Challenges of Implementing Software Variability in `Eclipse OMR`: An Interview Study

Batyr Nuryyev, Sarah Nadi
University of Alberta, Edmonton, Canada
{nuryyev, nadi}@ualberta.ca

Nazim Uddin Bhuiyan, Leonardo Banderali
IBM, Markham, Canada
{nazim.uddin.bhuiyan, leob}@ibm.com

***Abstract*—Software variability is the ability of a software system to be customized or configured for a particular context. In this paper, we discuss our experience investigating software variability implementation challenges in practice. `Eclipse OMR`, developed by IBM, is a set of highly configurable C++ components for building language runtimes; it supports multiple programming languages and target architectures. We conduct an interview study with 6 `Eclipse OMR` developers and identify 8 challenges incurred by the existing variability implementation, and 3 constraints that need to be taken into account for any re-engineering effort. We discuss these challenges and investigate the literature and existing open-source systems for potential solutions. We contribute a solution for one of the challenges, namely adding variability to enumerations and arrays. We also share our experiences and lessons learned working with a large-scale highly configurable industry project. For example, we found that the "latest and greatest" research solutions may not always be favoured by developers due to small practical considerations such as build dependencies, or even C++ version constraints.**

***Index Terms*—software variability, variability implementation, language runtimes, Eclipse OMR**

I. Introduction

Many software systems are designed to be configurable and adaptable to various user requirements or to the context they run in. For example, an operating system may run on a desktop or on a mobile device. Concepts such as *software product lines* [18], [27] (SPLs) or *software families* [21] have been introduced to deal with such needs. There has been extensive work studying open-source configurable systems [9], [26], [32], [46], [48], [51], [53] as well as a few industrial case studies [12], [23], [28], [52]. New programming paradigms to support variability implementation have also been introduced, including aspect-oriented programming [29], feature-oriented programming [43], and delta-oriented programming [45]. However, with the exception of basic preprocessor-based or feature-toggle based mechanisms [22], many of these paradigms find their place mainly in academic papers, with limited adoption in practice. It is therefore important to investigate the variability implementation needs of real industrial systems to understand which variability implementation mechanisms work for them.

In this paper, we present our case study experience investigating the software variability challenges and needs of `Eclipse OMR` [3], an industrial open-source software system developed by IBM. `Eclipse OMR` is a set of open-source C++ components that can be used to build language runtimes [3]. `Eclipse OMR`'s variability comes from its support for multiple architectures, and equally important, its support for creating language runtimes for multiple programming languages. Its goal is to allow any language developer to reuse and customize its components to create, for example, their own just-in-time (JIT) compiler or garbage collector.

In a previous position paper, we explained `Eclipse OMR`'s non-typical variability implementation strategy, namely static polymorphism through extensible classes [37] (recapped in Section II-C). When we started working with the team in 2017, we thought that static polymorphism is the main problem that leads to their variability implementation being convoluted and hard to understand. Thus, we investigated the performance impact of switching to the more typical dynamic polymorphism, with the motivation that it may solve some of the faced comprehension issues [38]. While our previous work focused on the trade-offs between static and dynamic polymorphism, we realized that there are still open problems that may require more re-engineering effort. Thus, in this paper, we take a step back to collect and understand all the current challenges in order to develop a more holistic picture for the required variability support.

To collect this information, we interviewed 6 IBM developers who either directly contribute to `Eclipse OMR` or use it as part of their daily work. We identified eight challenges and three constraints that should be taken into account when looking for a new variability design. We then surveyed the literature and existing large open-source software systems to identify the current variability implementation strategies that may resolve these challenges and constraints. We found that many of these strategies do not resolve all challenges or are infeasible, due to the required time investment of the `Eclipse OMR` team. Thus, we took the alternative pragmatic approach of picking challenges that can be solved with local re-engineering changes and gradually addressing them one by one. In this paper, we discuss our implemented solution to one such challenge, namely supporting the variability and extension of data in arrays and enumerations, which has been merged into the `Eclipse OMR` repository [6].

Large open-source industrial systems with software variability are hard to come by, which is why the Linux kernel has attracted much of the research in this area [26], [34], [40], [46]. This paper contributes the variability implementation details, challenges, and constraints of a large industrial system.

Additionally, we resolved one of the challenges with a solution that addresses the problem of lack of extension mechanisms for arrays and enumerations in C++. Such a solution may be adopted by other systems facing the same problem. Finally, our experience working with Eclipse OMR provides valuable insights for similar research-industry collaborations in this area. First, we found that it is necessary to communicate with as many developers from the team as possible to get a more comprehensible picture of the challenges that need to be solved. Our interview study proved invaluable in this aspect. Second, we found that it is important to dig deeper into technical and practical constraints the team may have since the typical state-of-the-art solutions researchers think of may not necessarily be welcome by developers. Instead, they may prefer simpler solutions that, for example, do not entail additional build dependencies or developer training.

II. Background

A. Software Variability

Companies often want to produce similar software, yet for different clients or contexts. For example, the same application might work differently on a desktop as opposed to on an embedded system (e.g., reduced functionality for performance on the latter). Yet, one does not build a system for each environment from scratch. To reduce code duplication and increase reuse of the existing codebase, the idea of *software variability* (i.e., the ability of a software to be changed, configured or customized for a particular context [31]) emerged. By maximizing code sharing among the system's variants, highly variable (or *highly configurable*) software helps companies minimize the time and money spent on software development [9]. Highly configurable software is usually defined in terms of the *features* (i.e., units of functionality) it offers. To support variability, a *variability implementation mechanism* is needed to determine how different features or parts of the code get selected at compile time or run time. The simplest variability implementation mechanism is traditional if statements [9]; we discuss current mechanisms in detail in Section V-A.

B. `Eclipse OMR`

`Eclipse OMR` is a set of open-source C++ components for building language runtimes [3]. A *language runtime* can be seen as the layer of indirection between the language and the operating system which takes care of things such as the application memory layout [8]. In modern languages such as Java and C#, the runtime is a complete environment that the program runs in, on top of the operating system. `Eclipse OMR` allows building this type of runtime. For example, it allows building a JIT compiler and/or a garbage collector that take cares of the behavior of the program while it is running.

