those issues; other issues were outside our control such as cases were the OMR design have to be changed. In cases where the

solution was outside our control, we started a discussion in the Eclipse OMR community to find a solution for these problems. We now discuss the different challenges we ran into and how we resolved each of them.

7.4.1 Picking the Right Commit

One problem we had when submitting the PRs is picking the right commit to build our changes upon. The OMR repository and OpenJ9 (discussed in Section 5.2) repository are developed independently and sometimes commits in OMR temporarily break the ones in OpenJ9. Hence, when picking the commit on which we rebase or apply our work on in the OMR and OpenJ9 Github repositories, we had to first clone the projects with the picked commits and try building them without introducing our changes. Only after making sure they compiled and ran some tests successfully, could we safely reapply our work over those commits.

7.4.2 Finding the problematic functions

After finding the right commits to build upon, we tried virtualizing all of the 180+ overridden functions in the `CodeGenerator` class hierarchy and rebuilding the project; the build failed. More specifically, the build included a Java binary executable which was giving a *segmentation fault* [99]. Debugging the segmentation fault was challenging since building the project depended on OMR's build system, which had no straightforward option for generating debugging metadata. Hence, we could not use debuggers like *gdb* [42] to find the error. We had to ask the IBM team how to generate such debugging data. They redirected us to a blog post [27] of an IBM developer who had similar problems before and needed to debug his code. The blog post describes how the IBM developer hacked his way through the running environment and build system to generate debugging information and running gdb in that environment. In addition, we had no information or documentation about that crashing Java binary; understanding its functionality would require an intermediate understanding of OMR's build system. Hence, after getting the debugging information, we had to collaborate with the IBM team through

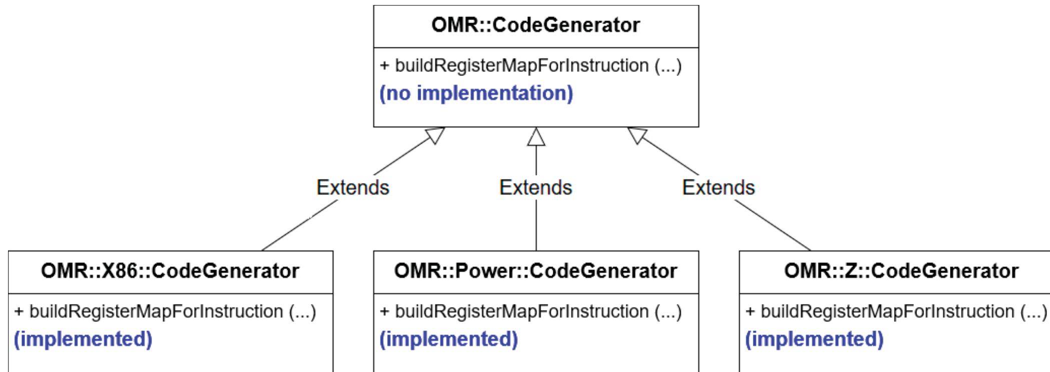

Figure 7.1: Visualization of `buildRegisterMapForInstruction` function that is declared but has no implementation in `OMR::CodeGenerator`

joint debugging sessions to understand what was breaking the build process. Collaborating with the IBM team was also time-consuming due to their busy schedules during the period of our research.

Hence, we had to introduce our changes in phases and build them locally in order to find the error. We introduced the changes in batches of around 20 functions each. If building the project with that change resulted in a working Java binary, we would virtualize another 20 functions. If not, we would try building the project with only the first 10 of these functions virtualized and so on. We keep narrowing the changes down to get to the point where we virtualize only one function. When such a build results with a successful Java binary, it implies that the changed function was not the problematic one. If that build resulted with a Java binary that makes a segmentation fault, it verifies that the changed function was a problematic function. We did this with all the functions and discovered all problematic functions which we later further inspect and describe the source of the problem in the following.

7.4.3 Problem 1: Functions with no Implementations

Some functions such as `void buildRegisterMapForInstruction(TR_GCStackMap*)` are declared in a base class (`OMR::CodeGenerator`) but have no implementation in that class. Their implementation comes in all the derived classes instead (`OMR::X86::CodeGenerator`, `OMR::P::CodeGenerator`, `OMR::Z::CodeGenerator`) as visualized in Figure 7.1. These functions were legal when static polymor-

phism was used however when virtualized, the compiler complained about such functions. We had to declare them as *pure virtual* [100] instead. Declaring a function as pure virtual is promising the compiler that the function declared to be pure virtual will be defined in a derived class.

We also encountered a function, `TR::Instruction * generateSwitchTo-InterpreterPrePrologue(TR::Instruction *prev, uint8_t alignment,-uint8_t alignmentMargin)`, that was declared in `OMR::X86::CodeGenerator` but implemented in `J9::X86::CodeGenerator`, which means it is only defined in the OpenJ9 project. Such cases cannot be declared as pure virtual since when declared pure virtual, building the OMR project independently from OpenJ9 would fail. This is because the OMR project would have no implementation for a pure virtual function. In such cases, we had to create an empty implementation for the function (returning a null pointer in this case).

7.4.4 Problem 2: Functions called in constructors

Virtualizing functions that are called in a constructor form a challenge since virtual calls in constructors are not resolved as expected. Consider the example in Figure 7.2. The problem arises when an object of type `J9::X86::AMD64::-CodeGenerator` is constructed; the current class' contructor calls its parent classes' constructors, including the `OMR::X86::AMD64::CodeGenerator` constructor. As shown in the implementation of that constructor in Listing 7.4, the `initialize` function is called, which resolves to the implementation in `OMR::X86::CodeGenerator` since it is the function's most specific implementation in the `CodeGenerator` class hierarchy. In `OMR::X86::CodeGenerator::-initialize`, `initializeLinkage` function is called, which is only implemented in `OMR::CodeGenerator` as per the UML in Figure 7.2. That implementation contains a call to the `createLinkage` function, which is overridden as shown in the UML in Figure 7.2. Hence, `createLinkage` is a candidate of virtualization.

With static polymorphism, calling `self()->createLinkage()` would resolve to the function's implementation in `J9::X86::AMD64CodeGenerator` which is the most specific implementation in the hierarchy. However, when virtualized and `self()` is removed, *C++* would not call the implementation in a

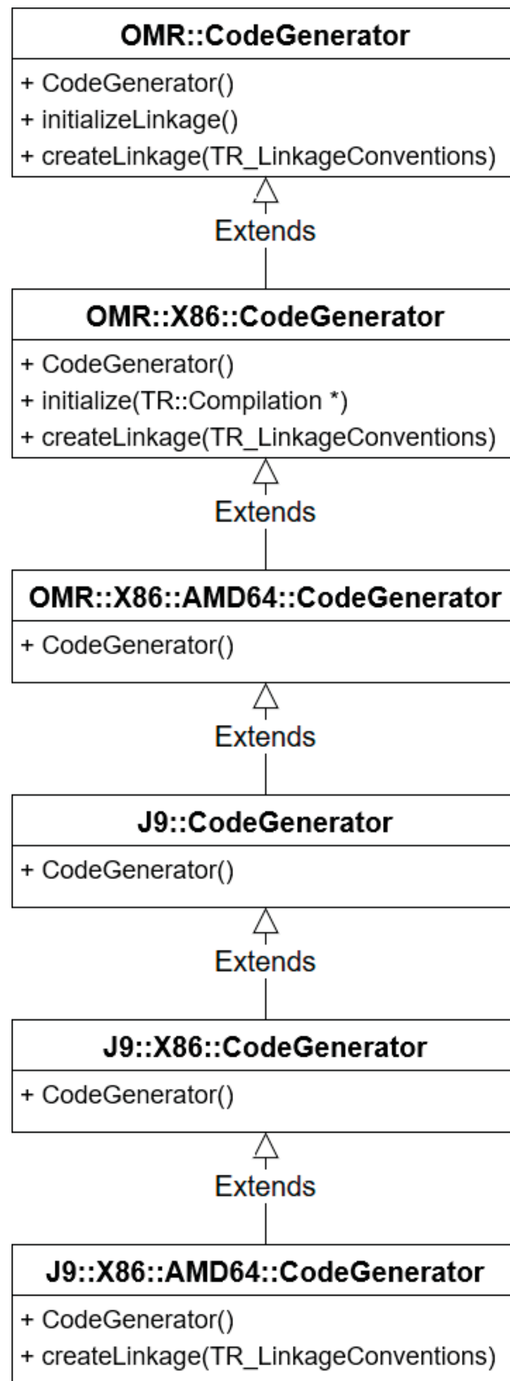

Figure 7.2: UML of an example of a call of a virtual function in a `CodeGenerator` class constructor

```
1  OMR::X86::AMD64::CodeGenerator() {
2     ...
3     self()->initialize(...)
4     ...
5  }
6  OMR::X86::CodeGenerator::initialize(TR::Compilation *comp) {
7     ...
8     self()->initializeLinkage()
9     ...
10 }
11 OMR::CodeGenerator::initializeLinkage() {
12    ...
13    self()->createLinkage()
14    ...
15 }
```

Listing 7.4: Code of the example of a call of a virtual function in a `CodeGenerator` class constructor

class that is not fully constructed [101]. In other words, the implementation of `J9::X86::AMD64::CodeGenerator::createLinkage` would not be called since the object is not fully constructed yet. Instead, the implementation in the next class up the hierarchy that has the implementation of that function (i.e., `OMR::X86::CodeGenerator::createLinkage`) would be called. This would lead `createLinkage` function to not behave as expected since its most specific implementation for the *J9* project is not triggered.

