Using Static Analysis to Support Variability Implementation Decisions in C++

Samer AL Masri, Sarah Nadi
University of Alberta
AB, Canada
{almasrinadi}@ualberta.ca

Matthew Gaudet*
Mozilla
Ottawa, ON, Canada
mgaudet@mozilla.com

Xiaoli Liang, Robert W. Young
IBM Canada
Markham, ON, Canada
{xsliang,rwyoung}@ca.ibm.com

ABSTRACT

Eclipse OMR is an open-source C++ framework for building robust language runtimes. The OMR toolkit includes a dynamic Just-In-Time (JIT) compiler, a garbage collector, a platform abstraction library, and a set of developer tooling capabilities. 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. That is, all implementation classes that depend on the selected language and architecture are decided at compile time. However, OMR developers now realize that the current variability design decision, specifically the static polymorphism implementation, has its drawbacks. They are considering using dynamic polymorphism instead of static polymorphism. Before making such a fundamental design change, however, it is crucial to collect function information and overload/override statistics about the current variability in the code base.

In this paper, we present OMRStatistics, a static analysis tool that we built for OMR developers to help them collect this information. Specifically, OMRStatistics (1) visualizes the class hierarchy from OMR's current static polymorphic implementation, (2) visualizes the function overloads and overrides with their respective locations in the source code, (3) collects important information about the classes and functions, and (4) stores all the collected information in a database for further analysis. Our tool OMRStatistics allows OMR developers to make better design decisions on which variability extension points should be switched from static polymorphism to dynamic polymorphism.

KEYWORDS

clang plugin, static analysis, software variability analysis, C++, static polymorphism, dynamic polymorphism, build path variability

ACM Reference Format:
Samer AL Masri, Sarah Nadi, Matthew Gaudet, and Xiaoli Liang, Robert W. Young. 2018. Using Static Analysis to Support Variability Implementation Decisions in C++. In *22nd International Systems and Software Product Line Conference - Volume A (SPLC '18), September 10–14, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3233027.3233043

*This work was done while the author was employed by IBM Canada.

SPLC '18, September 10–14, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6464-5/18/09…$15.00
https://doi.org/10.1145/3233027.3233043

1 INTRODUCTION

Background. Software Product Lines (SPLs) promote systematic software reuse by providing a way of configuring different, and yet similar, products from the same set of artifacts [19]. SPLs offer a set of features, where each *feature* corresponds to some behavior or functionality implemented in the system [7]. A variant of the SPL is configured by selecting the desired feature combination. For example, the Linux kernel is considered a SPL [41] that can be configured to work on different hardware platforms and supports different features: a Linux kernel configured to include USB support on ARM architecture is different than one that is configured with HDMI support on X86 architecture.

To achieve the software variability that appears in SPLs, various variability implementation mechanisms have been discussed in the literature [7], such as using conditional statements, parameters, design patterns, frameworks, or components. One of the simplest and most commonly used variability implementation mechanism is the *C* preprocessor's `#ifdef` directives to support *conditional compilation.* This has led to a vast amount of research and tools for supporting variability in preprocessor-based code (e.g, [27, 28, 37, 38, 42–44, 49]). However, in practice, developers may be using other variability implementation mechanisms that may not perfectly align with the mechanisms commonly discussed in the literature, and which may not have good tool support.

Eclipse OMR. In this paper, we discuss Eclipse OMR [2], which was open sourced by IBM in 2016, as an industrial case study of a configurable system that uses an uncommon combination of variability implementation mechanisms. OMR is an open-source framework for building language runtimes. It provides the building blocks for just-in-time compilers, garbage collectors and more, each of which can be customized to a targeted language [20]. The cross-platform components also support multiple operating systems and target architectures: X86 (AMD64 and i386), Power, Z, and ARM. 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/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.

The OMR framework's extension model is based on build system variability and static polymorphism. The OMR compiler component, which is our main focus in this paper, 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.

The Problem. Variability implementation decisions may often need to be revisited during the lifetime of a given software product line [17]. The choice of using static polymorphism, instead of the more typical dynamic polymorphism, was driven by performance concerns, and was decided at the beginning of the project. Given that OMR variants are language run-time environments, they need to be optimized as much as possible. The assumption was that language extensions will need to override many functions from the core functionality and hence dynamic polymorphism will have a negative impact on the run-time performance of the products. In addition, most classes in downstream projects will only have one real implementation of these methods, so adding the `virtual` keyword and the cost of dynamic dispatch is not necessary. After developing OMR, developers now realize that one major downside to using the current variability implementation mechanism is the difficulty of onboarding potential consumers of the framework, as it is challenging to reason about and extend OMR without detailed understanding of all the pieces. Static polymorphism also forces OMR developers and OMR consumers to use some conventions that are uncommon for C++ to ensure it works as expected. Due to these drawbacks, OMR developers are currently considering switching to dynamic polymorphism to enable the explicit definition of extension points using the `virtual` keyword. The hypothesis is that, in practice, changing the extension points to dynamic polymorphism would not add a significant cost to the runtime performance of the product. In order to investigate this hypothesis, OMR developers need to have a better understanding of the current hierarchies in the code and how/if downstream projects (i.e., additional programming languages) use the various functions in the code.