In principle, `Eclipse OMR` can be used to build a language runtime for any programming language. For example, assume Alice developed a new language called `Alpha`. As a language designer, Alice needs to decide on the syntax and semantics of the language. She also needs to decide on whether she wants to improve the performance of certain aspects of her language by relying on a JIT compiler. For example, in Java, the JIT compiler compiles the bytecode of a called method into machine code, while performing certain optimizations that provide performance gains especially if a function is called multiple times. `Eclipse OMR` helps Alice with the latter task. For example, to implement a JIT compiler, Alice would write her own code to describe how to translate `Alpha`'s language constructs into `Eclipse OMR`'s Intermediate Language (IL). The `CodeGenerator` class in `Eclipse OMR` is then responsible for translating the generated IL into machine code. Alice can also extend the `CodeGenerator` class to optimize part of this code generation process. Similarly, if her language needs new IL constructs not currently supported (e.g., complex numbers), she can extend the current IL through the `Node` class.

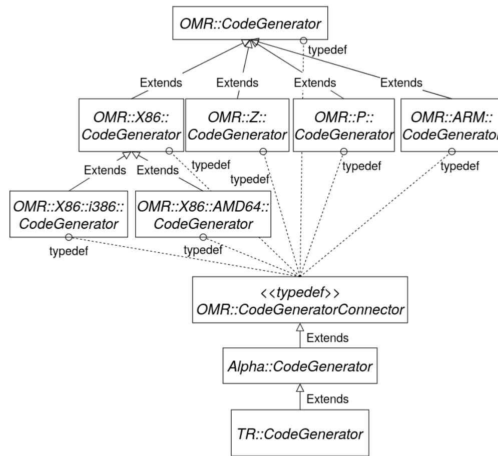
`Eclipse OMR`, open-sourced in 2016, has ~ 1.2 million lines of code and supports various hardware and operating system platforms. `Eclipse OMR` supports x86 (32-bit and 64-bit), ARM (32-bit and 64-bit), Power, and IBM's z/Architecture. At the time of writing this paper, `Eclipse OMR` has already been used for Java (through its largest client, the Eclipse OpenJ9 virtual machine [5]), Ruby, Smalltalk, Python, Lua, and WebAssembly.

C. `Eclipse OMR`'s Current Variability Implementation

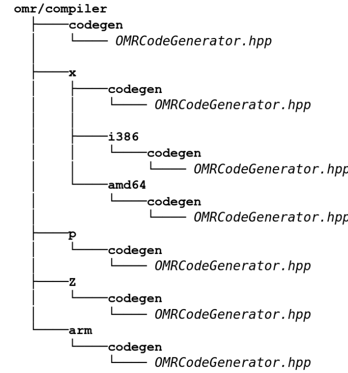
Overall, `Eclipse OMR` has three variability dimensions: (1) features, (2) languages, and (3) architectures. Features include reusable and extensible compilation algorithms such as IL optimization, code generation, and instruction selection.

a) Static polymorphism with extensible classes: Given its domain, `Eclipse OMR` must be highly performant at runtime and have little memory footprint. Thus, `Eclipse OMR` uses *static polymorphism*, which means that the concrete classes of all objects are decided at compile time. This is as opposed to the more adopted dynamic polymorphism where the concrete type of an object and thus the concrete method to call on that object is resolved at runtime. `Eclipse OMR` developers use static polymorphism to build *extensible classes*, a linear inheritance hierarchy of class extensions that are organized in a special way (explained shortly) to allow the compiler to find the most specialized implementation to use. Combined with additional build-time mechanisms such as the C preprocessor, as well as language constructs such as C++ namespaces, extensible classes get rid of dynamic dispatch, which may slow down run-time speed due to the use of virtual tables.

b) Namespaces: `Eclipse OMR` uses the same name for all classes in a given hierarchy and uses namespaces to distinguish them. For example, `OMR::CodeGenerator` class is the most general `CodeGenerator` class, whose implementation is reusable for all architectures. On the other hand, `OMR::X86::AMD64::CodeGenerator` class is designated solely for 64-bit X86 architecture. In addition to avoiding naming collisions, the namespaces encapsulate variability across architectures in a natural way. Further, the nesting of namespaces, as shown in the example of `CodeGenerator` class in Figure 1a, describes a degree of



(a) Example of extensible class hierarchy



(b) Example of directory structure

Fig. 1: Examples of `Eclipse OMR`'s variability implementation mechanisms [37]

specialization. The level ranges from general to specific; a namespace containing the implementation for X86 architecture is more general than the one containing the implementation for AMD64. Before extending any `Eclipse OMR` class, the client must declare a namespace representing their language. In our example, this is the `Alpha` namespace shown in Figure 1a.

c) Directory structure: To organize namespaces, `Eclipse OMR` follows a specific directory structure (shown in Figure 1b). The topmost directory, `omr/compiler/`, contains the `OMR` namespace, which contains the functionality common to all architectures, and usable by any language. The most nested directory (e.g., `omr/compiler/x/amd64/`) contains the specific implementation for 64-bit X86 architecture (`x` in the path stands for X86).

d) Connectors: The above hierarchy means that Alice needs to decide which specific `CodeGenerator` class to extend from the given hierarchy. This means that every time she wants to compile for a different architecture, she needs to change the class in her `extends` clause. Naturally, such "hard-coding" is impractical and not encouraged. To let the client's code work with any architecture, `Eclipse OMR` uses the idea of connector classes. A *connector* is simply a `typedef` of a specific class extension. This `typedef` is present in every class in the hierarchy, as shown by the dashed lines in Figure 1a (e.g., `typedef OMR::ARM::CodeGenerator CodeGeneratorConnector`). That way, Alice does not need to choose which architecture class from the hierarchy to extend and simply always extends the connector, e.g. `CodeGeneratorConnector` in the figure.

e) Include paths: At this point, the reader is probably wondering how the connector class would resolve to the right architecture class. This is where include path prioritization comes in. In the build system, include paths for each architecture are set up such that the most specialized class is always seen first. For example, to compile `CodeGenerator` class for AMD64 (64-bit version of X86) architecture, the include paths passed in the build system would be as follows: `-Iomr/compiler/x/amd64 -Iomr/compiler/x -Iomr/compiler`. Thus, when compiling the `CodeGenerator` class, the build system will first search for a class declaration in `omr/compiler/x/amd64`. If it does not find it there, it will search for the class in `omr/compiler` and so on. At the first class declaration it finds, it will also find the connector `typedef` statement. These `typedef` statements are guarded similar to include guards such that they can only be defined once. Thus, through this include path prioritization, the connector class always resolves to the most architecture-specific class in the hierarchy, which in turn extends all previous more generic classes.