We found two functions with this case which we did not virtualize yet: `createLinkageForCompilation()` and `createLinkage(TR_LinkageConventions)`. In order to solve this problem, we suggest the following solution. The two problematic functions both have implementations that are not dependent on the state or member functions of the object being constructed. Hence, it is safe to export them to a different class. Therefore, we suggest creating a class, `LinkagePolicy` that only has those two functions. The class would be declared in all the namespaces of OMR, like other extensible classes in the project, which allows the flexibility of having different implementations of the functions for different architectures. We then instantiate a `LinkagePolicy` object before every call for a problematic function and call the problematic function as a member of that `LinkagePolicy` instance. For example, instead of calling `createLinkage()`, we first create an instance of the `LinkagePolicy` class, let's call it `policyInstance`, and then we call `policyInstance.createLinkage()`.

Although this is a plausible solution, we cannot be taking design decisions for the OMR community. The issue is raised in Github [89] for the OMR community to discuss it. However, we have no consensus on a solution yet. We need to wait for the community's decision and act accordingly

7.4.5 Problem 3: Submitting PRs

Submitting PRs with our changes was tricky due to the online tests that run on our changes. Our changes were targetting both OMR and OpenJ9 source codes which were two different repositories. Every time we update the PR on one repository, the project is built with our changes and tested on their *Travis CI* build server.

Travis CI has different settings in each repository and hence works differently. For OMR, the project would be built independantly from OpenJ9 which raised challenges similar to the one described above (in Section 7.4.3 for function `TR::Instruction * generateSwitchToInterpreterPrePrologue-(TR::Instruction *prev, uint8_t alignment, uint8_t alignmentMargin)`). For OpenJ9, the project would be built in without taking our changes in the OMR PR into consideration. So for example, it would assume the `CodeGenerator` classes are tagged with `OMR_EXTENSIBLE` and hence need the `self()` function in every member function call, which would cause the PR to have errors. To solve this, we indicate in the OpenJ9 PR description that it depends on a PR in the OMR repository. This allows the OMR team to change the settings and base the OpenJ9 PR on its corresponding one in the OMR repository.

7.4.6 Problem 6: Infrastructure Issues

To verify that our changes did not break any functionality in the project, the binaries built from our PR are tested on IBM proprietary test suits. These tests can only be accessed by the IBM team. Hence, we had to pass our changes to them to be tested. Based on the feedback of such tests, we sometimes had to schedule more debugging sessions to fix the errors. Scheduling sessions that fit in all the participants' schedules resulted in some delays in our research which we tried to use for writing papers and other artifacts.

7.5 Benchmarking

7.5.1 Setup

In order to benchmark Eclipse OMR and check if there was any significant degradation of runtime performance, we ran two benchmarking software: the first is a Java web application that runs on WebSphere Liberty [62]. The second is a rule engine benchmarked by a set of rules. In both benchmarks, we ran the setup using two versions of Eclipse OMR: the altered Eclipse OMR that uses dynamic polymorphism in its variability implementation mechanism and the original Eclipse OMR before we did our changes. Eclipse OMR that was used as our experimental control group is built on the commit of has: *31d-44e18ee711b11a750755e81d6f9bc7f2a9806*. It is the last commit before we did our changes. On the other hand, the Eclipse OMR that contained our changes is built on the commit of hash: *a9a70c316b452bc61afae7530e3147f273546ee9* which marks the last of our changes. It is worthy to note that some of the OMR components are not utilized by these tests. Those components include OMR's language-independant testing framework, *fvtest*, and the OMR-specific tools contained under the *tools* component and directory. However, our changes were mostly in the compiler component and we verified that this part is being tested by both benchmarks. It is also worthy to note that both benchmarking software are IBM proprietary products, hence we did not have access to them; instead, part of the compiler's team who was working with us on this project ran the tests and sent us the results.

The first benchmark test is called *DayTader3* and is based on WebSphere Liberty, a Java application server that uses OpenJ9 [105] (which uses the Eclipse OMR compiler component) to compile the Java web applications running on it. The machine on which that benchmark ran is an IBM machine called *PoisonIvy*. It has an *Intel 'E5-2650 v2 Ivybridge' AMD64 (64-bit)* central processing unit, 144 Gigabytes of random-access memory, running on a *Linux - RedHat Enterprise Linux 7 GA x86-64* operating system. The web application is an online trading system [51]. The test basically gives the web application a large number of transactions, and the throughput is measured as

the number of transactions that were completed by the server. The throughput tracking happens after the warmup is done. For this benchmark, the warmup was running transactions on the server for 540 seconds before the throughput tracking started. A transaction in our case is an act done by the user; such as buying, selling, viewing portfolio, looking up stock prices... Two configurations were used in the test, *LargeThreadPool* and *LargeThreadPoolWarm*. The latter optimizes the performance of Eclipse OMR in general. Both configurations have been traditionally used by IBM developers when benchmarking, hence we followed their steps and got results for both benchmarks. Since WebSphere Liberty uses OpenJ9 and Eclipse OMR to compile and run the web application that ran on it, measuring the throughput, the number of transactions that were completed, gives us insight about the runtime performance of Eclipse OMR.

The second benchmark test, called *ODM*, is a business rule management system [50] which contains a rule engine. The rule engine is configured with a set of rules and makes decisions based on these rules. The engine is configured to use OpenJ9 and Eclipse OMR to compile and run its Java code. After warming up the Java Virtual Machine (OpenJ9) for 240 seconds, the benchmark here is to test the engine with two sets of rules, *Segmentation300-RulesFastpathRVEJB* and *Segmentation5FastpathRVEJB*, then measuring the throughput. The throughput, in this case, is the number of decisions completed. Since Eclipse OMR is used in running the rule engine, the throughput is an indicator of the runtime performance of OMR. The algorithm used by the engine is called *Fastpath* [37].

7.5.2 Results

After running the benchmarks 10 times, we were provided with the average throughput (mean), standard deviation, minimum throughput, and maximum throughput. We used these values to plot Figures 7.3, 7.4, 7.5, and 7.6. The throughput changes are negligible and can be associated to other variables in the experiment. In Figures 7.3 and 7.5 (DayTrader3-LargeThreadPool and ODM-Seg300 respectively), we observe a minor increase in throughput; when

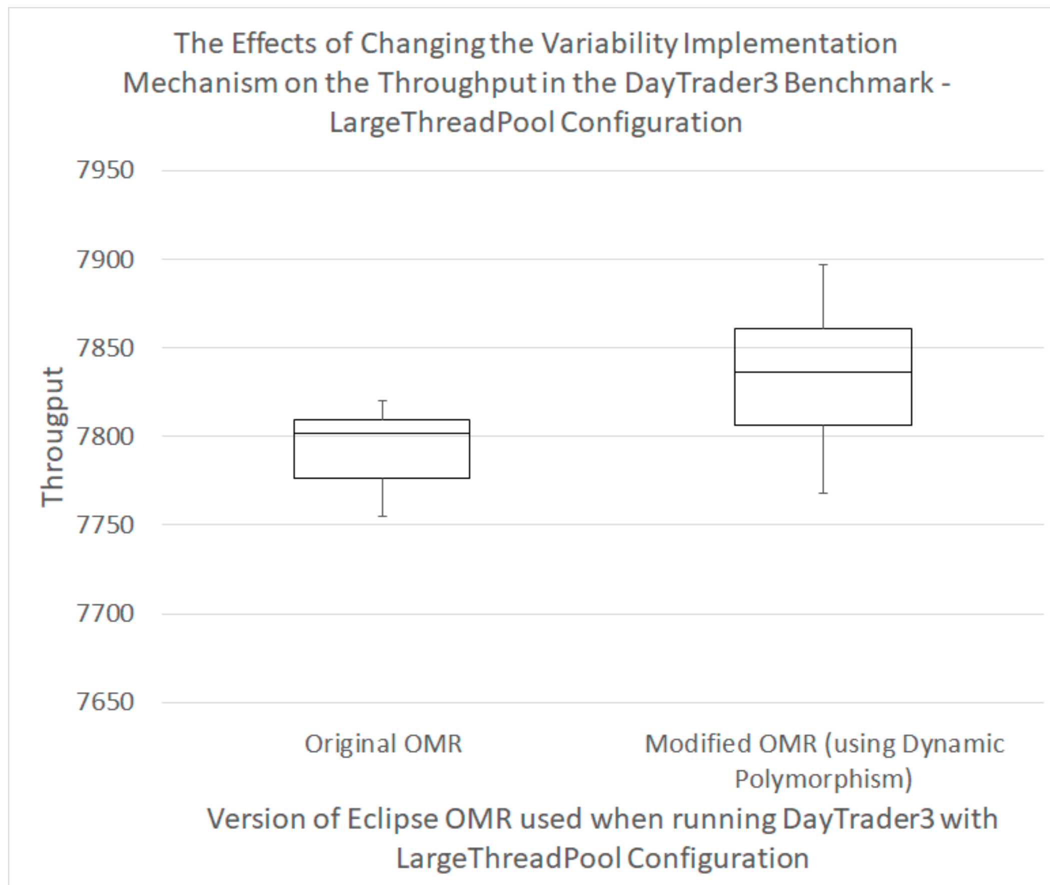

Figure 7.3: The Effects of Changing the Variability Implementation Mechanism on the Throughput in the DayTader3 Benchmark - LargeThreadPool Configuration