Contributions of this Paper. In this paper, we report on how we supported OMR developers with their variability implementation decisions. Specifically, we created a Clang (a C family front-end for the LLVM compiler [3]) [1] plug-in, called OMRStatistics, that (1) visualizes the class hierarchy from OMR's current static polymorphic implementation, (2) visualizes the function overloads and overrides with their respective locations in the source code, (3) collects important information about the classes and functions (such as lists of functions in each class and callsite information), and (4) stores all the collected information in a database for further analysis. Running OMRStatistics on the OMR code showed that client developers have to search in a wide range of classes for the functions (extension points) they need to extend in order to connect their language with OMR. In addition, we found out that only 3.4% of all the functions in these classes are expected to be extended. Powered with this information, OMR developers now believe that changing to dynamic polymorphism will (1) make it clearer and more intuitive for developers to find the right extension points to use when extending OMR with their language, and at the same time (2) will have a minimal impact on the run-time performance of OMR.

Paper Organization. The rest of this paper is organized as follows. Section 2 presents some background information about the Eclipse OMR project, which is the target of our OMRStatistics tool. In Section 3, we describe the problem of static vs. dynamic polymorphism in more details to motivate the need for OMRStatistics. Section 4 describes the details of OMRStatistics. Section 5 describes the statistics we obtained by applying OMRStatistics to OMR, as well as how OMR developers used these results. Section 6 presents related work and Section 7 concludes this paper.

2 BACKGROUND: ECLIPSE OMR

In this section, we describe the open-source project Eclipse OMR and its variability implementation in more detail.

2.1 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). The project was open sourced in March 2016, and intends to unlock the inner workings of the JVM, without imposing Java semantics, to create a common platform for building language runtimes [2].

Eclipse OMR is mostly written in C++. It includes a dynamic Just-In-Time (JIT) compiler, a garbage collector, a platform abstraction library, and a set of developer tooling capabilities. These OMR components can easily be integrated into an existing runtime by language developers, providing support on different hardware architectures: *X86(AMD64 and I386)*, *Power*, and *Z*. In this paper, we will focus our discussions and experiments on the JIT compiler component of the Eclipse OMR project. Considering that Eclipse OMR works with different languages and architectures, it has been designed with software variability in mind. It can be thought of as an SPL, where each combination of language and architecture is a variant or product.

2.2 Variability Implementation

OMR is designed to support the five architectures mentioned above and any number of programming languages. Ignoring some low-level details that we will not discuss in this paper, OMR's variability implementation can be reduced to three main ideas: a nested directory structure corresponding to different layers in the implementation, static polymorphism, and include path variability.

2.2.1 Directory Structure. Since supporting architectures implies that classes must have different implementations, each class is declared in different directories representing the different architectures supported in the project. Each declaration includes an implementation for a specific architecture. Figure 1 shows an example of a class, *TreeEvauator*, which is implemented differently depending on the host architecture. *TreeEvaluator* is declared in the header file *OMRTreeEvaluator.hpp*, which is found in the directory of every architecture (under the subdirectory: *codegen*). Notice that the different header files all share the same name; this is a critical fact that

```
Compiler
 ├── codegen
 │    └── OMRTreeEvaluator.hpp
 ├── x86
 │    ├── codegen
 │    │    └── OMRTreeEvaluator.hpp
 │    ├── i386
 │    │    └── codegen
 │    │         └── OMRTreeEvaluator.hpp
 │    └── AMD64
 │         └── codegen
 │              └── OMRTreeEvaluator.hpp
 ├── Power
 │    └── codegen
 │         └── OMRTreeEvaluator.hpp
 ├── Z
 │    └── codegen
 │         └── OMRTreeEvaluator.hpp
 └── ARM
      └── codegen
           └── OMRTreeEvaluator.hpp
```

Figure 1: Example of OMR Directory Structure

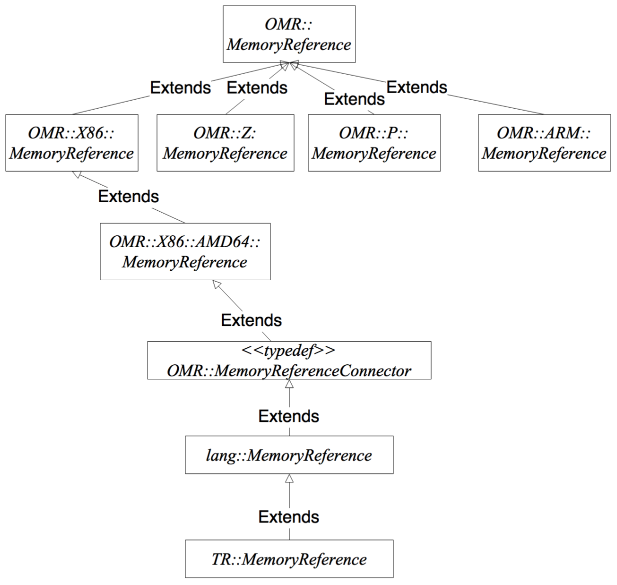


Figure 2: Example of a Class Hierarchy in OMR

allows the include path variability to function correctly. However, each declaration in a file uses a namespace that corresponds to the directory it is in (more in Section 2.2.2). Functionality that is common to multiple architectures would reside in a common parent directory. For example, the common *TreeEvaluator* implementation for *I386* and *AMD64* architectures resides in a *TreeEvaluator* class in *X86* directory. Note that OMR consists of more than 1,000 classes that build up its various functionality across multiple architectures. However, only certain classes are meant to be extended by users. Such classes, called *extensible classes*, are tagged with a keyword, `OMR_EXTENSIBLE`, when they are declared.