f) Concrete classes: In addition to creating the right class hierarchy, there will of course be instances of the desired class used in other parts of `Eclipse OMR`. However, there is no way to know which client languages will end up using `Eclipse OMR`, which means we need a generic way of referring to a concrete class that is always the most specialized class in the hierarchy. To unify things, `Eclipse OMR` has the `TR` namespace which must extend the language namespace. The `TR` namespace is the final namespace in the hierarchy and the `TR` class is the concrete class from which objects can be instantiated. Thus, OMR developers can directly use that namespace without needing to know in advance what namespace the client is going to choose.

g) self(): Whenever a `TR` object is used, developers expect that it always resolves to the most derived implementation; the resolution must happen at compile time. However, if, for example, a method `foo()` is invoked on a `TR` object inside `OMR::X86::CodeGenerator`, the compiler can only look for `foo()` inside `OMR::X86::CodeGenerator` and its parent classes. However, if this code is being compiled for `OMR::X86::AMD64`, `foo()` could actually be defined in `OMR::X86::AMD64::CodeGenerator` which is further down in the hierarchy. To force the compiler to *always* scan the inheritance hierarchy bottom-up regardless of which class we are in in the hierarchy (i.e., to always find the most derived implementation), `Eclipse OMR` developers use a special method called `self()`. `self()` performs a static cast of

TABLE I: Participant Overview

ID	Role	Years in Company
P1	Senior Compiler Developer	19 years
P2	JIT Compilation Team	2 years
P3	Compiler Developer	5 years
P4	Z Code Generator	5 years
P5	JIT Builder Team	2.5 years
P6	General OpenJ9 intern	8 months

the `this` pointer in a given class hierarchy to the class in the `TR` namespace of that hierarchy. Using `self()`, developers force the compiler to search from the bottom of the hierarchy.

To summarize, Eclipse OMR's compiler component manages variability with the help of extensible classes that leverage static polymorphism. To enable extensible classes to behave in a specific way, Eclipse OMR developers rely on C++ namespaces, compiler include-path prioritization, connector classes, and the `self()` function. This unique approach helps Eclipse OMR stay language-agnostic, allows the clients to extend and reuse a namespace specialization of their choice, as well as organizes variability across architectures using a linear hierarchy to represent a given class.

III. Interview Study Description

Based on our internal discussions with our direct Eclipse OMR collaborators and our previous work with them [37], [38], we knew that the current variability implementation strategy had its shortcomings. However, we only had a limited view of these shortcomings. Thus, to guide any large-scale engineering efforts, we wanted to make sure we hear from more members in the team to have a comprehensive view. Thus, in this paper, we conduct an interview study to answer the following questions: *Are there any challenges related to Eclipse OMR's current variability implementation strategy?* and *Are there any constraints that need to be considered for future changes?* Based on the information gathered from participants, our goal is to come up with a set of challenges that future re-engineering efforts should try to overcome, and the set of constraints that these efforts should take into account.

a) Participant Recruitment: Our target population is developers who directly contribute code to OMR or who work on products that use OMR, such as OpenJ9. We sent invitation emails to 11 candidate developers, and 6 agreed to participate in the study. Five of these interviews were carried in person at IBM Toronto Software Lab, while one was carried over Skype. Table I provides an overview of our participants.

b) Interview Setup and Data Analysis: We followed a semi-structured approach for the interviews, where we had a pre-defined list of questions to guide the interview, but allowed participants to deviate from these questions. We also asked new follow-up questions depending on the direction of the conversation. That said, for all interviews, we made sure to ask participants about challenges they face and technical characteristics of OMR that need to be taken into account for any new design. Five of the interviews lasted between 47min – 1hr, while one lasted 20min. We recorded all interviews, with participants' permission and after university ethics clearance, and transcribed them. We then followed an open coding approach [16] where we read through the whole transcript and assigned a label to different parts of the interview. We did not attempt to categorize things at that point, but simply annotated it with descriptive phrases or labels that summarize the subject of discussion. An example phrase includes "*configuring IDE for different platforms may be a bit tedious*". We then went through all phrases and grouped them into meaningful categories, as well as whether they are describing current design challenges or constraints.

Afterwards, we prepared a presentation to share with the whole Eclipse OMR team through their bi-weekly architecture meeting [4]. This allowed us to verify our interpretations of the interviews and our synthesized results. There were 15 participants in this meeting, several of who were not part of our interviews or any previous discussions, which allowed us to get additional validation and understand which challenges are most important to the team. The presentation lasted around 30 minutes and was followed with almost 45 minutes of discussion. A recording of the meeting is available online [7].

IV. Eclipse OMR Variability Challenges and Constraints

We now describe the challenges and constraints synthesized from our interview study.

A. Challenges

We collect eight challenges associated with the existing variability implementation mechanism, which should ideally be addressed by any new variability implementation mechanism.

CHAL 1. *No standalone end product:* As P3 mentions, one of the biggest challenges when starting with Eclipse OMR is that it is not a standalone product that one can build and run. P4 also discusses this point by saying that *"I think it's hard for people to [get started with OMR], because we don't build any kind of static libraries per se, ..., there's no library that ever gets built [in OMR]. It's kind of meant to be extended and then you build whatever you want; you can statically link it, you can make it a dll and dynamically link it. It's kind of a barrier to entry where people can't envision [how to] use this thing [and] what [exactly] is it?"* Typically, when starting with a system, a developer either adds that project as a dependency and uses its Application Programming Interface (API) in their code (e.g., in the case of a library or a framework) or builds it and runs it first and then decides how to extend its functionality. Instead, to leverage Eclipse OMR, a developer needs to identify which classes to extend and which functions they need to override. To observe the effect of their extensions, a developer needs to run a program written in their target language using the language run-time extension they created using Eclipse OMR. Creating such an extension requires a deep understanding of Eclipse OMR's architecture and how extensible classes work, which is usually beyond the knowledge of a beginner.

