Software Variability Through C++ Static Polymorphism

A Case Study of Challenges and Open Problems in Eclipse OMR

Samer AL Masri[†], Nazim Uddin Bhuiyan[†], Sarah Nadi[†], Matthew Gaudet[‡]
University of Alberta[†], IBM Canada[‡]
{almasri,nazimudd,nadi}@ualberta.ca,magaudet@ca.ibm.com

ABSTRACT

Software Product Line Engineering (SPLE) creates configurable platforms that can be used to efficiently produce similar, and yet different, product variants. SPLs are typically modular such that it is easy to connect different blocks of code together, creating different variations of the product. There are many variability implementation mechanisms to achieve an SPL. This paper shows how static polymorphism can be used to implement variability, through a case study of IBM's open-source Eclipse OMR project. We discuss the current open problems and challenges this variability implementation mechanism raises and highlight technology gaps for reasoning about variability in OMR. We then suggest steps to close these gaps.

1 INTRODUCTION

As software becomes more pervasive, the same system often has to work in various environments, and also cater to specific customer needs. For example, the underlying firmware of a printer needs to support different models, some of which may have additional functionality such as supporting email scans. *Software variability* is the ability of a software system or artifact to be efficiently extended, changed, customized, or configured for use in a particular context [12]. To avoid creating a new software system for each new platform or customer, *Software Product Lines* (SPLs) were introduced. An SPL is a collection of similar software products that support different *features* (units of functionality), but share a common code base [5]. SPLs provide a simple but efficient way of implementing software variability to reuse existing programs [11]. SPLs have been adopted in both industry and open source systems [5].

In order to create SPLs, it is important to have an underlying variability implementation mechanism that separates common code, or functionality, that is included in every product from configurable functionality that is only included in certain products. Developers can then configure the system to indicate the desired functionality. There are many possible variability implementation mechanisms, such as using a preprocessor, parameters, the build system, feature-oriented programming, or even simply design patterns [5]. Regardless of the mechanism, a system with n optional, configurable features can result in 2^n product variants, which makes it hard to reason about all the SPL products. *Variability-aware* analyses, or *family-based* analyses [5], where all variants of a software are simultaneously analyzed, emerged to help with such reasoning. While there are many variability implementation mechanisms [5], most of the literature that studied variability in practice focused on build-time variability, specifically using a pre-processor for conditional compilation. However, different variability implementation mechanism pose different challenges. Exploring variability implementation mechanisms used in practice and understanding the challenges they impose on family-based analyses is important to drive SPL research forward and ensure technology transfer.

In this work, we discuss how *static polymorphism* in `C++` can be used to implement variability in an industrial setting. Specifically, we discuss IBM's open-source *Eclipse OMR* [2] project, a language-agnostic library of run-time components developed in C++ which leverages variability to support any programming language on multiple architectures. We discuss the challenges involved in OMR's variability design and highlight the current technology gaps in supporting its maintenance through reasoning about all variants of the system. We sketch out a path to bridge this gap by describing how variability-aware analyses for this setting can be designed.

2 BACKGROUND AND RELATED WORK

Variability Implementation Mechanisms There are different binding times for software variability such as at build-time, load-time, and run-time. There are various variability implementation mechanisms that can be used for the different binding times [5, 25], 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* [8]. FOP is a composition-based approach for building software by dividing an SPL into feature modules [5]. Several tools and programming languages were proposed to support feature-oriented programming., including the AHEAD tool suite [8], FeatureHouse [6], and FeatureC++ [7]. 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.

The most studied variability implementation mechanism in the literature is using the `C` preprocessor's `#ifdef` directives. The Linux kernel in particular, with over 12,000 configurable features, is a popular subject of study. Many researchers studied the evolution of the Linux kernel (e.g., [19]) or inconsistencies that arise from relying on the `C` preprocessor (e.g. [21, 26]). Examples of other discussed mechanisms include plug-in based systems (e.g., WordPress [22]), systems with run-time variability using `if` conditions (e.g., Mozilla Firefox [9]), systems using feature toggles (Google Chrome) [23],

CASCON '17, October 2017, Markham, ON, CA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and systems with load-time configuration options (e.g., Android apps [18]). While object-orientation and design patterns have been discussed as variability implementation mechanisms [5], to the best of our knowledge, there has been no work that discusses a large industrial or open-source project that uses these variability mechanisms. Specifically, we found no work discussing the impact and challenges of using static polymorphism to implement variability.