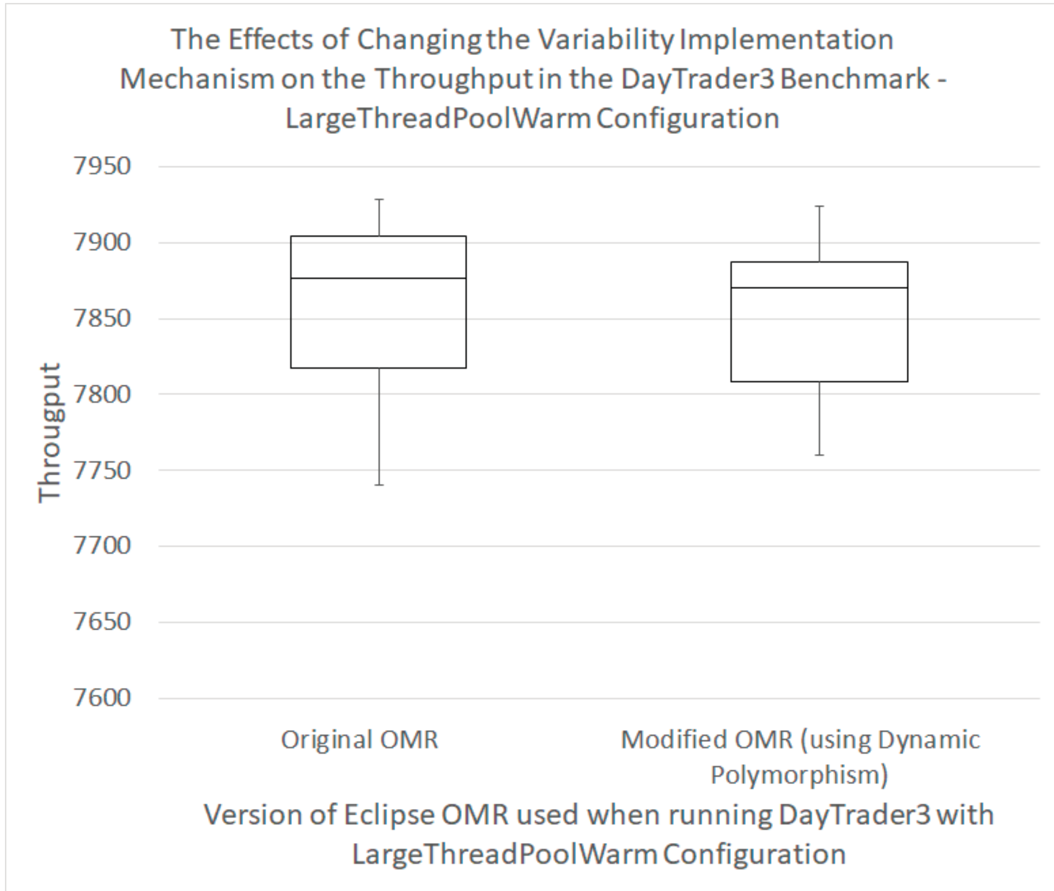

Figure 7.4: The Effects of Changing the Variability Implementation Mechanism on the Throughput in the DayTader3 Benchmark - LargeThreadPool-Warm Configuration

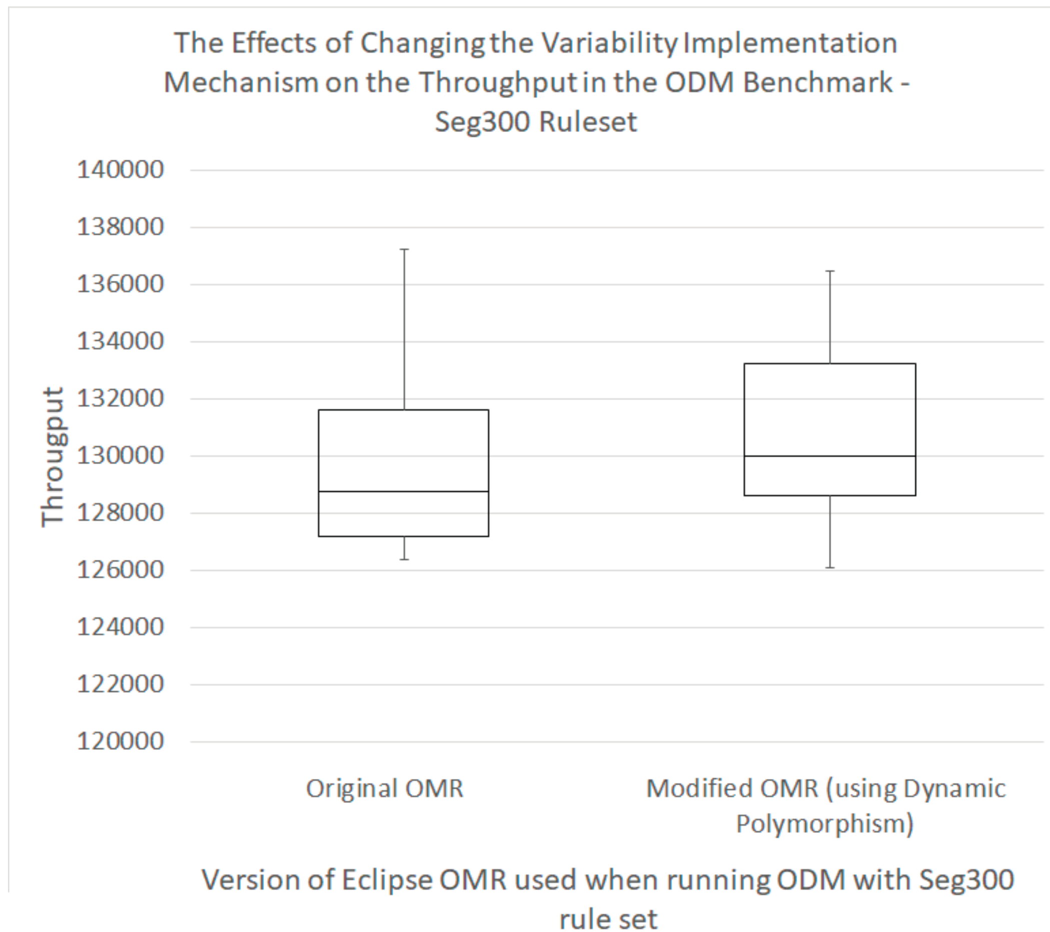

Figure 7.5: The Effects of Changing the Variability Implementation Mechanism on the Throughput in the ODM Benchmark - Seg300 Ruleset

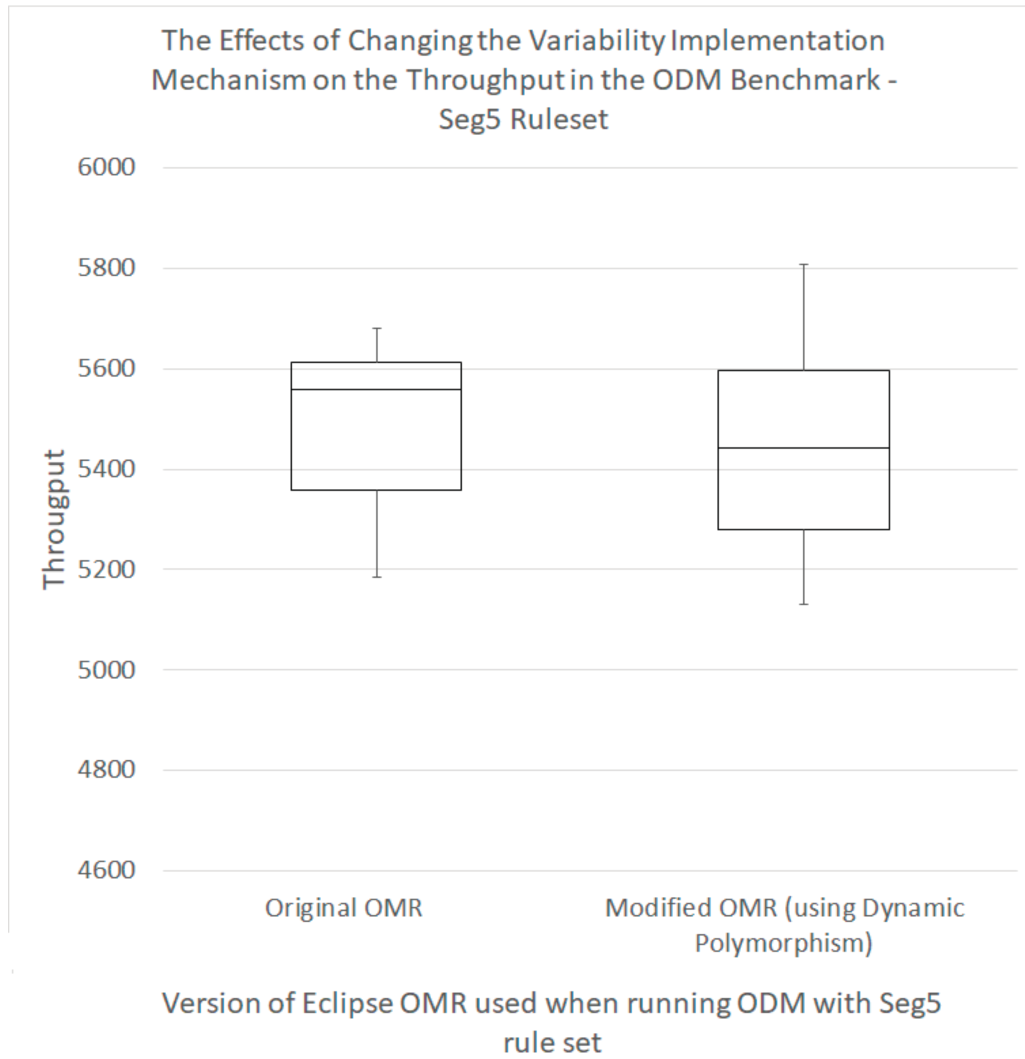

Figure 7.6: The Effects of Changing the Variability Implementation Mechanism on the Throughput in the ODM Benchmark - Seg5 Ruleset

comparing the means, the increase is 0.5% and 1.2% respectively which is considered negligible. On the other, in Figures 7.4 and 7.6 (DayTrader3-LargeThreadPoolWarm and OMD-Seg5 respectively), we observe a minor decrease in the throughput (the means decrease 0.8% and 0.1% respectively) which is also considered negligible.