CHAL 2. *Unclear extension points:* Given the use of extensible classes, no functions are marked as `virtual` which means that the extension points a new client developer needs to consider are not well defined. For example, certain behavior of the code cache must be specified by the developer as extension points. However, not all these expected extension points are easily identifiable. For example, the function `CodeCacheManager::allocateCodeCacheSegment` is expected to allocate a block of executable memory and then wrap it in a `TR::CodeCacheMemorySegment` instance. However, it has no implementation in the `OMR` namespace and so language developers are required to provide a "sensible" implementation themselves. It is up to the language developer to decide what is "sensible" for their project since there is no starting point implementation in `Eclipse OMR` and there is no documented way of implementing this function. Right now, the developer would typically look at existing `Eclipse OMR` clients, such as OpenJ9, and try to mimic what they do.

CHAL 3. *The use of `self()`:* Perhaps the most brought up topic in the interviews was the use of `self()`. The need to use `self()` to call functions in the same class stems from using extensible classes in `Eclipse OMR`. All participants agreed that this is unintuitive for newcomers since they probably never used something similar in their previous work. The use of `self()` also makes the code ugly and harder to understand. Since forgetting to use `self()` is common, the `Eclipse OMR` team has developed a linter that warns developers if they forget to use it. Given that there is no explicit list of which methods need to be extended, the linter checks that `self()` is used for *all* method calls of the methods in the same class, even if none of the downstream projects currently use this method. This ensures that the correct implementation of the function gets invoked if downstream projects override this function down the line. While this prevents future problems from occurring, developers find the linter warnings frustrating when they know that this function is currently not being overridden anywhere in the hierarchy.

CHAL 4. *Connector classes:* Connectors are another side effect of using static polymorphism through extensible classes. They complicate the design and create another layer of indirection. P4 mentions that they have seen other developers commonly forget to use the connector class on static functions. Since static functions can also be overridden between `Eclipse OMR` and the project they consume, this may lead to incorrect behavior especially with the presence of sub architectures. Thus, the use of connectors adds another layer of complexity and indirection that needs to be taken into account.

CHAL 5. *Comprehension for concrete configurations:* Given that the same class has different architecture versions, it is often not clear which functions are available for a given architecture. Assume a developer is interested in x86 and wants to know which functions are available in the `CodeGenerator` hierarchy. In that case, as a result of the current static polymorphism, the developer needs to resolve all include paths to see a single linear hierarchy for x86. Alternatively, the developer can have a view of the particular architecture, or language/architecture combination, in their IDE. However, several participants pointed out that not all IDEs have an easy support for this since manual configuration of the paths related to the target architecture is needed. To do this, developers typically run a mock build on the target platform, collect all needed include paths and then put them into their IDEs (P4). Once correctly configured, code navigation works as expected (e.g., finding usages of a function) although if a new file gets added to that architecture, the IDE will not automatically include that file in its build path.

CHAL 6. *Tedious and Non-obvious Code Edits:* Given how extensible classes work, adding a new constructor any where in the hierarchy (e.g., in the `X86` class) requires declaring the same constructor in the `TR` namespace. Similarly, adding or changing the constructor of an extensible class in the `OpenJ9` namespace (e.g., adding a parameter) requires changing the constructor of the same class in the (1) `OMR` namespace, (2) `TR` namespace in the `Eclipse OMR` repository, and (3) `TR` namespace in the OpenJ9 repository. Note that the first two changes lie in the `Eclipse OMR` repository. As a developer working on the separate repository of `OpenJ9`, it is not directly obvious that a change in `Eclipse OMR` is also needed for local changes.

CHAL 7. *The Java legacy:* Historically, `Eclipse OMR` grew out of IBM's previous Java J9 just-in-time compiler, Testarossa. As a consequence, a lot of the current design and supported operations revolve around the Java world. While `Eclipse OMR` is supposed to be a language-independent framework for developing language run-times, there are still functions in OMR that are specific to Java. The code in these functions are often guarded with the macro `J9_PROJECT_SPECIFIC`. A search for this macro reveals over 650 cases of code that is only enabled in OpenJ9 builds. One example of such a Java-only function that is present in an extensible class is `OMR::Power::TreeEvaluator::reverseLoadEvaluator`. The function itself is present in OMR, but its implementation code is guarded by `J9_PROJECT_SPECIFIC`. These guards cause *kitchen sink problem*, which is the phenomenon that OMR will end up containing many language-specific functions in common code that is exposed to all OMR users.

CHAL 8. *Missing extension mechanisms:* One of the current limitations of extensible classes, and any native C++ inheritance-based mechanism, is that they do not support `enum`, `union`, or array extensions. As a workaround, `Eclipse OMR` currently relies on various macro and include tricks. Let us take the example of *opcodes*, which are `Eclipse OMR`'s IL operations that get translated to machine code. Figure 2 shows an excerpt of the `ILOpCodes` enumeration declared in `ILOpCodes.hpp`. The original declaration in this header file actually only includes another header file (`ILOpCodesEnum.hpp`), which in turn includes `OMRILOpCodesEnum.hpp` which then contains a long list of the enumeration values. In other words, `Eclipse OMR` developers currently use file inclusion as a mechanism of creating some hierarchy of the different opcodes, where files

```
enum ILOpCodes {
   //Originally #include "il/ILOpCodesEnum.hpp"
   FirstOMROp,
   BadILOp = 0,  // illegal op hopefully help with ...
   aconst,   // load address constant ...
   iconst,   // load integer constant ...
   ...
};
```

Fig. 2: ILOpCodes enumeration declaration in ILOpCodes.hpp

are included in a specific order as needed. For illustration in the figure, we already perform these includes and show some of the `enum` values that would get included. Each of the enumerated values corresponds to an opcode. Each opcode has some properties, such as its name and data type. These properties are declared in a separate file as an array of structs, shown in Figure 3. Similar to the previous file, there is a chain of includes that leads to including the properties shown in the listing. The order of opcodes in the enumeration in Figure 2 must match the order of properties in Figure 3. Figure 3 shows the first array entry (index 0) which corresponds to the properties of the first enumeration value of 0, `BadILOp`.

This design causes several issues that were discussed by three participants. First, having to match declaration orders in two separate data structures that are spread across multiple files is tedious and error prone. Another issue is that any changes in these opcodes or their order have a downstream effect on other language developers. For example, in OpenJ9, the opcodes are declared in a similar way: first, the file containing the OMR opcodes is included and then additional entries (whether opcode enum constants or a property entry in the array) are included afterwards. Breaking any of the order assumptions can result in reading wrong/bad data. Since the order violation does not result in build breakage, resulting problems are often harder to catch. `Eclipse OMR` developers try to mitigate this by being very diligent in reviewing any changes to these declarations. Reviewing just a few lines of code can take several hours because of the manual cross-referencing required. Finally, the fact that a single declaration (whether an enumeration or a table) spans multiple files contributes to code spread, where developers may easily forget to edit one of the files that affect the final composition of this declaration. Interestingly, there is a large warning note at the top of `OMRILOpCodesEnum.hpp` that lists 13 other files to examine in order to add an opcode or change opcode order.