2.2.2 Static Polymorphism. We use the notion of *specialization* to reflect on the "concreteness" of a namespace: a specialized namespace is one that contains the implementation of a specific architecture whereas a less specialized namespace is one that contains implementation common to multiple architectures. In that sense, a namespace that contains implementation for *AMD64* architecture is more specialized, or specific, than a namespace that contains implementation for *X86* architecture in general.

In order to explain the static polymorphism part of the variability implementation, we take one example of a class in OMR that has a varying implementation depending on the host architecture. As shown in Figure 2, the *MemoryReference* class is found in multiple namespaces, The namespace signifies the architecture that this MemoryReference implementation targets. Each *MemoryReference* class inherits from a class with the same name in a more general namespace. For example, `OMR::X86::AMD64::MemoryReference` inherits from `OMR::X86::MemoryReference`.

The current hierarchy means that there are multiple classes that have the name *MemoryReference*, but these classes have different namespaces and implementations depending on the target architecture: when compiling for *AMD64*, the appropriate *MemoryReference* class that should be used by the rest of the project is *OMR::AMD64::MemoryReference*, whereas when compiling for *I386*, the appropriate class is `OMR::I386::MemoryReference`. The idea is that we should simply be able to include and use *MemoryReference* anywhere in the code, and get the right implementation according to the architecture we are building for. To be able to accomplish such behavior, OMR developers created the namespace *TR* which always extends the right *MemoryReference* class. If, for instance, the *TreeEvaluator* class is calling a function from *MemoryReference*, it would use `TR::MemoryReference` and trust that the latter is pointing to the right MemoryReference hierarchy base. Hence, every hierarchy is expected to terminate with a 'concrete' class, which is in the *TR* namespace.

In order to find the right base class that `TR::MemoryReference` should extend, OMR developers created a Connector class for each hierarchy. For `MemoryReference`, that class would be `OMR::MemoryReferenceConnector`, and the `TR::MemoryReference` class would extend this connector class (See Figure 2). Each implementation of the `MemoryReference` would then have a `typedef` between `OMR::MemoryReferenceConnector` and its own class. Each typedef is protected (similar to the idea of include guards), such that only one typedef is seen by the compiler in a given configuration. Based on the include path prioritization (Section 2.2.3), the `typedef` in the most specialized class will always be seen first and thus will be the class that `TR::MemoryReference` extends. Note that all the concrete types to be used are determined at compile time.

In order to make sure that static polymorphism works as expected, OMR developers had to introduce a code convention that must be used. Consider Figure 3, when calling function `a()` from class C, the compiler would use the `this` pointer, which points to class C, and call `this->a()`. After not finding the function, it would refer to class B and try calling `this->a()`. Note that the `this` pointer is now pointing to class B and not class A. After not finding `a()` in class B, it would refer to class A and try calling `this->a()`. When executing the implementation of `A::a()`, the compiler would reach Line 6, where it should execute `this->b()`. Since the compiler is executing a function of class A, `this` points to class A, and `A::b()` would be executed. However, based on the intended OMR

Listing 1: Example to demonstrate the desired behavior with static polymorphism. On Line 6, function b from class B should be invoked instead of that from class A

```
1  class A{
2  public:
3    A() {};
4    void a() {
5      printf("function a from class A\n");
6      b(); //change to self()->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   return 0;
30 }
```

behavior where functionality from the most specific class needs to be called, the compiler is supposed to call `B::b()` since class B is the first parent class of C that contains an implementation of `b()`. To ensure this behavior, OMR developers created a `self()` function that always returns a pointer to the most specific or concrete class in a hierarchy; in our example, `self()` would return a pointer to C. Therefore, the convention in OMR is to use `self()` instead of `this` to call any fields or functions from an instance of an extensible class. Thus, the call on Line 6 would be `self()->b()`.

2.2.3 *Include Path Variability.*

The case in Figure 2 touches on another complication: if in some cases, the most specific architecture does not have a declaration of some class, how would the compiler know that it should search in the parent directory? Here is where the include path prioritization compiler feature is used.

In order to compile a `MemoryReference` class for *I386* architecture, the include paths passed to the compiler in the build system would be as follows: `-Iomr/compiler/x/i386/codegen -Iomr/compiler/x/codegen -Iomr/compiler/codegen`. The order is important, because compilers prioritize the paths passed first when resolving `#include` references. Hence, in the above case, the compiler would search for *OMRMemoryReference.hpp* in *omr/compiler/x/i386/codegen/* first and would not find the class (since there is no specific implementation for *I386*). After that, the compiler searches for it in *omr/compiler/x/codegen* and finds the file there.

2.2.4 *Hierarchies.*

Looking at the bigger picture, each class in OMR is basically a hierarchy of classes, starting from the *OMR* namespace, and ending in the most specific namespace of the architecture. It is important to note that at compile time, the hierarchy would be linear based on the architecture and language used. For example, when compiling for *AMD64* architecture, the hierarchy of classes for *MemoryReference* would be as follows: *OMR::MemoryReference –> OMR::X86::MemoryReference –> OMR::X86::AMD64::MemoryReference*. Whereas when compiling for *Power* architecture, the hierarchy would be: *OMR::MemoryReference –> OMR::Power::MemoryReference*.

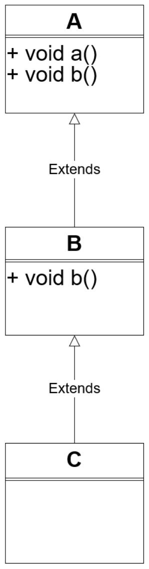


Figure 3: UML diagram for Listing 1

Since the library is intended to be used by a language developer, developers are expected to extend OMR's classes in a new namespace called after the target language. For example, when having Ruby as a consumer of the OMR technology on *AMD64*, developers would create `Ruby::MemoryReference` that extends `OMR::MemoryReferenceConnector` which, in that case, is typedef'd to `OMR::X86::AMD64::MemoryReference`.