Chapter 8

Discussion

So far in the thesis, we described how we collected information in order to encourage or discourage OMR developers to consider moving from using static to using dynamic polymorphism in their variability implementation mechanism. We created OMRStatistics, which was our main resource to collect information, and found out that: (1) ~10.61% of the classes are expected to be extended by developers. (2) The functions that are overridden and expected to be overridden by client developers are spread in only ~8.37% of the classes. (3) When taking that in terms of the total number of functions, only ~10.11% of all functions are overridden and expected to be overridden by client developers. (4) When taking that in terms of extensible class hierarchies, less than ~20% of a class hierarchy's functions are expected to be extended. Those results encouraged OMR developers to consider changing the variability implementation mechanism from using static polymorphism to using dynamic polymorphism.

After that, we helped OMR developers change the variability implementation mechanism by creating MVH, a Python tool that helps with virtualizing the functions. While virtualizing, multiple problems arose. Some of them raised open questions that we did not expect earlier. For example, there were overridden member functions that are called in constructors; virtualizing such functions caused errors. The solution we came up with, after discussions with our IBM collaborators in this research, is to export those functions into another class so that when they are called in a constructor, those functions would

not be member functions of the object being constructed anymore and hence would not cause any issue. However, the solution for this problem is a design decision that needs to be taken by all the OMR developer team (what we referred to earlier as the *Bigger Team*). Hence, we did not implement our suggested solution as part of our changes yet; we are awaiting the decision of the OMR developers team.

8.1 Answering the Research Questions

In the introduction, we proposed the following research questions: What are the practical implications of using static polymorphism versus dynamic polymorphism? Are there any consequences for changing the variability implementation mechanism that use static polymorphism to use dynamic polymorphism?

After working on this project, we realized an important implication of static polymorphism in *C++*. When a polymorphic function is called, *C++* is designed to look at the declaration of the callee class declaration and all the parent class declarations of that callee, it is not designed to look at children class declarations. This, when mixed with static polymorphism's static function call resolution, ends up in a roadblock for developers using such an implementation. The problem described in Section 4.1 is an example of a general problem any developer would face when using static polymorphism in *C++*. When calling `instance.a()` in Figure 4.1, expecting functions `A::a()` and `B::b()` to run is not an expectation specific for the OMR implementation; it is the intuitive behavior that is the case in dynamic polymorphism. However, it is not the case in static polymorphism due to the static function call resolution.

There are multiple ways to get around that problem, including static casting to the derived class before calling member functions. This idea is commonly used in CRTP (see Section 2.3). Building on that idea, OMR developers decided to cast directly to the most derived class of a hierarchy, instead of casting to the directly derived class, as done in CRTP. Although this solves the

problem described in Section 4.1, adding a static cast every time a member function is called makes the code more complicated and harder to understand for outside developers.

On another hand, due to the dynamic polymorphism implementation in *C++* a class that has an unimplemented virtual function is considered an *abstract* class and is not allowed to be instantiated. This formed a challenge for us when changing OMR's variability implementation mechanism to use dynamic polymorphism; since some functions were declared in a class that has multiple children classes but only defined in one of these children classes. In static polymorphism, instantiating any of the children classes with that function unimplemented would not cause an error as long as the function is not called. However, in dynamic polymorphism, children classes with the function unimplemented are considered abstract and hence instantiating such children throws a compilation error. This is practically a downside of static polymorphism; since static polymorphism allows cases of instantiating classes with unimplemented functions to pass without the compiler complaining.

Coming to dynamic polymorphism, the practical downside we faced, when comparing it to static polymorphism, was the additional level of indirection introduced which slightly degrades the runtime performance. As shown in Section 7.5, virtualizing the class hierarchy with the most number of overridden functions in OMR did negligible degradation to the runtime performance. Although negligible now, degradation might increase in the future if calls to polymorphic functions increased heavily in OMR or client code. Hence, moving to dynamic polymorphism comes with the constant concern of future performance degradation if virtual calls are abused. In other words, moving to dynamic polymorphism limits the development of OMR and client source code to calling polymorphic functions a minimal amount of times.

When discussing static and dynamic polymorphism, it is interesting to see OMR's implementation mechanism in terms of the two kinds of polymorphism. Eclipse OMR's implementation uses static polymorphism in a similar way to how CRTP uses static polymorphism since static casting to derived classes is used. However, Eclipse OMR's implementation uses protocols used in dy-

namic polymorphism but not in static polymorphism such as abstract classes; since OMR's implementation does not allow the instantiation of any extensible classes other than the ones in the `TR` namespace, it is implicitly declaring them as abstract. Here we can see how OMR's implementation used static polymorphism for its optimized runtime performance but also used concepts that are used in dynamic polymorphism but not in static polymorphism (abstract classes) to hide the consequences of static polymorphism. Since static polymorphism allowed classes with unimplemented functions to be instantiated, OMR developers tried to forbid instantiating most of these classes by *abstracting* all classes in non-`TR` namespaces.

8.2 Implementation that Best Fits Eclipse OMR

In our research, we take a specific variability implementation mechanism that is used in the industry but not discussed in literature and we try to alter it to make it better. We do realize however that our altered implementation mechanism might not be the *best* fit for the project since the *best* fit depends on multiple components that we cannot necessarily consider in this research. To decide on the best implementation for OMR we need to consider the following:

- *Maintainability*: A variability implementation mechanism can be judged based on how maintainable it is. Here come components such as whether a project has code style guidelines that successfully foster readability; how enforced are such guidelines and how consistent the source code of a project is in that sense; how much is the separation of concerns principle applied in a project. Although this is an important component of an implementation that might affect how good is it for projects in general and OMR in specific, more longitudinal studies are needed to determine this.

- *Long Term Analysis*: One factor that needs to be considered when deciding whether the OMR's altered variability implementation mechanism is a better alternative of the older one is to measure the number of inter-

actions that happen with the project and compare them to the number of interactions that happened before. Part of the future work of this research is to measure the change in frequency of asked questions about OMR in forums and emails and the number of PRs and contributions to study if these numbers are increasing or decreasing. Although the change of the variability implementation is not the only cause for these changes (after all, the project is getting more exposure over time due to marketing and other outreach methods), however the change in these numbers, combined with other factors, can be an indicator of the success of the implementation mechanism in improving OMR's outreach to the community.

- *Developer Opinions*: A successful implementation mechanism in the software industry, when viewed from a business perspective, is the one that is least costly to implement and maintain. Maintaining a software project with minimal costs is highly correlated with minimal person-hours of work. When working on a project, developers need to be comfortable with a variability implementation mechanism to be able to develop it quicker. Hence, one component to take into consideration when discussing the success of an implementation is how familiar and compatible developers are with it. Part of the future work of this research is to survey OMR developers and get their opinions on the altered variability implementation mechanism in OMR.

  Another important factor here is the readability of the code since when a code is more readable it is more understandable by developers and easier to maintain. This is one of the main motivations for the implementation change we are leading in Eclipse OMR.

- *Performance*: Last but not least, performance is one of the main factors of measuring the success of an implementation. When looking at Eclipse OMR as a library of runtime components, performance is a priority when measuring an implementation's success. In order to respect that while changing the variability implementation mechanism, in our research we

benchmark Eclipse OMR after our changes to ensure that a minimal degradation of runtime performance.

8.3 Open Questions

One open question was raised in the previous section, that is: what variability implementation best fits Eclipse OMR? We discussed above the factors that may be necessarily considered when judging the best implementation for a project. Hence, we do not necessarily consider an implementation that uses dynamic polymorphism as the best. After all, there are many variability implementation mechanisms out there and some of them might do better on the criteria we suggested above than implementations that use polymorphism in general. However, we did come up with this decision by taking developers' opinions into consideration (see Section 7.1) and benchmarking performance (see Section 7.5). Therefore, we believe that using dynamic polymorphism in OMR's variability implementation mechanism would better fit the project than using static polymorphism.

Another open question here is how much good does the removal of `self()` do to the project. Although based on our discussions with the team, the common opinion is that removing the `self()` function does increase the readability of the code, we have no concrete proof of that. In order to get concrete proof, long-term analysis and surveying the developers are needed, which are part of the future work of this research as indicated in the previous section.

When virtualizing the `CodeGenerator` hierarchy, we ran into multiple problems which challenged us and triggered us to change fundamental parts of the code to fix them. For example, virtual functions called in the constructor (see Section 7.4.4) might cause us to change all calls to the constructor of `OMR::X86::CodeGenerator` classes. It is worthy to note that we did not know about this problem when we started virtualizing the functions; we only discovered this problem by virtualizing the functions and then running the functionality tests of OMR. Although we are now aware of all the problems that happen when virtualizing the overridden functions in the `CodeGenerator` class

hierarchy, we are still not sure about problems that would arise when virtualizing overridden functions in other class hierarchies. Hypothetically, there might be problems that need more critical changes than the *virtual function calls in constructor* problem and Eclipse OMR developers might not be okay with the change needed to fix such a (hypothetical) problem. This will keep being an open question till all functions in the project are virtualized successfully.