Summary: To summarize, `Eclipse OMR` developers experience the following common challenges: (1) the current variability implementation mechanism adds multiple layers of complexity that hinder code comprehension, (2) extensible classes cause code spread within the codebase itself. The combination of code spread across multiple repositories with the tight integration between downstream projects and `Eclipse OMR` result in the need for non-obvious code edits; (3) while the current OMR code base is meant to be generic enough to support many languages, it still contains Java-specific code as a result of the original code base; and finally, (4) since there is currently no built-in C++ enum or array extension mechanism, `Eclipse OMR` developers use workarounds through including multiple files in an enum or table declaration, and have to manually keep track of the order of the enum constants in different files that must match each other. We deduce that any new design should prioritize simplicity and usability of the codebase. It should also ideally ensure a cleaner separation of concerns or, pragmatically, more streamlined consistency checks and editing capabilities across all related code and repositories, including support for extending enums and tables.

```
OMR::OpCodeProperties OMR::ILOpCode::_opCodeProperties[] =
{
//#include "il/ILOpCodeProperties.hpp"
   {
   /* .opcode                 = */ TR::BadILOp,
   /* .name                   = */ "BadILOp",
   /* .dataType               = */ TR::NoType,
   ...
   },
};
```

Fig. 3: Properties corresponding to the opcodes from Fig. 2, declared in OMRILOps.cpp

B. Constraints

Participants also mentioned several constraints that must be taken into for any design `Eclipse OMR` follows.

*a) **CONST 1.** Strongly connected components:* One of the variability implementation mechanisms that `Eclipse OMR` developers previously explored was a template-based solution, specifically the Curiously Recurring Template Pattern (CRTP) [11]. Templates can potentially solve problems such as the use of `self()`. However, a C++ template requires all definitions to be in header files which means that whenever the template header file is included, the compiler needs to compile all the implementation code rather than simply treat these methods as external declarations. This results in larger binaries and longer build times. Due to these reasons, the `Eclipse OMR` team moved away from using any templates in the project, and would also like to avoid using them (or any other design that results in the same strongly connected component issue) again in any future design.

*b) **CONST 2.** Impact on downstream projects:* `Eclipse OMR` has many client projects, the largest of which is OpenJ9. Any changes in the `Eclipse OMR` codebase must eventually be integrated into the client projects as well, and must avoid breaking any client code (e.g., See IV-A, Challenge 8 where the order of opcodes in OpenJ9 must correspond to the order of opcodes in `Eclipse OMR`). Since `Eclipse OMR` strives to support many languages, introducing any changes to the existing variability mechanism or developing a new one should have minimal impact on downstream projects (i.e., specific language runtimes), such as OpenJ9.

*c) **CONST 3.** C++ Version:* `Eclipse OMR` supports various environments, and the level of C++ compiler support on these platforms differs. For example, on the AIX operating system, the XL C/C++ compiler performs more optimizations to the native code it produces resulting in faster run-time performance than the code produced by its GCC counterpart on the same operating system. Since run-time performance is essential in `Eclipse OMR`, the team uses the XL C/C++

compiler for AIX. However, while the C++ language has already evolved into C++20, the version of XL C/C++ used for AIX builds does not even support all features of C++11. This means that the `Eclipse OMR` team is stuck coding to the lowest common denominator to avoid large discrepancies between the various platforms it supports. Thus, while there may be new useful language features, such as `constexpr` or initializer lists for non-aggregate types (e.g., initializer list for `std::vector`), such features cannot be used since they are not supported by the compilers of all the target environments.

V. Existing Variability Implementation Strategies

We now explore the literature and existing case studies to identify variability implementation mechanisms that may overcome the above challenges.

A. Strategies in the Literature

The *"Feature-Oriented Software Product Lines"* book by Apel et al. [9] provides a comprehensive survey of existing variability implementation strategies. Thus, we use it as a reference to explore known variability implementation strategies in the literature. In general, variability implementation strategies differ in their nature: they can be tool-based or language-based, as well as fine-grained (e.g., `if` statements) or coarse-grained (e.g., components).

The C preprocessor, specifically using `#ifdef`, is a well-known tool-based, fine-grained mechanism for conditional compilation which has been extensively used and studied, despite being criticized for reducing readability and comprehension [32], [39], [46], [50]. Similarly, build systems, e.g., `Make`, may be used to conditionally compile specific files. At a low level, OMR already uses `#ifdefs` for include-like guards for defining connectors on each architecture (see Section II-C) and CMake to manage which features (e.g., garbage collector) get compiled. However, using `#ifdefs` and the build system alone are not enough to cater to `Eclipse OMR`'s variability, such as offering external extension points, which is why extensible classes are used.

Software variability can also be achieved through simple design patterns (e.g., strategy pattern [24]) or through a structured framework design (similar to the Android platform or Eclipse plugins [2], [25]). A framework usually provides explicit extension points. In a sense, `Eclipse OMR` is already a white-box framework, which means that client developers need to be familiar with the internals of the classes and methods. However, due to legacy baggage and the complex nature of its extensible classes, the current extension points are not obvious.

Feature-oriented programming (FOP) is a language-based coarse-grained variability implementation mechanism [9]. It encapsulates each feature in a single *feature module* and uses the notion of *refinements* that allow to extend classes without changing existing implementations. FOP naturally allows for direct feature traceability due to feature modularity, and thus, could clarify OMR's extension points. However, it has been used mostly in academia and little in the industry, possibly due to language support and tooling requirements. Despite FOP's potential benefits (e.g., less code spread and easier code navigation), it would be infeasible to re-engineer OMR to apply FOP since that requires (1) developer time investment because OMR expert knowledge is needed, (2) new tools for developers to learn, and (3) new tools that must be installed on the build servers and developers' local environments.