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. Thüm et al. [27]'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 [17] and SuperC [15] 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. [16] 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 [4]. Similar variability-aware analysis efforts have been proposed for systems using load-time or run-time variability [22, 27]. 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.

3 ECLIPSE OMR BACKGROUND

Eclipse OMR [2] is an open-source `C++` library introduced by IBM. It consists of multiple components for building language run-times, such as a compiler, garbage collector (`GC`), 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.

Project History OMR traces back to the IBM Java Virtual Machine, J9, as well as its Just in Time (JIT) compiler, Testarossa [24]. Testarossa is a multi-target compiler technology that translates Java bytecode to machine code, in order to accelerate program execution. It currently supports X86, Power, Z, and ARM platforms.

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. The idea of OMR was that many SPLs can be built from the core component 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.

As a result, the compiler component was refactored to use static polymorphism to express variability, moving language-specific code into subclasses. The `GC` also used subclasses; however, it has so far used dynamic dispatch to reflect polymorphism, although a move to delegation is afoot at the time of writing. In our previous work [14], we described the refactoring process of the compiler component, as well as the lessons learned from it, but without focusing on the SPL and variability perspective – which is the focus of this paper.

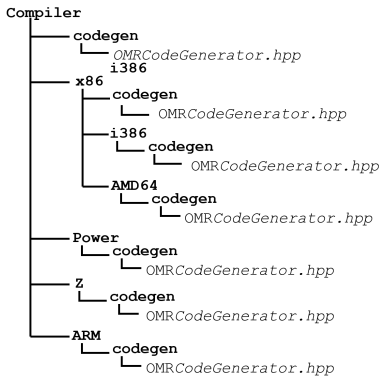
Project Structure and Size OMR has 777,546 lines of code (LOC) according `SLOCCount` [3] in June 2017, and 73 contributors according to GitHub's statistics. Most of the source files (80%) are `cpp` files, and the source code is divided into individual component directories: Compiler in `compiler`, Garbage Collector in `gc`, etc. This allows the flexibility of having different variability implementation in different components. For example, the `GC` component heavily uses `#ifdef` directives whereas the Compiler mostly uses static polymorphism. This paper will focus on the variability implementation in the Compiler component.

Static Polymorphism *Polymorphism* is commonly used in object-oriented languages to enable access to multiple related behaviors, and can be used to implement variability [5]. In `C++`, polymorphism is commonly implemented using *virtual* functions that are dynamically bound to their implementation. This is called *dynamic polymorphism*, and is achieved by having pointers to direct the function to the right implementation [10]. However, dynamic polymorphism adds performance overhead [13], which is why OMR resorted to *static polymorphism* that resolves all inheritence chains and function calls at compile time. Static polymorphism is usually associated with the Curious Recurring Template Pattern (CRTP) [10]. In C++, CRTP exploits template classes and static casting to implement inheritance. However, OMR uses a different implementation of static polymorphism that we explain in detail in Section 4.