2.3 OMRChecker

As mentioned in the introduction, this implementation of static polymorphism, include path variability, and directory structure obliges developers to follow some conventions when extending the extensible classes. One convention is that all implementations of a type are declared in classes with the the same name but reside in different namespaces (such as `OMR::X86::MemoryReference`). Another convention is using the `self()` function instead of the `this` pointer. In order to enforce such conventions on developers, OMR developers created a Clang plug-in, called OMRChecker, to statically analyze the code and verify that no client code breaks this convention. OMRChecker only checks classes tagged with `OMR_EXTENSIBLE` since these are the only classes meant to be extended.

3 PROBLEM DESCRIPTION

Polymorphism is one way of implementing variability where the same function may have different behavior depending on the receiver type [18]. The receiver type can either be determined at run time (*dynamic polymorphism*) or at compile time (*static polymorphism*). In this section, we discuss the pros and cons of static polymorphism when put into perspective with the OMR project, and how switching from static to dynamic polymorphism for the extensible parts of OMR may positively impact the project's source code and efficiency.

3.1 Dynamic vs Static Polymorphism

Dynamic polymorphism is implemented by adding the `virtual` keyword to the function that is intended to be overridden. To resolve a virtual function, the program follows a pointer to the right version of the function [14]. Due to the `virtual` keyword, dynamic polymorphism makes it easier to spot the variability points of a class, i.e., the functions whose functionality might differ in other classes in the hierarchy. However, dynamic polymorphism introduces a run-time overhead when following pointers to find the right implementation of a function. This might impact program performance if the number of virtual function calls is high, which is why OMR developers moved away from using dynamic polymorphism.

On the other hand, with static polymorphism, no `virtual` keyword is used. Given a function call, the compiler would start searching for the implementation of the function in the caller class. If the function is not found, the compiler would search for the function in the parents of that class and so on. Hence, at compile time, all functions are linked to their right implementation. This positively impacts the run time of the program, when comparing it to dynamic polymorphism.

3.2 Consequences of Static Polymorphism in OMR

The choice of static polymorphism in OMR has multiple downsides: (1) it does not clearly identify the extension points where client developers are supposed to extend OMR in their new language and (2) it obliges developers to follow certain code conventions for the polymorphism to work successfully, such as using the `OMR_EXTENSIBLE` tag and the `self()` function mentioned earlier.

These conventions may sometimes be a deterrent for language developers. For example, in the case of a class that has some functions that are intended to be extended and others that are not, a class tag would not be very helpful. In this case, using dynamic polymorphism and adding the keyword `virtual` for functions that should be extended would be a better idea.

Using the `self()` function when calling a method of the same object is not a C++ convention that developers are used to; the default way is to use `this`. Similarly, for class static functions, developers must take care to write their static function calls using the most concrete class in the *TR* namespace to ensure that static methods can be overridden via static polymorphism as well. Code with incorrect usage of these conventions will still successfully compile, and but will dispatch to the wrong function; this makes developing code for OMR unintuitive.

Therefore, in summary, static polymorphism is adding conventions that the community contributors are obliged to follow, making the code less approachable and harder to read.

3.3 Moving to Dynamic Polymorphism

Given the above downsides, some of the OMR developers are advocating the change to dynamic polymorphism. The current hypothesis is that the vast majority of specialization exists in methods that are not called with a sufficient frequency to substantially impact 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, we created a tool, OMRStatistics, that analyzes the methods and classes of OMR's source code. OMRStatistics can be helpful in multiple ways: first, the information provided would help check whether the amount of overridden functions in extensible classes is enough to significantly impact performance if virtualized. In addition, OMRStatistics will help OMR developers document the API boundaries, and reason about extensibility on a per-method/API basis. In summary, OMRStatistics helps developers reason about the variability in their source code.

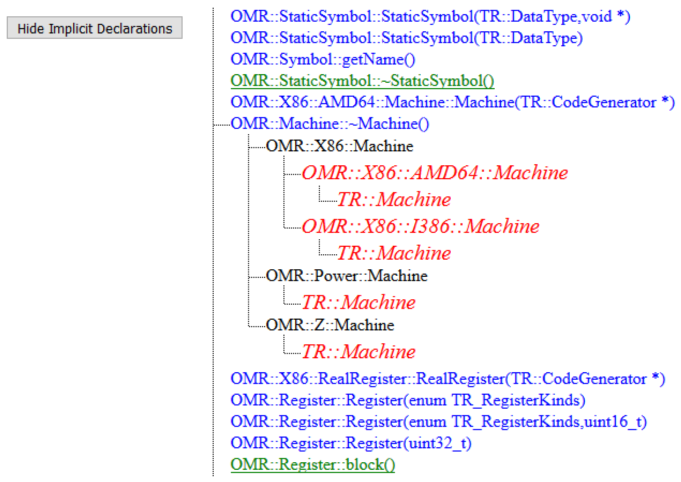
In fact, all the information collected in OMRStatistics is stored in a database, which helps OMR developers answer their questions through querying the database. Some of their main concerns when considering to change to dynamic polymorphism are found in these questions:

- Q1: How many total classes are there in OMR and how many of these are extensible classes?
- Q2: How many methods do we have in total and how many of them are overridden in client/extension code?
- Q3: Is most functionality added through static polymorphism?