Summary: On one hand, some variability mechanisms address specific low-level needs and cannot be used off the shelf to improve `Eclipse OMR`'s overall variability implementation. On the other hand, performing a migration, especially to coarse-grained mechanisms such as feature-oriented programming, requires extensive preplanning and investment that the `Eclipse OMR` team cannot invest at the moment.

B. Related Case Studies

Many companies have reengineered their software and hardware into product lines. For example, the Software Product Line Conference Hall of Fame contains more than 20 organizations that adopted SPLs for their projects [47]. There are also numerous industry case studies in various domains such as temperature monitoring [30], biometric systems [44] UML diagramming [20], and geographic information systems [10], [14], [15], [33], [36], [42], [54].

Most industry case studies adopted product-line engineering in an *extractive manner*, i.e., creating an SPL out of existing software variants (end products), while our case study is about re-engineering an *existing* variability implementation. For instance, while Chae et al. [17]'s domain is close to ours (MLPolyR compilers), they re-engineered different existing compilers into an SPL. Another extractive case is Polyglot compiler, which performs source-to-source compilation to Java or Java bytecode [41]. Finally, some studies survey current state of variability management tools without going into concrete implementation mechanisms for each case [13], some lack specific technical details (probably due to confidentiality reasons) [49], [54], while others are working with simpler and smaller projects. In fact, ArgoUML's original implementation consists of about 120 KLOC, whereas OMR has ~1.2 MLOC [20]. Microsoft Common Language Runtime (CLR), part of the .NET framework, is in the same domain of language runtimes as OMR [1]. However, while CLR is already a language runtime, OMR is rather a collection of components for *building runtimes*. Moreover, to the best of our knowledge, there are no CLR case studies that could be compared to `Eclipse OMR` in terms of variability design.

Summary: While many existing industry and academic case studies extract an SPL from existing set of assets, we are trying to re-engineer an already existing configurable system. Unlike most other studies, `Eclipse OMR` is a large-scale project with multiple variability dimensions (language, architecture, and extensible features such as code generation). In addition, while many industrial SPLs do not share their project's source code and do not provide precise technical information, `Eclipse OMR` is open source which allows us to share more technical details and insights, which may be useful for others in a similar situation.

VI. Addressing The Enum Problem

After looking at existing techniques in the previous section, we decided to take an incremental approach where we address one challenge at a time, while respecting the identified constraints. Such step-by-step approach has a minimal impact on the main ongoing development of the project and also helps the researchers on the project stay up to date with IBM developers through continuous code reviews and pull requests. Based on discussions with the team, we agreed to start with Challenge 8.

A. *Discussed Alternatives*

Based on searching for systems with similar needs and on discussions with the `Eclipse OMR` team, we considered three potential alternative solutions for challenge 8.

A DSL Approach, LLVM TableGen: LLVM is a collection of reusable and modular compiler tools for building front-end and back-end compilers [35]. Its back-end classes reason about architecture-specific information in a generic way; for example, to perform a *register allocation*, a back-end class has to know what registers are available on the target architecture. This architecture-specific information is obtained through "target description" (`.td`) files, which are used to generate header files with the necessary information about an architecture. These `.td` files correspond to a domain-specific language (DSL) known as *TableGen*. We considered implementing a similar DSL for `Eclipse OMR` to encode opcode information from the scattered enums and arrays. For example, a single DSL class may represent an opcode, and the class could be extensible so that more fields, each representing an opcode property, could be added (or overriden) by the client. The DSL approach would allow for automated derivation of header files from a single definition containing built-in (OMR) and client (language-specific) opcodes.

A Simpler DSL-like Approach: A simpler DSL-like alternative to TableGen is to store opcode information in a file format, such as JSON. The JSON file(s) would contain the relevant opcode information that is used to generate C++ headers. For example, `Eclipse OMR` would have one central JSON file that stores all OMR opcodes and their properties, and each language client would have another JSON file that stores additional language-specific opcodes. During compilation, a Python script can process all the JSON files to generate a corresponding set of C++ headers for the different classes.

SpiderMonkey and macro expansion: SpiderMonkey is a JavaScript engine maintained by Mozilla, comprised of components such as interpreter, JIT compiler, and garbage collector [19]. SpiderMonkey has its own set of bytecodes (opcodes). Unlike OMR that manages opcode information in a "distributed" manner, SpiderMonkey specifies all opcodes in a single `opcodes.h` header file. The header file contains all opcode information such as value, token, length, and description. The opcode information is contained in *C++ macros*. The `opcodes.h` file can be consumed anywhere where certain opcode information is required (e.g., declaring an array with all opcodes' lengths) by simply including the header file and defining a custom macro function that will fetch a desired opcode property.

```
#define FOR_EACH_OPCODE(MACRO) \
  MACRO(\
    /* .enumValue = */  iconst,  \
    /* .name      = */  "iconst", \
    /* .dataType  = */  TR::Int32, \
    ...
  ) \
  MACRO(\
    /* .enumValue = */  fconst,  \
    /* .name      = */  "fconst", \
    /* .dataType  = */  TR::Float, \
    ...
  ) \
  ...
```

Fig. 4: Excerpt from our implemented OMROpcodes.hpp

Discussion of Alternatives: All the above solutions will have minimal effect on downstream projects such as OpenJ9 and will allow incremental migration where we can start with `Eclipse OMR` and add later support for downstream projects. Due to its expressiveness and being a well-engineered solution, our initial intuition was that a DSL solution would be the one most welcomed by `Eclipse OMR` developers; or at least the simpler JSON/Python alternative. However, the team indicated that they prefer not to install any additional dependencies, whether these dependencies are new language parsers or even Python. They also did not want to have to learn a new language. Based on the practical considerations above, the `Eclipse OMR` team preferred the SpiderMonkey macro-based solution due its canonical form (macros are native to C/C++ compilers) and minimalism (no additional dependencies are required).

B. *Our Implemented Solution*

We now discuss the implementation details of the adopted SpiderMonkey-inspired macro-based solution. The core idea behind the implemented solution is the centralization of all opcode information in a single header file (*OMROpcodes.hpp*). The header file contains exactly one macro, namely `FOR_EACH_OPCODE` shown in Figure 4. This macro takes one argument `MACRO`, which corresponds to multiple entries with the same name inside the `FOR_EACH_OPCODE` macro definition. Each of these entries, marked by `MACRO`, represent one opcode with arguments representing its various properties. In other words, each argument in a `MACRO` corresponds to an opcode property, such as its name or return type. At a high level, `FOR_EACH_OPCODE` invokes a passed in `MACRO` function for every opcode in the list, returning a required piece of information about all opcodes.