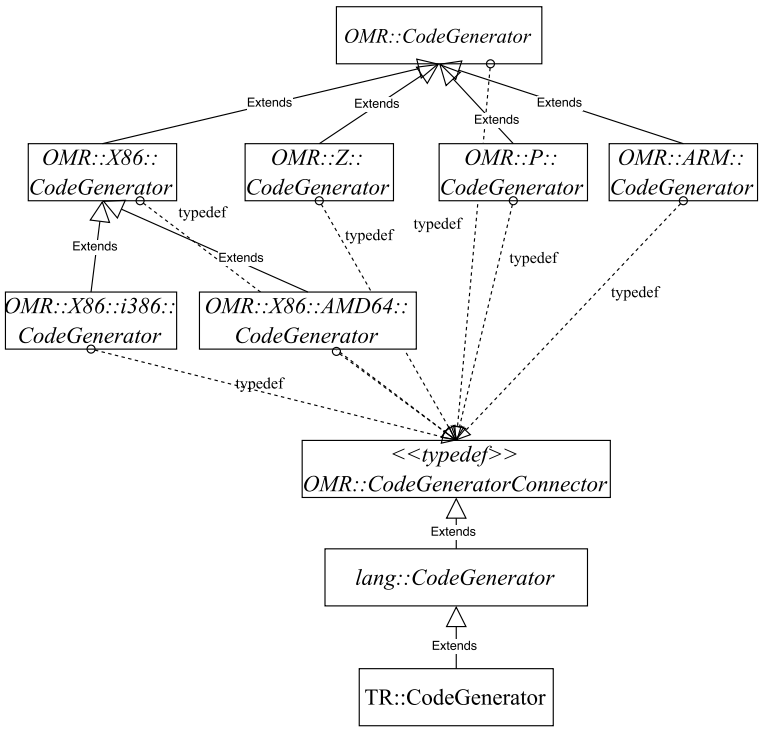
4 VARIABILITY IMPLEMENTATION IN OMR

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 general, the variability implementation mechanism used in the compiler component is *static polymorphism*. In terms of implementation, OMR developers initially wanted to use the typical CRTP used to implement static polymorphism (see Section 3). However, due to the concern of introducing large amounts of template code that may bloat the system, a customized notion of static polymorphism was



(a) Directory structure of the compiler component



(b) The `CodeGenerator` hierarchy. Classes in italic represent abstract classes.

Figure 1: Example file hierarchy and inheritance hierarchy in Eclipse OMR

created based on the idea of extensible classes. *Extensible 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. This special organization depends on several building blocks that we now discuss.

4.1 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. `CodeGenerator` is one of the 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 code across all products is placed in a directory directly under `compiler`, whereas the specific code that customizes `CodeGenerator` for the Z architecture, for example, is inside the `z` directory. Figure 1a shows the directory structure of the compiler component.

4.2 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 1a. Each `CodeGenerator` class extends its less specific counterpart in the parent directory. 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 1a, has the namespace `OMR`, while the rest of the classes have nested namespaces according to the corresponding architecture. Figure 1b shows the full extensible class hierarchy for `CodeGenerator`.

4.3 Connectors

Note how all classes 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 class hierarchy for every supported namespace. 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

```
#ifndef OMR_CODEGENERATOR_CONNECTOR
#define OMR_CODEGENERATOR_CONNECTOR
#else
#error multiple definition of OMR::X86::i386::CodeGenerator
#endif
namespace OMR {typedef
            OMR::X86::i386::CodeGenerator CodeGeneratorConnector;}
namespace OMR {
  namespace X86 {
    namespace i386 {
      class OMR_EXTENSIBLE CodeGenerator :
                    public OMR::X86::CodeGenerator {
        ...
      }
}}}
```

Listing 1: Using typdef to connect OMR::X86::i386::CodeGenerator to OMR::CodeGeneratorConnector

still uncertain due to architecture variability. For example, assume a function `f()` is implemented in the `CodeGenerator` classes in all namespaces. If the language-specific `CodeGenerator` class tries to call `f()`, the compiler would complain that `f()` is ambiguous.

With static polymorphism, all variability in the inheritence 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 class called *connector* for each existing extensible class hierarchy. For example, *CodeGenerator* has a corresponding *CodeGeneratorConnector* class; another extensible class hierarchy in `compiler` called `OMRMachine` has an `OMRMachineConnector`, and so on. The connector acts as a liaison between the class hierarchy it is representing and external classes that aim to extend or use this hierarchy. For example, if `CodeGenerator` needs to access a function from `OMRMachine`, it will use `OMRMachineConnector` instead of `OMRMachine`.