4 OMRSTATISTICS

In order to get more information about the source code in OMR, we created OMRStatistics. OMRStatistics is an open source static analysis tool, built as a Clang plugin [4]. 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 database to make it easier for developers to query. Additionally, it provides visualizations of the information in the form of diagrams and HTML pages. While we currently only ran OMRStatistics on OMR, 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 work on any C++ project.

We now describe how OMRStatistics works, the output files it produces, and the database where this information is eventually stored.



```
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 4: 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 compatibe for black-and-white printing, we made only virtual functions underlined, and implicit functions are italized and have larger fonts. Pressing the button on top hides the implicit functions.

4.1 Output Files

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

- *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.
- *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.
- *functionLocation.csv*: contains the information that links every function to the source file location where it was declared.
- *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*.
- *overloads.csv*: includes a record for every override of a function in OMR. Each record includes the base class, the signature of the overridden function, and the overriding class
- *overrides.html*: a web page that visualizations the overrides present in OMR. Figure 4 shows an excerpt of this visualization.
- *hierarchy.pdf*: a PDF file that visualizes the class hierarchies. A screen shot of the visualization is in Figure 5

4.2 OMRStatistics Setup

Since OMRStatistics is implemented as a Clang plugin, it runs its analysis while compiling a given source file. In order to run

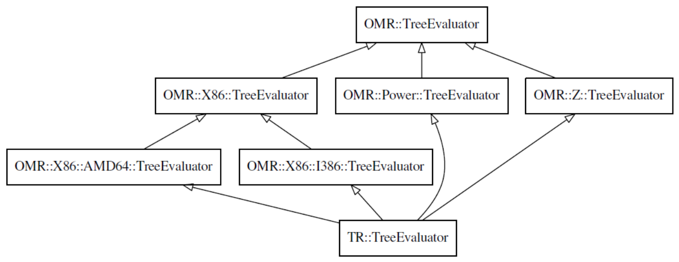

OMR::TreeEvaluator
OMR::X86::TreeEvaluator
OMR::Power::TreeEvaluator
OMR::Z::TreeEvaluator
OMR::X86::AMD64::TreeEvaluator
OMR::X86::I386::TreeEvaluator
TR::TreeEvaluator

Figure 5: Hierarchy visualization file

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. Although for now, we will 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 run the tool on less number of files [35].

4.3 OMRStatistics Approach

OMRStatistics works in three phases. The first phase starts by calling a class named `HMRecorder`, an extension of Clang's `RecursiveASTVisitor` class which goes through the declarations of classes and methods. `HMRecorder` processes all declarations found in the source code and saves the following: (1) a mapping between each class declaration and its method declarations, (2) whether classes are extensible or not, (3) whether a function is implicit or not, (4) mapping between each class declaration and its parent declarations.

The second phase is carried out in class `HMConsumer`, which consumes the information recorded by the HMRecorder, processes it, and outputs it in the relevant CSV output files. More specifically, the `HMConsumer` creates a node for each class or parent found in the previous phase, and connects them together according to the mapping between class and parent declarations. On the other hand, `HMConsumer` also creates a data structure for each method signature (called `MethodTracker` object). By using the mapping (provided by `HMRecorder`) between every class and its methods, the tool keeps track of all the method signatures occurrences in `MethodTrackers`.

The third phase is triggered automatically by the build system after compiling the project using Clang (the first and second phases). In this phase, python scripts process the CSV files trimming the outputs of duplicate records (due to compiling for all configurations, files might be compiled more than once, resulting in duplicate records), creating an SQL file to build the database, and generating multiple visualization files from the output files of previous phases.

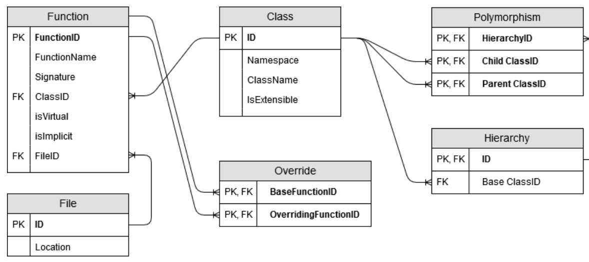


Figure 6: OMRStatistics relational database schema

4.4 Database

OMRStatistics produces a database that can be queried to find different information about how the functions are used. Figure 6 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.

5 RESULTS AND ANALYSIS

In this section, we discuss the three questions raised by OMR developers about their current static polymorphic implementation and how OMRStatistics is able to answer these questions and help them move one step closer to the best variability decision for the project. In order to explore the impact of moving forward with dynamic polymorphism, we run OMRStatistics on the souce code of OMR and OpenJ9. OpenJ9 is an IBM Java Virtual Machine that uses the OMR library and is the biggest consumer of OMR technology.

5.1 Data About Extensible Classes

Question 1: How many classes are in OMR altogether and how many of them are made into extensible classes? Querying the database shows that 149 of the 1365 classes in OMR (~10.61%) are marked as extensible. Considering only the extensible classes, the functions which OMR downstream projects are expected to extend are spread in 142 classes (~7.38% 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 desirable 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, the functions that are intended to be extension points will be made into virtual functions. This will make all possible extension points easy to find for OMR downstream project developers.

5.2 Data About Overridden Functions

Question 2: How many methods are there in all OMR extensible classes and how many of them get overridden?

The corresponding database queries reveal that OMR has 8,336 methods in extensible classes and only 466 of these methods, roughly 5.6%, are overridden. Based on the hypothesis of OMR developers (mentioned in section 3.3), the proposal is to virtualize functions that are extended in OMR. OMR developers are 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.

5.3 Data About Extensible Class Hierarchies

Question 3: Is most functionality added through static polymorphism?