Let us take the example in Figure 2 and see how this file would change to generate the enum definition directly from the central information in *OMROpcodes.hpp*. Figure 5 shows how the file now looks like after our implementation. The file includes *OMROpcodes.hpp* and defines a macro, called `GET_ENUM_VALUE` in this case, which indicates which values from an opcode definition are relevant to the current file. As we only care about the `enumValue` of an opcode, which is the first argument of a `MACRO` definition in *OMROpcodes.hpp*,

we can ignore the rest of arguments using *variadic argument* (ellipses as the second argument). GET_ENUM_VALUE then indicates that given a list of arguments (properties of an opcode), the return value should be the enumValue. Since there are numerous opcodes (and thus enumValues), we should also place a comma after each enumValue. Essentially, the macro simply returns the enumValue of the associated opcode. After GET_ENUM_VALUE is defined, we call the already defined FOR_EACH_OPCODE macro function which after preprocessing will expand to a list of enumValues of all opcodes. After preprocessing the code, the resulting file will look exactly like that in Figure 2. The difference now is that the developer will never need to directly edit this file in order to add new opcodes. In the same manner, one could fetch other properties as desired and generate the file in Figure 3.

We introduced our solution as a pull request in the Eclipse OMR repository [6]. All our changes were reviewed and extensively discussed with 2 members of the Eclipse OMR team and were eventually accepted and merged. Our solution covers the entire codebase of the compiler component. In total, we introduced one central header file (*OMROpcodes.hpp*) with 735 opcodes, each containing 14 properties. We replaced the content of 12 header files with a single macro in each (similar to the example in Figure 5). Now, the developers do not have to keep track of the numerous header files because the consistency is automatically enforced by a single macro containing all opcode information in *OMROpcodes.hpp*.

```
#include "il/OMROpcodes.hpp"

enum ILOpCodes {
#define GET_ENUM_VALUE(enumValue, ...) enumValue,
  FOR_EACH_OPCODE(GET_ENUM_VALUE)
#undef GET_ENUM_VALUE
};
```

Fig. 5: The re-designed OMRILOpCodesEnum.hpp file, which will eventually produce enum values from Figure 2.

VII. Lessons Learned

While some of the challenges and constraints we discussed may be specific to Eclipse OMR, other systems may face similar problems and can benefit from the related discussions. Additionally, our reported experience working with Eclipse OMR has transferable lessons as follows.

a) Understand the bigger picture: When we started this collaboration in 2017, we focused on the idea of extensible classes for a long time. Thinking that concepts such as self() are the root of all evil when it comes to difficulties working with the code, we focused on exploring the transition to dynamic polymorphism [38]. It took us a while to step back and realize that this is not the only problem and that dynamic polymorphism does not solve all problems (e.g., code spread, enumeration support, or lack of extension points). While we had regular discussions with our direct collaborators, we realized that a bigger conversation with the whole team is needed. This is why we conducted the interview study, which proved invaluable to understanding what is really needed. Additionally, participating in the Eclipse OMR architecture meeting allowed the research team to reach more Eclipse OMR developers and have a broader discussion of our findings.

b) Do not underestimate practical constraints: As researchers, we thought that the "latest and greatest" would be the most welcome option for the various requirements. However, the interviews and discussions with the team revealed that things like the C++ version (Section IV-B) or not wanting to install new dependencies on build servers (Section VI) are practical constraints that may rule out such options.

c) Identify feasible changes: We realized that any complete rehaul of the system to use coarse-grained strategies (e.g., feature-oriented programming) are almost impossible to do without the direct involvement of team members who have deep knowledge of the semantics and purpose of each piece of the code base. Obviously, such time investment is impractical in parallel to new features being developed. We found that from the research team side, incremental changes with continuous code review feedback was more feasible.

VIII. Conclusion

This paper described the interview study we conducted with Eclipse OMR developers to gather 8 challenges related to the current variability implementation mechanism. When considering these challenges, along with technical constraints and practical considerations, such as build dependencies or developer's time, we realized that none of the known variability implementation mechanisms can be used off the shelf. Instead, a combination of different mechanisms and local solutions seemed like the right way to go. To this end, we picked one of the challenges (extensibility and consistency of data structures such as enums and arrays) and implemented a simple solution for it. Our experience demonstrates that there are often practical considerations to take into account that prevent more elegant/elaborate solutions and favor simpler ones. Unlike other similar industry case studies that do not provide enough technical information, we hope that technical details provided in our case study will be helpful for other similar industry or research case studies.

Acknowledgments

This work is funded by the IBM Center for Advanced Studies (CAS). Thanks to all the interviewed developers. Thanks to X. Liang, R. Young, and D. Maier for their feedback.

References

[1] Common Language Runtime (CLR) overview. https://docs.microsoft.com/en-us/dotnet/standard/clr.

[2] Eclipse Marketplace. https://marketplace.eclipse.org/.

[3] Eclipse OMR. https://github.com/eclipse/omr.

[4] Eclipse OMR Architecture Meeting. https://github.com/eclipse/omr/issues/3607.

[5] Eclipse OpenJ9. https://github.com/eclipse/openj9.

[6] Our merged Eclipse OMR pull request for challenge 8. https://github.com/eclipse/omr/pull/4915.

[7] Recording of eclipse omr compiler architecture meeting. https://www.youtube.com/watch?v=F7FIE1QIUAE&t=1403s.

[8] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[9] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-oriented Software Product Lines: Concepts and Implementation*. Springer Science & Business Media, 2013.

[10] J. Bayer, J.-F. Girard, M. Würthner, J.-M. DeBaud, and M. Apel. Transitioning legacy assets to a product line architecture. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering — ESEC/FSE '99*, pages 446–463. Springer Berlin Heidelberg, 1999.

[11] E. Bendersky. The Curiously Recurring Template Pattern in C++, 2011. http://eli.thegreenplace.net/2011/05/17/the-curiously-recurring-template-pattern-in-c/.

[12] T. Berger, R.-H. Pfeiffer, R. Tartler, S. Dienst, K. Czarnecki, A. Wąsowski, and S. She. Variability mechanisms in software ecosystems. *Information and Software Technology*, 56(11):1520–1535, 2014.