We use `CodeGenerator` to explain how connectors bridge the gap between the correct most-derived class of the extensible class hierarchy and other OMR classes. 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 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::CodeGeneratorConnector`. 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 1b. Hence, as long as there is a way to guide the compiler to detect the typedef in the most derived class first, `CodeGeneratorConnector` 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 Section 4.5.

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 1b. 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`).

4.4 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 2, where class `A` is the base class of the class hierarchy (similar to `OMR::CodeGenerator`) and `C` is the most-specific implementation of the hierarchy (similar to `OMR::X86::AMD64::CodeGenerator`). 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 if it is from inside class `A`, the most specific implementation of `b()`, which is in class `B` in this case, is executed. Hence, the following is the expected output from the code in Figure 2: `function a from class A` followed by `function b from class B`. However, if we run the program using `C++`'s default behavior, we get: `function a from class A` followed by `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) of `C`. When we call `a()` from an instance of `C`, the self-pointer now points to class `A`, since `a()` is only found in class `A`. When we call `b()` from inside `a()`, we are implicitly calling `this->b()`. Since the self-pointer, at this point, is pointing to `A`, it will directly call `A::a()`.

However, OMR developers want to force the program to start searching for `b()` from the bottom of the class hierarchy again. To solve this problem, OMR developers created a function `self()` that always returns a fresh pointer of the target class. Hence, instead of calling `b()` from function `a()`, we call `self()->b()`. `self()` will return a pointer to class `C`, because it forces the program to look from the bottom of the hierarchy again to get the most specific implementation available. Given that the `TR` namespace contains most of the concrete classes of the different extensible hierarchies, `self()` usually returns a pointer of the concrete class in `TR`.

`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`. The linter checks that concrete classes in class hierarchies are in the correct namespace (usually the `TR` namespace) and `self()` replaces `this` in the appropriate places [20]. `OMR_EXTENSIBLE` is a code annotation introduced in OMR. When found before the declaration of a class, `OMRChecker`

will check this class and ensure that the above three rules are applied. OMRChecker can currently check one architecture at a time, but is mainly used for checking the x86 architecture.

4.5 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, 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 to a different class.

4.6 Ifdefs

OMR also uses #ifdef directives to implement variability, especially in the GC 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 GC 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.

5 CHALLENGES AND TECHNOLOGY GAPS

OMR's unique combination of #ifdefs, static polymorphism through include paths, and typdef connectors makes it hard to collectively reason about the variability of the system. This is in addition to the natural complexity of C++, such as polymorphism and templates. This section presents the current technology gaps and challenges for analyzing variability in OMR, and outlines ways to solve them.

Desired Support

OMRChecker is currently the only reasoning tool developers have. To run OMRChecker, developers need to set up the appropriate include paths for a particular architecture and language combination. They currently mostly run it for x86 and the main OMR project. Ideally, the checker should be able to simultaneously check all possible variants of OMR and notify the relevant developers about missing self() checks. Even further, OMRChecker currently only checks for the correct use of self() and the OMR_EXTENSIBLE attribute.

```
class A {
public:
    A() {};
    void a() {
        printf("function a from class A\n");
        b();
    }
    void b() {
        printf("function b from class A\n");
    }
};

class B : public A {
public:
    B() {};
    void b() {
        printf("function b from class B\n");
    }
};

class C : public B {
public:
    C() {};
};

int main() {
    C instance;
    instance.a();
    return 0;
}
```

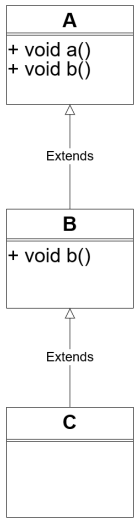


Figure 2: Example explaining the need for the self() function

There are many more queries and analyses that OMR developers can run to support them in the maintenance and evolution of OMR. For example, understanding if a given method is overridden and on which configurations can help developers refactor the system to improve modularity. If a method is only used by a particular language or architecture, then perhaps this method should be moved to a lower class in the hierarchy. Furthermore, checking if a member function always resolves to the same class or to different classes in the supported languages is useful for understanding the system behavior. Ideally, such queries and checks can be implemented as part of OMR's continuous integration to detect problematic changes.