New functionality in OMR can be added either by adding new methods in derived classes or by overriding existing methods and alerting their behavior. When analyzing the source code of the product resulting from OMR and OpenJ9, we find 83 extensible hierachies and calculated the average class hierarchy depth to be 4.05 classes. We also find that on average, only 14.15% of an extensible hierarchy's functions are overridden. This implies that the majority of variability points in OMR are not in the form of method overrides, but are instead in the form of additional new functionality in the derived classes. In other words, the low percentage of overridden functions suggests that client developers extend OMR mostly by adding new functionality in derived classes instead of overridding existing functions. Hence, moving from static polymorphism to dynamic polymorphism will only affect a low percentage of functions and variability points and hence is likely to have a minimal impact on the performance of OMR.

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.

5.4 Moving Forward with Dynamic Polymorphism

The previous facts and data demonstrate why it is favorable to change from static to dynamic polymorphism. However, there will be some extra work, beyond simply adding the `virtual` keyword to functions, as the existing variability mechanism allows certain imprecisions in implementation.

One such challenge will be missing base-class implementations. Consider Figure 7, a function `a()` is declared and implemented in class A of all supported architectures of OMR: `OMR::X86::A`, `OMR::Power::A`, `OMR::ARM::A`, and `OMR::Z::A`; however, it is never declared in `OMR::A`. This currently works with the use of `self()` when calling `a()` since the search for the function is bound to start at the bottom of the hierarchy. However, when virtualizing class A, if `OMR::A::b()` refers to `this->a()`, the compiler will throw an error as function `a()` is not defined in this class. One solution for this challenge is to statically analyze the function calls in the project, finding such cases and defining them in their respective class in the *OMR* namespace in order to prevent the project from throwing

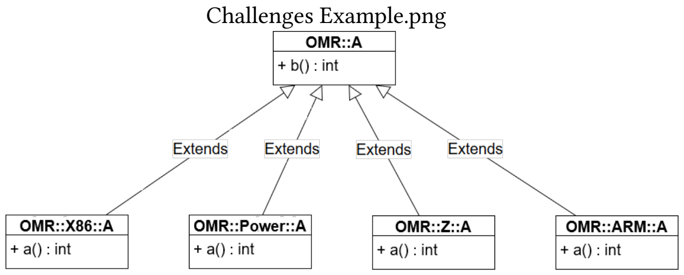


Figure 7: Example of foreseen challenge when changing to dynamic polymorphism

a compilation error when moving to dynamic polymorphism. This solution is part of our future work.

6 RELATED WORK

The goal of OMRStatistics is to help OMR developers make better design decisions related to the variability implementation of OMR; hence, we discuss related literature in the following directions: (1) variability implementation mechanisms, to review classical and previous variability mechanisms adopted by others, (2) variability evaluation metrics offering a better understanding of the effect variability implementations might have on configurable products, and (3) tools that support software variability. Given the industrial context of our paper, we give practical applications of related work, when applicable.

6.1 Variability Implementation Mechanisms

In their report [9], Bechmann. et al. discuss how to make educated decisions about including variability mechanisms in software product lines and provide an overview of some key variability implementations. Some of the mentioned implementations that we will be discussing further are: parameters, plug-ins or framework programming, and inheritance.

Parametrization is using parameters to change a general program's behavior [24]. The software would contain the implementation for all its variants and the parameters would control which variant is executed. MADAM [23] project is a practical example of a project that uses parametrization as part of their variability implementation.

Another variability implementation used in practice is the framework variability mechanism. It is based on using an initial implementation or code base that can be reused to create solutions for different use cases presented by costumers. What makes a framework based mechanism different than other variability mechanisms is the explicit extension points, called *hot spots* [8]. One subcategory of the framework variability mechanism is the black-box framework [40] which is used in Firefox [47]. A configurable software that is developed to conform to the black-box framework is designed such that it facilitates creating independent subapplications that extend its functionality. For example, Firefox has the concept of plugins, where user developers can create plugin software independantly without understanding the Firefox source code or how it is implemented.

However in practice, it is common for software to use multiple variability implementations at once. For instance, Mozilla Firefox uses the black-box component and the broker pattern [47]. The broker pattern [16, 39] falls under another variability implementation mechanism, design patterns. Other popular design patterns used in variability [8] include the observer pattern [25] and the template pattern [36].

Another way to implement variability is through inheritance [7]. In our previous position paper [35], we described the OMR project and how it uses static polymorphism to implement variability along with the challenges OMR developers currently face. The OMRStatistics tool we present in this paper is a concrete realization for addressing one of the challenges we discussed there.

For a complete list of possible variability implementation strategies, we refer the reader to the work by Apel et al. [7].

6.2 Variability Metrics

Various variability metrics have been proposed to evaluate variability implementations in product lines. Liebeg et al. [33] analyze more than forty CPP projects according to metrics introduced by the authors, suggesting alternative variability implementations. Hunsen et al. [26] study twenty seven CPP projects in order to study the similarity between CPP implementation in open source and industrial projects. In their paper, the authors define the similarity between implementations by a set of variability metrics. Andre et al. [46] use the concept of service utilization to come up with evaluation metrics for variability in SPLs.