8.4 Threats to Validity

In addition to the open questions that impose some threats to the validity of our research, we discuss more issues here that may threaten our research's validity.

8.4.1 Threats to External Validity

In this section, we discuss a threat that hinders our observations, that are collected from OMR and OpenJ9 source codes, from being generalized to all languages that might extend OMR in the future. Here, the specific experiment we are referring to is: collecting information about the OMR and OpenJ9 source codes (using OMRStatistics) to test the hypothesis mentioned in Section 4.4.

- *OpenJ9 Does not Represent All Future Client Codes*: When assessing both variability implementations and deciding which kind of polymorphism is better fit for the project in Chapter 6, we ran OMRStatistics on OMR and OpenJ9 and argued that collecting information of OpenJ9 gives us insight of how OMR is expected to be used by language developers. We wanted to acknowledge here that although we think OpenJ9 sets how language source base and client codes are *likely* or *expected* to use OMR, other languages might actually use OMR differently than we expected. For example, when saying that according to OpenJ9 around 20% of functions are overridden, which is a low number and hence the project is not likely to have a performance degradation when its functions are virtualized, we say that assuming that OpenJ9 is how most projects will

be using OMR. We acknowledge here that some hypothetical projects might override all functions in OMR and raise the percentage of overridden functions to a critical number where our argument is not valid anymore. However, since OpenJ9 is a fairly big project (1,279,809 lines of code according `SLOCCount` [2] on October 2018) that is using OMR, we believe that it shows a typical usage of OMR; using it in a different way might not be the intended way of using it.

8.4.2 Threats to Construct Validity

In this section, we discuss obstacles that didn't allow us to collect all the information we planned to collect at the beginning of the chapter and how those obstacles form threats to the construct validity of our experiment. Here, the experiment we are referring is the same as the one referred to earlier in this chapter: collecting information about the OMR and OpenJ9 source codes (using OMRStatistics) to test the hypothesis mentioned in Section 4.4.

- *Not Considering Template Classes*: While analyzing OMR's source code, we had the option to collect function and class information of template classes however we ignored such classes since our main objective is to help OMR developers with their extensible class implementation design and we confirmed with OMR developers that none of the extensible classes are template classes. Hence, it was not part of our direct objectives to tackle this. So we kept as an open issue in our Github repository [97] for future developers who would want to use our tool for their projects to implement. However, this is considered a threat to the validity of our research since we mention that the tool collects information about *C++* code in general.

- *Ignoring Some Source Codes in OpenJ9*: In order to get all function and class information from the OpenJ9 project, we had to run OMRStatistics on the source code of OpenJ9 for all architectures. To do that, we had to successfully compile the source code of OpenJ9 for all architectures with clang. Hence, cross-compilation was needed to compile

OpenJ9 for *Power* architecture, for instance, using a host (our server) that has an *X86* architecture. This was a tedious task. OMR already has cross-compilation happening since the project already runs *OMR_ Checker* on all architectures. We had to understand OMR's build system and identify all environment variables that are defined when cross-compilation happens and set these variables in our environment when cross compiling the project with clang. This *cloned* build system almost worked for all cross-compilation, except for 19 files that caused compilation errors. Hence, these files were not considered when collecting information in OMRStatistics. We are not entirely sure if that error was due to an erroneous cloning of the OMR build system used in cross-compilation or if the OMR build system was not behaving correctly. OMR does have different machines with all the architectures supported by the OMR library, hence the *OMR_ Checker* test can happen from different machines and no cross-compilation would be needed. This means that if there was an error in the cross-compilation functionality in the OMR build system, it might not be known for developers since the cross-compilation functionality might not be used in their testing. In any case, those 19 files were not checked which forms a threat to the validity of our research.

- *Not having Function Call Information*: Although our intention was to get function call information before proceeding with the project, we later realized that we were not considering all the cases in our algorithm. The way we extract function calls is by visiting every class and processing the body of each function in the classes. If the body contains any member call statement, the statement is processed and function call information is extracted from it. Other statements are ignored. However, other statements (such as assignment statements) might include nested function call statements which we did not check for. Fixing this is one of the issues on our Github repository [98]. However, for now, this is not an accurate way to get function calls and hence we did not consider them when deciding

whether to change OMR's variability implementation mechanism to use dynamic polymorphism or not. Before resorting to that, we tried to use Clang's API to get the call graph information. Clang's `CallGraph` class has a function, called `addToCallGraph`, that populates the `CallGraph` object with all the calls for a specific declaration. We used that to get the function calls however we found out that there was a bug Clang's function that duplicated some of the calls. In other words, some functions that are only called once are reported by Clang's API to be called twice. We later found out that this is a known bug reported on Clang's bug tracking system [30]. Hence, although we find that a low percentage of functions are overridden in the code (see Section 6.3), the functions might be called frequently enough to degrade runtime performance significantly. Our benchmark results show that this, however, is not the case so far.

On the other hand, we also discuss a threat to the construct validity of our other experiment: virtualizing functions in OMR and measuring the impact on runtime performance.

- *Not Virtualizing all Functions in* `CodeGenerator`: Almost all functions in `CodeGenerator` were virtualized successfully. However, there are two functions that were not; these are the ones that were called in the `CodeGenerator` constructor (see Section 7.4.4). The worst case scenario would be that those two functions are called frequently enough to significantly degrade the runtime performance. We cannot prove that this is not the case without function call information.

Chapter 9

Related Works

The objective of our research is to explore the effect of using different kinds of polymorphism in OMR's variability implementation mechanism and help developers make better design decisions related to OMR's variability implementation. Hence, we discuss related literature in the following directions: (1) we review classical and previous variability implementation mechanisms adopted by others (2) we explore tools that support variability, (3) we present how others defined variability evaluation metrics of variability implementations mechanisms in configurable products, and (4) we discuss SPL management and look at other software's variability evolution methods. Given the industrial context of our research, we give practical applications of related work, when applicable.

9.1 Variability Implementation Mechanisms Discussed in the Literature

There are different binding times for software variability such as build-time, load-time, and run-time. There are various variability implementation mechanisms that can be used for the different binding times [8], [91], ranging from simple mechanisms such as using the build system to compile particular modules depending on the feature selection to more sophisticated development paradigms such as *feature-oriented programming* [11]. FOP is a composition-based approach for building software by dividing an SPL into feature modules [8]. Several programming languages were proposed to support feature-

oriented programming, including FeatureC++ [9]. While FOP specifically addresses software variability by introducing the idea of refinements as opposed to inheritance, it has not yet been widely adopted in practice, and to the best of our knowledge, there are no large industrial or open-source systems that use FOP. Other variability implementation mechanisms discussed in the literature include object-orientation and design patterns [8]. Examples of design patterns that can be used in variability implementation mechanisms include the broker pattern [21], [83], the observer pattern [46] and the template pattern [71]. To the best of our knowledge, there has been little work that discusses how large industrial or open-source projects use these variability implementation mechanisms in practice. Specifically, we found no work discussing the impact and challenges of using static polymorphism to implement variability.

For a complete list of possible variability implementation strategies, we refer the reader to the work by Apel et al. [8].

9.2 Variability Implementation Mechanisms in Practice

The most studied variability implementation mechanism in the literature is using the *C* preprocessor's `#ifdef` directives, which was discussed in the introduction. Other mechanisms used in practice include parametrization, plugins or framework programming, and inheritance.

Parametrization is using parameters to change a general program's behavior [44]. In this case, the software contains the implementation for all its variants, and the parameters would control which variant is executed. Examples of parametrization include systems with run-time variability using `if` conditions (e.g., Mozilla Firefox [15]), systems using feature toggles (Google Chrome) [85], and systems with load-time configuration options (e.g., Android apps [64]).

Plugin based systems implement a component framework. A component framework is one that allows the addition of independently built plugins at specified locations called variation points [80]. Such plugins would change

the behavior of the system. Examples of plugin based system include WordPress [76], OpenRAVE [29], and VET [72].

However, in practice, it is common for software to simultaneously use multiple variability implementation mechanisms. For instance, Mozilla Firefox's variability implementation mechanism includes using the black-box component and the broker pattern [45].

9.3 Tools Supporting Variability

In this section, we discuss various techniques and tools that were designed to help developers reason about the variability in their system.

9.3.1 Variability Aware Analysis

Understanding the code of a software system and properly testing it is essential for software quality and maintenance. This becomes more complicated in the context of an SPL with n optional features since instead of having one product to analyze and test, there are 2^n products. Analyzing all these products in separation is not feasible. Thus, many analysis strategies have been proposed in the literature ranging from sampling configurations to analyzing all configurations in a more efficient way. Thum et al. [95]'s recent survey summarizes all these analysis techniques. Since sampling strategies are not complete, much research effort has focused on *variability-aware analyses* that simultaneously analyze all products. To avoid a brute-force mechanism, they analyze shared code only once and analyze multiple variants of the code only when necessary. Again, creating variability-aware analyses has been most popular in the context of *C* code with `#ifdef` directives. TypeChef [53] and SuperC [41] are two such efforts. While TypeChef has been used for analyzing many preprocessor-based systems, including some in Java, it does not currently support *C++*. Hu et al. [48] provided a related effort, based on symbolic execution, to analyze the conditional compilation of *C++* header files. However, symbolic execution is typically expensive and does not always scale to large systems [6]. Similar variability-aware analysis efforts have been proposed for systems using load-

time or run-time variability [76], [95]. Other tools include `FeatureIDE` [96], a variability aware IDE that analyzes projects and maps their code artifacts to features, and `RequiLine` [70], a tool that supports requirements engineering in SPLs. While previous efforts can guide the design of a variability-aware analysis tool for *C++*, there does not currently exist a robust tool for this purpose, especially when additional variability is created through the notion of static polymorphism. Part of the future work of this research is to adjust the source code of Clang to add support for variability aware compilation of the Eclipse OMR project.