Providing developers with such support indicates the need for variability-aware analysis of the system. A variability-aware analysis framework allows developers to implement different checks and queries and leave it up to the framework to *lift* the given analysis to the whole SPL. Such a framework must be able to handle C++ code with all its features such as polymorphism, templates, function overloading, as well as #ifdefs, and variations in include paths determined by the build system. To the best of our knowledge, there exists no variability-aware analysis framework that can handle this setup. Existing frameworks are mostly for analyzing C, cannot handle variability due to different include paths, and are mostly standalone research-effort tools.

Instead of building a framework from scratch, we suggest using an existing analysis framework. Ideally, it should be one that OMR developers and the open-source community are already familiar with. Since OMRChecker is implemented as a Clang plugin, and future queries that developers might need can easily be implemented in Clang, we believe that modifying Clang to become variability-aware is the best way forward. We now discuss three main challenges involved in creating a variability-aware analysis tool for OMR, and how we will use Clang to solve them.

Handling of Include Paths

Challenge As discussed in Section 4.5, OMR leverages the compiler's prioritization of include paths to achieve variability. We call

the include paths that correspond to one platform a *set* of include paths. Checking multiple platforms is challenging since the analysis, and in turn the compiler, needs to execute the same prioritization process for multiple *set*s of include paths. Hence, the challenge is to make Clang search in all the provided include path *set*s every time it finds an `#include` directive. Having nested `#include` directives inside header files also adds complexity to the matter.

Suggested Solution Currently, `#include` directives are handled by searching for the included file in the include paths, creating a lexer instance for the corresponding file, and then adding this lexer to a stack of lexers. Control then goes back to the pre-processor which will take the top lexer from the stack and lex the tokens inside. The way Clang handles include paths needs to change. We will change the searching step so that instead of searching for the files in one set of include paths, we can search for the file in all the sets that are provided. In other words, when having 3 platforms, Clang should be provided with 3 sets of include paths, one for each platform. While searching for a file, 3 files are expected to be found, each corresponding to a different platform. Since the 3 files will be added to the same stack of lexers, we are in the process of designing a mechanism to separate the lexer instances from each other based on what platform they belong to. To ensure efficiency over a brute force mechanism, we will leverage any sharing that occurs. In other words, `#include`s that resolve to the same path on all architectures, ie., common code, should be processed only once. However, to create a ground truth and a benchmark to compare against, and to get quick feedback from developers about the analyses they might want to implement on top of a system that can handle variable include paths, we are currently implementing a brute force solution as a first step. We are making changes to the main driver of Clang to run multiple compiler instances in a single Clang execution, with each compiler instance having its own set of include paths. Our next step is to explore the commonalities and exploit sharing.

Handling of #ifdefs

Challenge `#ifdefs` are another source of variability in OMR. Although `#ifdef` directives are not heavily used in the compiler component, an accurate analysis still needs to consider them. C++ adds more language features over C, which when combined with `#ifdefs` increases the complexity of variability-aware analysis.

Suggested Solution While TypeChef [17] or SuperC [15] may be adapted for C++, we believe that it is more practical to extend these ideas to Clang to provide a framework that is more popular among systems developers. We plan to adapt TypeChef's idea of presence conditions [17] and integrate it with Clang. *Presence conditions* are attributes assigned to tokens if they are to be consumed only if a certain macro is defined. We plan to change Clang's code at the lexing stage to assign a condition to a token if the token is within a conditional block. To gain efficiency over brute force method, we will change Clang's parser to split and join processing as it enters and exits from conditional blocks similar to TypeChef [17].

Handling of C++ features

Challenge Variations in include paths and macro definitions can completely alter the structure of classes and templates and inheritance hierarchies of derived classes. C++ is a complex programming language that requires careful analysis. No existing tools can currently perform a variability-aware analysis of full C++ code.

Suggested Solution We will leverage Clang's existing infrastructure to be able to analyze C++ code. As a first step, we will analyze programs using limited C++ features, and improve our tool in several iterations until it can handle all C++ features.