Since OMRStatistics proposes altering the variability implementation mechanism in Eclipse OMR, a potential future work would be to use, an adaptation of, the variability metrics proposed in literature to assess the new variability implementation.

6.3 Tools Supporting Variability

There are various research and industrial tools developed to support reasoning about software variability.

Based on a survey done by Berger et al. [15], the two most common industrial variability modeling tools used to support variability are `pure::variants` and `GEARS`. `Pure::variants` is a feature modeling plugin for Eclipse that adds variability support to Eclipse [5] in order to support product line variability development. However, there are other feature modeling plugins for Eclipse such as `FeaturePlugin` [6]. There also exists tool suites that support variability development such as `DOPLER` [21] and the `AHEAD` [10].

On the other hand, `GEARS` is a code analysis tool developed by BigLever Software that focuses on software mass customization in software product lines [32]. Mass customization was also tackled by Ronny et al. [30]. In their research, Ronny et al. use the PuLSE approach [13] 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 [35].

Other tools support variability by analyzing the source code and providing insights about the project. For example, `FeatureIDE` [45] is a variability aware IDE that analyzes projects and maps their code artifacts to features. `RequiLine` [48] is a tool that supports requirements engineering in SPLs.

Another tool that offers variability aware analysis is Typechef [29]. Typechef aims to analyze all variants of a configurable software, implemented using C and the C preprocessor. It works by preprocessing the source code into conditionalized tokens that are then parsed into conditional ASTs. These ASTs are then used as the basis of many analyses. However, Typechef was developed to analyze software written in C (and Java). It currently does not support C++ projects. In addition, it supports projects whose variability is implemented using only preprocessor directives, which is not the case in Eclipse OMR. Hence, we were not able to leverage its capabilities in our research.

Various other techniques are used to support variability. Eisenbarth et al. [22] 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. [31] to extract configuration dependencies for projects whose variability is implemented with C preprocessors (CPP) [26], and by Loesh et al.[34] to visualize product features and configurations. Another tool that supports projects with CPP variability implementations is created by Baxter et al. [11]. Their tool uses DMS, a transformation system used to gradually alter and orient a software's design for more efficient maintenance [12] 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.

7 CONCLUSION

In this paper, we presented an industry system, Eclipse OMR, that uses static polymorphism to impelement variability. We provide evidence to help developers assess how changing OMR's variability implementation mechanism to dynamic polymorphism will impact the overall performance and complexity of the project.

During the development of OMR, developers 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 hypothesis of OMR developers is that virtualizing the functions that act as extension points of the project will not have a significant impact on the runtime performance. Attempting to motivate or discourage this hypothesis, we created a tool, OMRStatistics, that statically analyzes the code and extracts the information needed to make this decision.

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. Our future plan is to gradually suggest to OMR developers functions that can be changed to use dynamic polymorphism instead of static polymorphism, and measure the impact on runtime performance.

8 ACKNOWLEDGEMENT

This project is funded by IBM CAS 2017 Project #40.

REFERENCES

[1] [n. d.]. clang: a C language family frontend for LLVM. ([n. d.]). https://clang.llvm.org/

[2] [n. d.]. Eclipse OMR: Building Language Runtimes for the Cloud. ([n. d.]). https://www.eclipse.org/community/eclipse_newsletter/2016/october/article5.php

[3] [n. d.]. The LLVM Compiler Infrastructure. ([n. d.]). http://llvm.org/

[4] [n. d.]. OMRStatistics Github Repository. ([n. d.]). https://github.com/samasri/omr/tree/master/tools/compiler/OMRStatistics

[5] [n. d.]. pure::variants Eclipse Plug-in User's Guide. ([n. d.]). http://web.pure-systems.com/fileadmin/downloads/pv_userguide.pdf

[6] Michal Antkiewicz and Krzysztof Czarnecki. 2004. FeaturePlugin: Feature Modeling Plug-in for Eclipse. In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange (eclipse '04)*. ACM, New York, NY, USA, 67–72. https://doi.org/10.1145/1066129.1066143

[7] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Publishing Company, Incorporated.

[8] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Publishing Company, Incorporated.

[9] Felix Bachmann and Paul Clements. 2005. *Variability in Software Product Lines.* Technical Report CMU/SEI-2005-TR-012. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7675

[10] Don Batory. 2004. Feature-Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 702–703. http://dl.acm.org/citation.cfm?id=998675.999478

[11] Ira Baxter and Michael Mehlich. 2001. Preprocessor conditional removal by simple partial evaluation. In *Proceedings Eighth Working Conference on Reverse Engineering*. 281–290. https://doi.org/10.1109/WCRE.2001.957833

[12] Ira Baxter and Christopher W. Pidgeon. 1997. Software change through design maintenance. In *1997 Proceedings International Conference on Software Maintenance*. 250–259. https://doi.org/10.1109/ICSM.1997.624252

[13] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean-Marc DeBaud. 1999. PuLSE: a Methodology to Develop Software Product Lines. (01 1999), 122-131 pages.

[14] Eli Bendersky. [n. d.]. The Curiously Recurring Template Pattern in C++. ([n. d.]). https://eli.thegreenplace.net/2011/05/17/the-curiously-recurring-template-pattern-in-c/

[15] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wkasowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '13)*. ACM, New York, NY, USA, Article 7, 8 pages.