[13] T. Berger, J.-P. Steghöfer, T. Ziadi, J. Robin, and J. Martinez. The state of adoption and the challenges of systematic variability management in industry. *Empirical Software Engineering*, 01 2019.

[14] D. Brugali and N. Hochgeschwender. Software product line engineering for robotic perception systems. *International Journal of Semantic Computing*, 12(01):89–107, 2018.

[15] A. Buccella, A. Cechich, M. Pol'la, and M. Arias. Software product line reengineering: A case study on the geographic domain. *Journal of Computer Science and Technology*, 16:14–28, 2016.

[16] P. Burnard. A method of analysing interview transcripts in qualitative research. *Nurse education today*, 11(6):461–466, 1991.

[17] W. Chae and M. Blume. Building a family of compilers. In *2008 12th International Software Product Line Conference*, pages 307–316, 2008.

[18] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[19] M. contributors. Spidermonkey internals, nov 2019.

[20] M. V. Couto, M. T. Valente, and E. Figueiredo. Extracting software product lines: A case study using conditional compilation. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 191–200, 2011.

[21] K. Czarnecki. Overview of generative software development. In J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, editors, *Unconventional Programming Paradigms*, pages 326–341. Springer Berlin Heidelberg, 2005.

[22] M. Fowler. Featuretoggle. https://martinfowler.com/bliki/FeatureToggle.html.

[23] N. Fußberger, B. Zhang, and M. Becker. A deep dive into android's variability realizations. In *Proc. of the 21st International Systems and Software Product Line Conference - Volume A*, SPLC '17, pages 69–78. ACM, 2017.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[25] Google, Inc. Android. https://www.android.com/.

[26] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, and S. Apel. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering*, 21(2):449–482, Apr 2016.

[27] S. E. Institute. Software Product Lines Overview, 2017.

[28] M. Jaring and J. Bosch. Representing variability in software product lines: A case study. In G. J. Chastek, editor, *Software Product Lines*, pages 15–36. Springer Berlin Heidelberg, 2002.

[29] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, pages 220–242. Springer Berlin Heidelberg, 1997.

[30] E. Kuiter, J. Krüger, S. Krieter, T. Leich, and G. Saake. Getting rid of clone-and-own: Moving to a software product line for temperature monitoring. In *Proc. of the 22nd International Systems and Software Product Line Conference - Volume 1*, SPLC '18, pages 179–189. Association for Computing Machinery, 2018.

[31] W. Lab. Software variability.

[32] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proc. of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 105–114. ACM, 2010.

[33] J. Liebig, S. Apel, C. Lengauer, and T. Leich. Robbydbms: A case study on hardware/software product line engineering. In *Proc. of the First International Workshop on Feature-Oriented Software Development*, FOSD '09, pages 63–68. Association for Computing Machinery, 2009.

[34] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 81–91. ACM, 2013.

[35] LLVM. The LLVM Compiler Infrastructure. http://llvm.org/.

[36] J. Martinez, X. Tërnava, and T. Ziadi. Software product line extraction from variability-rich systems: The robocode case study. In *Proc. of the 22nd International Systems and Software Product Line Conference - Volume 1*, SPLC '18, pages 132–142. Association for Computing Machinery, 2018.

[37] S. A. Masri, N. U. Bhuiyan, S. Nadi, and M. Gaudet. Software variability through c++ static polymorphism: A case study of challenges and open problems in eclipse omr. In *Proc. of the 27th Annual International Conference on Computer Science and Software Engineering*, CASCON '17, pages 285–291. IBM Corp., 2017.

[38] S. A. Masri, S. Nadi, M. Gaudet, X. Liang, and R. W. Young. Using static analysis to support variability implementation decisions in c++. In *Proc. of the 22Nd International Systems and Software Product Line Conference - Volume 1*, SPLC '18, pages 236–245. ACM, 2018.

[39] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. The love/hate relationship with the c preprocessor: An interview study. In *Proc. of the 29th European Conference on Object-Oriented Programming (ECOOP '15)*, pages 999–1022, 2015.

[40] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *Proc. of the 36th International Conference on Software Engineering (ICSE '14)*, pages 140 –151, 2014.

[41] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In G. Hedin, editor, *Compiler Construction*, pages 138–152. Springer Berlin Heidelberg, 2003.

[42] U. Pettersson and S. Jarzabek. Industrial experience with building a web portal product line using a lightweight, reactive approach. *SIGSOFT Softw. Eng. Notes*, 30(5):326–335, Sept. 2005.

[43] C. Prehofer. Feature-oriented programming: A fresh look at objects. In M. Akşit and S. Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, pages 419–443. Springer Berlin Heidelberg, 1997.

[44] L. Rincón, E. Muñoz, J. Martinez, M. Pabón, and G. Álvarez. Extractive spl adoption applied into a small software company. In *2016 XLII Latin American Computing Conference (CLEI)*, pages 1–8, 2016.

[45] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In J. Bosch and J. Lee, editors, *Software Product Lines: Going Beyond*, pages 77–91. Springer Berlin Heidelberg, 2010.

[46] J. Sincero, R. Tartler, D. Lohmann, and W. Schröder-Preikschat. Efficient extraction and analysis of preprocessor-based variability. In *Proc. of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 33–42. ACM, 2010.

[47] S. P. L. C. (SPLC). Hall of Fame - SPLC. https://splc.net/.

[48] M. Svahnberg, J. Van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software: Practice and experience*, 35(8):705–754, 2005.

[49] A. Tang, W. Couwenberg, E. Scheppink, N. A. de Burgh, S. Deelstra, and H. van Vliet. Spl migration tensions: An industry experience. In *Proc. of the 2010 Workshop on Knowledge-Oriented Product Line Engineering*, KOPLE '10. Association for Computing Machinery, 2010.

[50] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem. In *Proc. of the sixth conference on Computer systems*, pages 47–60, 2011.

[51] 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.

[52] F. J. Van der Linden, K. Schmid, and E. Rommes. *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Business Media, 2007.

[53] B. Zhang, S. Duszynski, and M. Becker. Variability mechanisms and lessons learned in practice. In *Proc. of the 1st International Workshop on Variability and Complexity in Software Design*, pages 14–20. ACM, 2016.

[54] G. Zhang, L. Shen, X. Peng, Z. Xing, and W. Zhao. Incremental and iterative reengineering towards software product line: An industrial case study. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 418–427, 2011.