9.3.2 Other Tools Supporting Variability

Other than analysis tools, there are various research and industrial tools developed to support reasoning about software variability.

`GEARS` is a code analysis tool developed by BigLever Software that focuses on software mass customization in software product lines [58]. Mass customization was also tackled by Ronny et al. [56]. In their research, Ronny et al. use the PuLSE approach [14] to convert a product into a reusable core component of a product line. It is worthy to note that Eclipse OMR also started as a single product which was then transformed to the core of a product line, as described in our position paper [69].

Various other techniques are used to support variability. Eisenbarth et al. [36] use concept analysis, alongside with static and dynamic analysis, to correlate source code blocks to sets of features. Concept analysis is also used by Krone et al. [57] to extract configuration dependencies for projects whose variability is implemented with C preprocessors (CPP) [49], and by Loesh et al.[65] to visualize product features and configurations. Another tool that supports projects that use the *C* preprocessor in their variability implementations is created by Baxter et al. [12]. Their tool uses DMS, a transformation system used to gradually alter and orient a software's design for more efficient maintenance [13] by neatly removing preprocessor configurations of unsupported features.

Different from above, OMRStatistics works by statically analyzing the

Eclipse OMR code, to support variability design decisions by collecting information about the project's variability points. Developers can then use this information to decide about any needed changes to their variability implementation.

9.4 Variability Metrics

As mentioned before, using the preprocessor directives as a variability implementation mechanism is very popular. However, it is not always the best fit for a project. Liebeg et al. [63] analyze more than forty projects that use the *C* preprocessor as part of their variability implementation mechanism according to metrics introduced by the authors, suggesting alternative variability implementations. The main two metrics they used are program comprehension and feasibility of refactoring in a project. Hunsen et al. [49] study twenty-seven projects that use the *C* preprocessor in their variability implementation in order to study the similarity between such implementations in open-source and industrial projects. In their paper, the authors define the similarity between implementations by a set of variability metrics. Metrics used to define similarity include scattering, tangling, and nesting of `#ifdef` blocks.

Such metrics contribute to answering the open question imposed in Section 8.2 about how to decide on the best variability implementation mechanism for a project.

9.5 Software Product Line Management

Eclipse OMR evolved over history. It started from a single product, the *Testarossa* compiler, and slowly evolve to reach the SPL it is today. A Big part of our research deals with changing OMR's variability implementation mechanism and understanding the implication of that change. We now look at how different SPLs have been launched from single products and how their variability has been managed.

Metzger et al.[73] review 600 papers published in the last seven years that discuss SPLs and summarize the trends and anticipated challenges in software

engineering. This gives us insight into the challenges that might be faced when managing SPLs. According to Metzger et al., variability can be applied to form an SPL in three ways [73]: (1) proactively, when the variability design and implementation is planned before the implementation of a software, (2) reactively, when the main artifact is developed and then that artifact is reused, based on customer needs, to form an SPL, and (3) reengineering-driven, when a single product is developed and then it is *migrated* to an SPL. Creating a business case for using a proactive SPL approach is not easy, hence software systems often stay away from a proactive SPL approach [73].

A popular approach used in the software industry is the reactive approach. Alves et al. [5] describe a reactive extractive approach to manage and evolve product lines. The authors derive their suggested approach from Aspect Oriented Programming laws [22]. Buhrdorf et al. [20] describe Salion's (a software company) experience with reactive transition approach. The authors prove the approach's efficiency by studying its usage to launch seven new products Ghanam et a. [43] suggest a reactive transition framework to implement and manage variability in agile software. Agile is getting a rising popularity in the software industry and its integration with SPL engineering is getting more popular in literature. Hansenn et al. [47] investigate the integration between software product line approaches and agile software development through a case study of a Norwegian software company. In addition, The Journal of Systems and Software had a full issue discussing SPL engineering integration with agile methods [23].

More SPLs can be found on the Product Line Hall of Fame on the Systems and Software Product Line Conference website [82] which also contains papers related to the formation and evolution of those SPLs.

9.5.1 Recap

We reviewed the literature about variability implementation mechanisms and looked at examples of how variability is implemented practically. We also reviewed tools that are created to support variability in different ways, including variability aware analysis, concept analysis, and mass customization.

We then discussed literature about variability evaluation and looked at what metrics others considered when evaluating variability implementation mechanisms. Finally, we looked at practical examples of how variability was applied to products to launch software products lines and how variability is managed in SPLs.

Chapter 10

Conclusion

In this thesis, we discussed how variability can be implemented using static polymorphism. Specifically, we presented a practical case study of Eclipse OMR's variability implementation using a combination of extensible classes and include path variation. We discussed how OMR achieves the intended variability and supports new language extensions. Given the complicated and unique nature of how variability is implemented in OMR, OMR developers faced some challenges while implementing variability and had to make some design decisions to overpass the faced challenges. One decision was to use static polymorphism instead of dynamic polymorphism in order to protect the runtime performance from being impacted. However, after progressing in the development of the project as a product line, OMR developers have been realizing some problems with the static polymorphism implementation: the obscurity of extension points and the complexity added by the resulting code conventions. Hence, OMR's variability implementation design is being revisited.

The some OMR developers speculated that virtualizing the functions that act as extension points of the project will not have a significant impact on the runtime performance but will have a positive impact on newcomers understanding of the code. To help developers decide if it makes sense to switch to dynamic polymorphism, we created a tool, OMRStatistics, that statically analyzes the code and extracts the information needed to help OMR developers assess how changing OMR's variability implementation mechanism to

dynamic polymorphism will impact the overall performance and complexity of the project.

Based on the information we collected, switching the extension points will theoretically not have a significant impact on the runtime performance of the tool. However, the actual impact can only be measured by progressively changing extension points to dynamic polymorphism and observing the impact on performance. We hence went ahead and virtualized over 180 functions in the source code, solving multiple obstacles and challenges that we faced while virtualizing. After that, we benchmarked the project with our changes which revealed that our changes had no significant degradation in the runtime performance of the project.

Following the initiative we started in this research, future work includes continuing the virtualization process of the functions in Eclipse OMR. In light of the new data obtained by benchmarking the project with the altered variability implementation mechanism fully in place, the implementation mechanism will either be accepted or rejected by the stakeholders. If accepted, future work would also include surveying current and new OMR developers to understand the challenges and difficulties they are facing and to understand the effect of changing the variability implementation mechanism has on these challenges; whether the new variability implementation mechanism did practically facilitate working with the source code or not. As described earlier, this would happen through surveys aimed at current and new developers contributing to the OMR project, tracking the change in the number of questions asked on forums and emails, and tracking the change in the number of PRs and contributions to the project.

While Eclipse OMR is our inspiration for this work, our contributions are also intended to the rest of the community to understand the practical implications of using static polymorphism in an industrial big project and the difficulties faced when trying shift from static to dynamic polymorphism in the implementation of such a project. In addition, OMRStatistics can theoretically work on any *C++* project to collect class and method information. Hence, it has the potential to benefit a wide range of developers, helping them

reason about variability in $C{+}{+}$ code.

References

[1] *A C language family frontend for LLVM*, http://clang.llvm.org/, 2007. (visited on 09/05/2018). 14, 29

[2] D. A. Wheeler, *Sloccount*, https://www.dwheeler.com/sloccount/, 2004. 16, 82

[3] (). Agenda for june 6, 2018 OMR Compiler Architecture Meeting, [Online]. Available: https://github.com/eclipse/omr/issues/2571 (visited on 09/23/2018). 57

[4] N. et al., *A framework for software product line practice, version 5.0*, https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=495357, Online; accessed 14 August, 2018, 2012. 1

[5] V. Alves, P. Matos, L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho, "Extracting and evolving code in product lines with aspect-oriented programming," in *Transactions on Aspect-Oriented Software Development IV*, A. Rashid and M. Aksit, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 117–142, ISBN: 978-3-540-77042-8. 90

[6] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 367–381, ISBN: 978-3-540-78800-3. 87

[7] M. Antkiewicz and K. Czarnecki, "Featureplugin: Feature modeling plug-in for eclipse," in *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange*, ser. eclipse '04, Vancouver, British Columbia, Canada: ACM, 2004, pp. 67–72. 2

[8] S. Apel, D. Batory, C. Kstner, and G. Saake, *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013, ISBN: 3642375200, 9783642375200. 3, 11, 12, 85, 86

[9] S. Apel, T. Leich, M. Rosenmuller, and G. Saake, "FeatureC++ on the symbiosis of feature-oriented and aspect-oriented programming," in *Proc. of the 4th International Conference on Generative Programming*, ser. GPCE '05, Springer, vol. 3676, 2005, pp. 125–140. 86

[10] D. F. Bacon and P. F. Sweeney, "Fast static analysis of C++ virtual function calls," in *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '96, San Jose, California, USA: ACM, 1996, pp. 324–341, ISBN: 0-89791-788-X. 13

[11] D. Batory, "Feature-oriented programming and the AHEAD Tool Suite," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04, Washington, DC, USA: IEEE Computer Society, 2004, pp. 702–703, ISBN: 0-7695-2163-0. 2, 85