[16] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture.*

[17] Rafael Capilla, Jan Bosch, Kyo-Chul Kang, et al. 2013. Systems and Software Variability Management. *Concepts Tools and Experiences* (2013).

[18] John Cary, Svetlana G. Shasharina, Julian Cummings, John V. W. Reynders, and Paul Hinker. 1997. Comparison of C++ and Fortran 90 for object-oriented scientific programming. 105 (09 1997), 20–36.

[19] Paul Clements and Linda Northrop. 2001. Software product lines: Patterns and practice. *Boston, MA, EUA: Addison Wesley Longman Publishing Co* (2001).

[20] IBM Corporation. [n. d.]. The OMR Project Source Code. ([n. d.]). https://github.com/eclipse/omr

[21] Deepak Dhungana, Rick Rabiser, Paul Grünbacher, and Thomas Neumayer. 2007. Integrated Tool Support for Software Product Line Engineering. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, New York, NY, USA, 533–534.

[22] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. 2003. Locating Features in Source Code. *IEEE Trans. Softw. Eng.* 29, 3 (March 2003), 210–224. https://doi.org/10.1109/TSE.2003.1183929

[23] Jackeline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. 2006. Using Architecture Models for Runtime Adaptability. *IEEE Software* 23, 2 (March 2006), 62–70.

[24] Hassan Gomaa and Diana L Webber. 2004. 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.*

[25] Jan Hannemann and Gregor Kiczales. 2002. Design Pattern Implementation in Java and aspectJ. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. ACM, New York, NY, USA, 161–173.

[26] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kastner, Olaf Lessenich, Martin Becker, and Sven Apel. 2016. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering* 21, 2 (01 Apr 2016), 449–482. https://doi.org/10.1007/s10664-015-9360-1

[27] Christian Kastner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 311–320. https://doi.org/10.1145/1368088.1368131

[28] Christian Kastner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation. *SIGPLAN Not.* 46, 10 (Oct. 2011), 805–824.

[29] Andy Kenner, Christian Kastner, Steffen Haase, and Thomas Leich. 2010. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development (FOSD '10)*. ACM, New York, NY, USA, 25–32. https://doi.org/10.1145/1868688.1868693

[30] Ronny Kolb, Dirk Muthig, Thomas Patzke, and Kazuyuki Yamauchi. 2005. A case study in refactoring a legacy component for reuse in a product line. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. 369–378. https://doi.org/10.1109/ICSM.2005.5

[31] Maren Krone and Gregor Snelting. 1994. On the Inference of Configuration Structures from Source Code. In *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 49–57. http://dl.acm.org/citation.cfm?id=257734.257742

[32] CharlesW. Krueger. 2002. Easing the Transition to Software Mass Customization. In *Software Product-Family Engineering*, Frank van der Linden (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 282–293.

[33] J. Liebig, S. Apel, C. Lengauer, C. KĂďstner, and M. Schulze. 2010. An analysis of the variability in forty preprocessor-based software product lines. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 105–114.

[34] Felix Loesch and Erhard Ploedereder. 2007. Optimization of Variability in Software Product Lines. In *11th International Software Product Line Conference (SPLC 2007)*. 151–162. https://doi.org/10.1109/SPLINE.2007.31

[35] Samer Al Masri, Nazim Uddin Bhuiyan, Sarah Nadi, and Matthew Gaudet. 2017. 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 '17)*. IBM Corp., Riverton, NJ, USA, 285–291. http://dl.acm.org/citation.cfm?id=3172795.3172831

[36] James E. McDonough. 2017. *Template Method Design Pattern.* Apress, Berkeley, CA. 247–254 pages.

[37] Sarah Nadi, Christian Dietrich, Reinhard Tartler, Richard C. Holt, and Daniel Lohmann. 2013. Linux Variability Anomalies: What Causes Them and How Do They Get Fixed?. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 111–120.

[38] Hendrik Post and Carsten Sinz. 2008. Configuration Lifting: Verification Meets Software Configuration. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. IEEE Computer Society, Washington, DC, USA, 347–350.

[39] Frank Puhlmann, Arnd Schnieders, Jens Weiland, and Mathias Weske. 2005. *Variability Mechanisms for Process Models.*

[40] Han Albrecht Schmid. 1997. Systematic Framework Design by Generalization. *Commun. ACM* 40, 10 (Oct. 1997), 48–51.

[41] Julio Sincero, Horst Schirmeier, Wolfgang SchrĂűder-Preikschat, and Olaf Spinczyk. 2007. Is The Linux Kernel a Software Product Line?. In *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*.

[42] Stephane S. Some and Timothy C. Lethbridge. 1998. Parsing minimization when extracting information from code in the presence of conditional compilation. In *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on.* 118–125.

[43] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. 2011. Configuration Coverage in the Analysis of Large-scale System Software. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS '11)*. ACM, New York, NY, USA, Article 2, 5 pages. https://doi.org/10.1145/2039239.2039242

[44] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)* 47, 1 (2014), 6.

[45] Thomas Thum, Christian Kastner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70 – 85. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

[46] Andre van der Hoek, Ebry Dincel, and Nenad Medvidovic. 2003. Using service utilization metrics to assess the structure of product line architectures. In *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No.03EX717)*. 298–308. https://doi.org/10.1109/METRIC.2003.1232476

[47] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. 2001. On the notion of variability in software product lines. In *Proceedings Working IEEE/IFIP Conference on Software Architecture.* 45–54.

[48] Thomas von der MaÃ§en and Horst Lichter. 2004. RequiLine: A Requirements Engineering Tool for Software Product Lines. In *Software Product-Family Engineering.* Springer Berlin Heidelberg, Berlin, Heidelberg, 168–180.

[49] Minghui Yang, Chandrasekharan Iyer, and Charles Wetherell. 2006. Method and apparatus for performing conditional compilation. (2006).