6 CONCLUSION

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, include path variation, and #ifdefs. 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, we also highlighted the current technology gaps that prevent simultaneous reasoning about all possible OMR configurations. Such reasoning is important to allow OMR developers to continue evolving and maintaining the system. Following existing literature on variability-aware analysis of C code, we plan to extend Clang to create a variability-aware analysis framework for C++ code that can handle extensible classes, include path variation, and #ifdefs. Clang is a free, open-source, and widely used compiler framework, and many developers use it to build code-analysis plugins. While Eclipse OMR is our inspiration for this work, making Clang variability-aware, especially with respect to handling commonly used #ifdefs, means that any existing Clang plugin can be lifted to analyze a whole SPL written in C++. Our solution would benefit a wide range of developers, helping them reason about variability in C++ code.

ACKNOWLEDGEMENT

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

REFERENCES

[1] A C Language Family Frontend for LLVM, 2007. http://clang.llvm.org/.
[2] Eclipse OMR, 2016. https://github.com/eclipse/omr.
[3] D. A. Wheeler. SLOCCount, 2004. https://www.dwheeler.com/sloccount/.
[4] S. Anand, P. Godefroid, and N. Tillmann. *Demand-Driven Compositional Symbolic Execution*, pages 367–381. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
[5] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-oriented Software Product Lines: Concepts and Implementation.* Springer Science & Business Media, 2013.
[6] S. Apel, C. Kästner, and C. Lengauer. Language-independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013.
[7] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On The Symbiosis of Feature-oriented and Aspect-oriented Programming. In *Proc. of the 4th International Conference on Generative Programming*, volume 3676 of *GPCE '05*, pages 125–140. Springer, 2005.
[8] D. Batory. Feature-oriented Programming and the AHEAD Tool Suite. In *Proc. of the 26th International Conference on Software Engineering*, pages 702–703, 2004.
[9] 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*, ESEC/FSE 2015, pages 295–306. ACM, 2015.
[10] E. Bendersky. The Curiously Recurring Template Pattern in C++, 2011. http://eli.thegreenplace.net/2011/05/17/the-curiously-recurring-template-pattern-in-c/.
[11] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 355–364, 2013.
[12] R. Capilla, J. Bosch, and K.-C. Kang. *Systems and Softw. Variability Manag.* 2013.
[13] K. Driesen and U. Hölzle. The Direct Cost of Virtual Function Calls in C++. *SIGPLAN Not.*, 31(10):306–323, Oct. 1996.

[14] 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*, VMIL 2016, pages 24–27. ACM, 2016.

[15] 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*, PLDI '12, pages 323–334. ACM, 2012.

[16] 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*, pages 196–206, 2000.

[17] 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.*, 46(10):805–824, Oct. 2011.

[18] M. Lillack, C. Kästner, and E. Bodden. Tracking Load-time Configuration Options. In *Proc. of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 445–456. ACM, 2014.

[19] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski. Evolution of the Linux Kernel Variability Model. In *Proc. of the 14th International Conference on Software Product Lines: Going Beyond*, SPLC'10, pages 136–150, 2010.

[20] D. Maier, M. Gaudet, and L. Lorimer. Extensible Classes, 2016. https://github.com/eclipse/omr/commits/master/doc/compiler/extensible_classes/Extensible_Classes.md.

[21] 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*, CSMR '12, pages 107–116, 2012.

[22] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring Variability-aware Execution for Testing Plugin-based Web Applications. In *Proc. of the 36th International Conference on Software Engineering*, ICSE '14, pages 907–918. ACM, 2014.

[23] 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)*, pages 201–211, May 2016.

[24] 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.

[25] M. Svahnberg, J. van Gurp, and J. Bosch. A Taxonomy of Variability Realization Techniques: Research Articles. *Softw. Pract. Exper.*, 35(8):705–754, July 2005.

[26] R. Tartler, D. Lohmann, J. Sincero, and W. Schroder-Preikschat. Feature Consistency in Compile-time-configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proc. of the 6th Conference on Computer Systems*, EuroSys '11, pages 47–60. ACM, 2011.

[27] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, June 2014.