[12] I. Baxter and M. Mehlich, "Preprocessor conditional removal by simple partial evaluation," in *Proceedings Eighth Working Conference on Reverse Engineering*, 2001, pp. 281–290. 88

[13] I. Baxter and C. W. Pidgeon, "Software change through design maintenance," in *1997 Proceedings International Conference on Software Maintenance*, Oct. 1997, pp. 250–259. 88

[14] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud, *Pulse: A methodology to develop software product lines*, Jan. 1999. 88

[15] F. Behrang, M. B. Cohen, and A. Orso, "Users beware: preference inconsistencies ahead," in *Proc. of the 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, Bergamo, Italy: ACM, 2015, pp. 295–306, ISBN: 978-1-4503-3675-8. 86

[16] E. Bendersky, *The curiously recurring template pattern in C++*, `http://eli.thegreenplace.net/2011/05/17/the-curiously-recurring-template-pattern-in-c/`, 2011. 12, 13

[17] T. Berger, D. Lettner, J. Rubin, P. Grunbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a feature?: A qualitative study of features in industrial software product lines," in *Proceedings of the 19th International Conference on Software Product Line*, ser. SPLC '15, Nashville, Tennessee: ACM, 2015, pp. 16–25, ISBN: 978-1-4503-3613-0. 10

[18] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wkasowski, "A survey of variability modeling in industrial practice," in *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, ser. VaMoS '13, Pisa, Italy: ACM, 2013, 7:1–7:8, ISBN: 978-1-4503-1541-8. 2

[19] J. Bosch, "Software variability management," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04, Washington, DC, USA, 2004, pp. 720–721, ISBN: 0-7695-2163-0. 9

[20] R. Buhrdorf, D. Churchett, and C. W. Krueger, "Salion's experience with a reactive software product line approach," in *Software Product-Family Engineering*, F. J. van der Linden, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 317–322. 90

[21] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture*. 1996. 86

[22] L. Cole and P. Borba, "Deriving refactorings for AspectJ," in *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, ser. AOSD '05, Chicago, Illinois: ACM, 2005, pp. 123–134, ISBN: 1-59593-042-6. 90

[23] K. Cooper and X. Franch, "Editorial," *Journal of Systems and Software*, vol. 81, no. 6, pp. 841–842, 2008, Agile Product Line Engineering, ISSN: 0164-1212. 90

[24] J. O. Coplien, *Multi-paradigm Design for C++*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999, ISBN: 0-201-82467-1. 11

[25] ——, "Multi-Paradigm design," PhD thesis, Vrije University Brussels, Jul. 2000. 3

[26] H. B. Dave and P. H. Dave, *Compilers: Principles and Practice*. Pearson India, 2012. 10

[27] (). Debugging OpenJ9 in Docker with GDB, [Online]. Available: `https://blog.openj9.org/2018/06/05/debugging-openj9-in-docker-with-gdb/` (visited on 10/03/2018). 62

[28] D. Dhungana, R. Rabiser, P. Grunbacher, and T. Neumayer, "Integrated tool support for software product line engineering," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07, Atlanta, Georgia, USA: ACM, 2007, pp. 533–534, ISBN: 978-1-59593-882-4. 2

[29] R. Diankov and J. Kuffner, "Openrave: A planning architecture for autonomous robotics," Apr. 2011. 87

[30] (). Do not duplicate call graph nodes for function that has definition and forward declaration, [Online]. Available: `https://reviews.llvm.org/D29643` (visited on 10/03/2018). 84

[31] E. Dolstra, G. Florijn, and E. Visser, "Timeline variability: the variability of binding time of variation points," Department of Information and Computing Sciences, Utrecht University, Tech. Rep. UU-CS-2003-052, 2003. 11

[32] K. Driesen and U. Holzle, "The direct cost of virtual function calls in C++," *SIGPLAN Not.*, vol. 31, no. 10, pp. 306–323, Oct. 1996, ISSN: 0362-1340. 13

[33] *Eclipse OMR*, https://github.com/eclipse/omr, 2018. 4, 14, 16

[34] (). Eclipse OMR: Building language runtimes for the cloud, [Online]. Available: https://www.eclipse.org/community/eclipse_newsletter/2016/october/article5.php (visited on 04/03/2018). 15

[35] (). Eclipse OpenJ9, [Online]. Available: https://github.com/eclipse/openj9/ (visited on 09/20/2018). 33

[36] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Trans. Softw. Eng.*, vol. 29, no. 3, pp. 210–224, Mar. 2003, ISSN: 0098-5589. 88

[37] (). Fastpath mode, [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SS7J8H/com.ibm.odm.dserver.rules.designer.run/optimizing_topics/con_opt_execmodes_fastpath.html (visited on 10/03/2018). 69

[38] G. Ferreira, M. Malik, C. Kastner, J. Pfeffer, and S. Apel, "Do #ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel," in *Proceedings of the 20th International Systems and Software Product Line Conference*, ser. SPLC '16, Beijing, China: ACM, 2016, pp. 65–73, ISBN: 978-1-4503-4050-2. 11

[39] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven, "Using architecture models for runtime adaptability," *IEEE Software*, vol. 23, no. 2, pp. 62–70, Mar. 2006. 3

[40] M. Gaudet and M. Stoodley, "Rebuilding an airliner in flight: A retrospective on refactoring IBM Testarossa Production Compiler for Eclipse OMR," in *Proc. of the 8th International Workshop on Virtual Machines and Intermediate Languages*, ser. VMIL 2016, Amsterdam, Netherlands: ACM, 2016, pp. 24–27, ISBN: 978-1-4503-4645-0. 15

[41] P. Gazzillo and R. Grimm, "SuperC : Parsing all of C by taming the preprocessor," in *Proc. of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, Beijing, China: ACM, 2012, pp. 323–334, ISBN: 978-1-4503-1205-9. 87

[42] (). GDB : The GNU Project Debugger, [Online]. Available: https://www.gnu.org/software/gdb/ (visited on 09/23/2018). 62

[43] Y. Ghanam, D. Andreychuk, and F. Maurer, "Reactive variability management in agile software development," in *2010 Agile Conference*, Aug. 2010, pp. 27–34. 90

[44] H. Gomaa and D. L. Webber, "Modeling adaptive and evolvable software product lines using the variation point model," in *37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the*, Jan. 2004. 86

[45] J. van Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in *Proceedings Working IEEE/IFIP Conference on Software Architecture*, 2001, pp. 45–54. 3, 87

[46] J. Hannemann and G. Kiczales, "Design pattern implementation in Java and aspectJ," in *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '02, Seattle, Washington, USA: ACM, 2002, pp. 161–173, ISBN: 1-58113-471-1. 86

[47] G. K. Hanssen and T. E. Fægri, "Process fusion: An industrial case study on agile software product line engineering," *Journal of Systems and Software*, vol. 81, no. 6, pp. 843–854, 2008, Agile Product Line Engineering, ISSN: 0164-1212. 90

[48] Y. Hu, E. Merlo, M. Dagenais, and B. Lague, "C/C++ conditional compilation analysis using symbolic execution," in *Proceedings 2000 International Conference on Software Maintenance*, 2000, pp. 196–206. 87

[49] C. Hunsen, B. Zhang, J. Siegmund, C. Kastner, O. Lessenich, M. Becker, and S. Apel, "Preprocessor-based variability in open-source and industrial software systems: An empirical study," *Empirical Software Engineering*, vol. 21, no. 2, pp. 449–482, Apr. 2016, ISSN: 1573-7616. 88, 89

[50] (). Improve operational performance using automated business rules to make better decisions, [Online]. Available: `https://www.ibm.com/cloud/automation-software/business-rules` (visited on 10/03/2018). 69

[51] (). Java EE6: Daytrader3 sample, [Online]. Available: `https://github.com/WASdev/sample.daytrader3#java-ee6-daytrader3-sample` (visited on 10/03/2018). 68

[52] C. Kastner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08, Leipzig, Germany: ACM, 2008, pp. 311–320, ISBN: 978-1-60558-079-1. 3

[53] C. Kastner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," *SIGPLAN Not.*, vol. 46, no. 10, pp. 805–824, Oct. 2011, ISSN: 0362-1340. 3, 87

[54] R. Kauppinen, "Testing framework-based software product lines," Master's thesis, University of Helsinki, Apr. 2003. 3

[55] A. Kenner, C. Kastner, S. Haase, and T. Leich, "TypeChef : Toward type checking #ifdef variability in C," in *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*, ser. FOSD '10, Eindhoven, The Netherlands: ACM, 2010, pp. 25–32, ISBN: 978-1-4503-0208-1. 2

[56] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, “A case study in refactoring a legacy component for reuse in a product line,” in *21st IEEE International Conference on Software Maintenance (ICSM’05)*, Sep. 2005, pp. 369–378. 88

[57] M. Krone and G. Snelting, “On the inference of configuration structures from source code,” in *Proceedings of the 16th International Conference on Software Engineering*, ser. ICSE ’94, Sorrento, Italy: IEEE Computer Society Press, 1994, pp. 49–57, ISBN: 0-8186-5855-X. 88

[58] C. Krueger, “Easing the transition to software mass customization,” in *Software Product-Family Engineering*, F. van der Linden, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 282–293. 88

[59] C. Lattner. (2011). Llvm, [Online]. Available: `http://www.aosabook.org/en/llvm.html` (visited on 09/05/2018). 14

[60] C. Lattner and V. Adve, “The LLVM compiler framework and infrastructure tutorial,” in *Languages and Compilers for High Performance Computing*, R. Eigenmann, Z. Li, and S. P. Midkiff, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 15–16. 14

[61] H. K. N. Leung and L. White, “A study of integration testing and software regression at the integration level,” in *Proceedings. Conference on Software Maintenance 1990*, Nov. 1990, pp. 290–301. 2

[62] (). Liberty overview, [Online]. Available: `https://www.ibm.com/support/knowledgecenter/SSEQTP_liberty/com.ibm.websphere.wlp.doc/ae/cwlp_about.html` (visited on 10/03/2018). 68

[63] J. Liebig, S. Apel, C. Lengauer, C. Kastner, and M. Schulze, “An analysis of the variability in forty preprocessor-based software product lines,” in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE ’10, Cape Town, South Africa: ACM, 2010, pp. 105–114. 2, 89

[64] M. Lillack, C. Kastner, and E. Bodden, “Tracking load-time configuration options,” in *Proc. of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE ’14, Vasteras, Sweden: ACM, 2014, pp. 445–456, ISBN: 978-1-4503-3013-8. 86

[65] F. Loesch and E. Ploedereder, “Optimization of variability in software product lines,” in *11th International Software Product Line Conference (SPLC 2007)*, Sep. 2007, pp. 151–162. DOI: `10.1109/SPLINE.2007.31`. 88

[66] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wkasowski, “Evolution of the linux kernel variability model,” in *Proc. of the 14th International Conference on Software Product Lines: Going Beyond*, ser. SPLC’10, Jeju Island, South Korea, 2010, pp. 136–150, ISBN: 3-642-15578-2, 978-3-642-15578-9. 3

[67] D. Maier, M. Gaudet, and L. Lorimer, *Extensible classes*, `https://github.com/eclipse/omr/blob/2d869042b8ead6efebdde293f182769d0730f112/doc/compiler/extensible_classes/Extensible_Classes.md`, 2016. (visited on 09/07/2018). 17, 29

[68] (). Manual virtualization helper, [Online]. Available: `https://github.com/samasri/omr/tree/master/tools/compiler/OMRStatistics/sourceCodeProcessors/manualVirtualizationHelper` (visited on 09/23/2018). 61

[69] S. A. Masri, N. U. Bhuiyan, S. Nadi, and M. Gaudet, "Software variability through C++ static polymorphism: A case study of challenges and open problems in Eclipse OMR," in *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '17, Markham, Ontario, Canada: IBM Corp., 2017, pp. 285–291. 5, 17, 33, 88

[70] T. von der Maßen and H. Lichter, "Requiline: A requirements engineering tool for software product lines," in *Software Product-Family Engineering*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 168–180, ISBN: 978-3-540-24667-1. 88

[71] J. E. McDonough, *Template Method Design Pattern*. Berkeley, CA: Apress, 2017, pp. 247–254. 86

[72] M. McGavin, T. Wright, and S. Marshall, "Visualisations of execution traces (VET): An interactive plugin-based visualisation tool," in *Proceedings of the 7th Australasian User Interface Conference - Volume 50*, ser. AUIC '06, Hobart, Australia: Australian Computer Society, Inc., 2006, pp. 153–160, ISBN: 1-920682-32-5. 87

[73] A. Metzger and K. Pohl, "Software product line engineering and variability management: Achievements and challenges," in *Proceedings of the on Future of Software Engineering*, ser. FOSE 2014, Hyderabad, India: ACM, 2014, pp. 70–84, ISBN: 978-1-4503-2865-4. 89, 90

[74] S. Nadi, C. Dietrich, R. Tartler, R. C. Holt, and D. Lohmann, "Linux variability anomalies: What causes them and how do they get fixed?" In *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 111–120, ISBN: 978-1-4673-2936-1. 3

[75] S. Nadi and R. Holt, "Mining kbuild to detect variability anomalies in linux," in *Proc. of the 16th European Conference on Software Maintenance and Reengineering*, ser. CSMR '12, 2012, pp. 107–116. 2

[76] H. V. Nguyen, C. Kastner, and T. N. Nguyen, "Exploring variability-aware execution for testing plugin-based web applications," in *Proc. of the 36th International Conference on Software Engineering*, ser. ICSE '14, Hyderabad, India: ACM, 2014, pp. 907–918, ISBN: 978-1-4503-2756-5. 87, 88

[77] (). OMR Compiler Architecture Meetings, [Online]. Available: `https://github.com/eclipse/omr/issues/2316` (visited on 09/23/2018). 57

[78] (). Omrstatistics, [Online]. Available: `https://github.com/samasri/omr/tree/master/tools/compiler/OMRStatistics` (visited on 09/20/2018). 32

[79] S. Ouali, N. Kraiem, and H. Ben Ghezala, "Framework for evolving software product line," vol. 2, May 2011. 9

[80] K. Pohl, G. Bockle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin, Heidelberg: Springer-Verlag, 2005, ISBN: 3540243720. 86

[81] H. Post and C. Sinz, "Configuration lifting: Verification meets software configuration," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08, Washington, DC, USA: IEEE Computer Society, 2008, pp. 347–350, ISBN: 978-1-4244-2187-9. 3

[82] (). Product line hall of fame, [Online]. Available: `http://splc.net/hall-of-fame/` (visited on 10/02/2018). 90

[83] F. Puhlmann, A. Schnieders, J. Weiland, and M. Weske, *Variability Mechanisms for Process Models*. 2005. 86

[84] (). Pure::variants eclipse plug-in user's guide, [Online]. Available: `http://web.pure-systems.com/fileadmin/downloads/pv_userguide.pdf` (visited on 04/03/2018). 2

[85] M. T. Rahman, L.-P. Querel, P. C. Rigby, and B. Adams, "Feature toggles: Practitioner practices and a case study," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, May 2016, pp. 201–211. 86

[86] M. Rosenmuller and U. Assmann, "Towards flexible feature composition : Static and dynamic binding in software product lines," 2011. 11

[87] M. Rosenmuller, N. Siegmund, S. Apel, and G. Saake, "Flexible feature binding in software product lines," *Automated Software Engineering*, vol. 18, no. 2, pp. 163–197, Jun. 2011. 11

[88] S. S. Some and T. C. Lethbridge, "Parsing minimization when extracting information from code in the presence of conditional compilation," in *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, Jun. 1998, pp. 118–125. 3

[89] (). Static polymorphic construction: A flaw in extensible classes, [Online]. Available: `https://github.com/eclipse/omr/issues/1797` (visited on 09/23/2018). 67

[90] M. Stoodley, "Under the hood of the Testarossa JIT compiler," Retrieved from: https://www.slideshare.net/MarkStoodley/under-the-hood-of-the-testarossa-jit-compiler, 2012, (visited on 09/03/2018). 15

[91] M. Svahnberg, J. van Gurp, and J. Bosch, "A taxonomy of variability realization techniques: research articles," *Softw. Pract. Exper.*, vol. 35, no. 8, pp. 705–754, Jul. 2005, ISSN: 0038-0644. 85

[92] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero, "Configuration coverage in the analysis of large-scale system software," in *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, ser. PLOS '11, Cascais, Portugal: ACM, 2011, 2:1–2:5, ISBN: 978-1-4503-0979-0. 3

[93] L. A. Team. (2003). The LLVM compiler infrastructure project, [Online]. Available: `http://llvm.org/` (visited on 09/05/2018). 14

[94] (). TeamViewer – remote support, remote access, service desk, online collaboration and meetings, [Online]. Available: `https://www.teamviewer.com` (visited on 09/18/2018). 2

[95] T. Thum, S. Apel, C. Kastner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Computing Surveys (CSUR)*, vol. 47, no. 1, p. 6, 2014. 3, 87, 88

[96] T. Thum, C. Kastner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "FeatureIDE : An extensible framework for feature-oriented software development," *Science of Computer Programming*, vol. 79, pp. 70–85, 2014, Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010). 88

[97] (). Tool is not guaranteed to be reading all the classes in the source code, [Online]. Available: `https://github.com/samasri/omr/issues/18` (visited on 09/25/2018). 82

[98] (). Use callgraph to track function calls, [Online]. Available: `https://github.com/samasri/omr/issues/48` (visited on 09/25/2018). 83

[99] (). What are segmentation faults (segfaults), and how can i identify what's causing them? [Online]. Available: `https://kb.iu.edu/d/aqsj` (visited on 09/28/2018). 62

[100] (). What is a "pure virtual" member function? [Online]. Available: `https://isocpp.org/wiki/faq/abcs#pure-virtual-fns` (visited on 09/23/2018). 64

[101] (). When my base class's constructor calls a virtual function on its this object, why doesn't my derived class's override of that virtual function get invoked? [Online]. Available: `https://isocpp.org/wiki/faq/strange-inheritance#calling-virtuals-from-ctors` (visited on 09/23/2018). 66

[102] (). WIP : Virtualized ELFRelocationResolver, [Online]. Available: `https://github.com/eclipse/omr/pull/2584` (visited on 09/23/2018). 59

[103] D. Woods, G. Brail, and D. Jacobson, *APIs: A Strategy Guide*. O'Reilly Media, Inc., 2011. 10

[104] M. Yang, C. Iyer, and C. Wetherell, "Method and apparatus for performing conditional compilation," US11495850, 2006. 3

[105] L. Zanivan. (). New open source JVM optimized for cloud and microservices, [Online]. Available: `https://medium.com/criciumadev/new-open-source-jvm-optimized-for-cloud-and-microservices-c75a41aa987d` (visited on 09/27/2018). 68