Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Simulating Variant Drift in Product-Line Variants Using AST-Based Refactoring

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von:	Sebastian Alexander Wilke
geboren am:	12.08.1995
geboren in:	Landsberg am Lech
Gutachter/innen:	Prof. Dr. Lars Grunske
	Prof. Dr. Timo Kehrer

eingereicht am: .... verteidigt am: ....

Abstract

Clone-and-own research focuses on assisting developers in managing variant-rich systems through better tools and automation. The shortage of publicly available clone-and-own projects led researchers to use variants generated from software product lines instead. In contrast to clone-and-own variants, product line variants lack variant drift. This is a problem, as research results might be biased and not generalizable. In this study, we investigate the feasibility to simulate variant drift in product line variants generated with VEVOS, by using Clang-tidy to apply AST-based refactoring. We found that there exists a variety of challenges to be tackled before the mentioned approach might be considered feasible. The rationales have to do with technical challenges of automated refactoring, as well as restrictions imposed by the architecture of VEVOS.

Contents

1 Introduction

Software is everywhere. It is part of a multitude of different products being sold today. The rapid developments in the industry are accompanied by moves towards more business-centric approaches. The necessity to provide mass customization in order to satisfy customer requirements, as well as staying competitive with other companies have demanded for new ways of developing software.
Software product line engineering is a strategic approach towards developing software which impacts business, organization and technology alike [Van der Linden et al., 2007]. At its core it is a platform encompassing every reusable asset (e.g. requirements or architecture) used throughout the development life cycle. While the use of this approach heavily promotes reusability, thereby lowering cost and time to market, the introduction involves high-upfront investments [Pohl et al., 2005]. That is because before new products (i.e., variants) can be derived from a common code base, the overall set of features and possible ways to combine them have to be specified and documented. The other predominantly used practice besides software product line engineering is known as clone-and-own. It works by copying (cloning) and modifying (owning) existing variants to generate new ones. Despite its short-term advantages like decrease of time to market, it leads to higher maintenance costs in the long run.
A recent line of research set out to explore the continuum between clone-and-own and software product lines [Kehrer et al., 2021]. This line of research usually focuses on better automation through techniques like variability mining [Fischer et al., 2014] and feature trace recording [Bittner et al., 2021]. To counteract the shortage of publicly available clone-and-own projects which could be used as experimental subjects [Schultheiß et al., 2020], Schultheiß et al. introduced the tool VEVOS, which enables researchers to generate benchmarks and simulate the evolution of cloned variants. Using VEVOS in combination with techniques based on document patching, the potential to automate the synchronization of clone-and-own variants has been researched. By using lightweight domain knowledge about which features are affected by a change and which variants implement affected features, variants can be synchronized with an accuracy of up to 93% [Schultheiß et al., 2022b].
However, while VEVOS is able to generate benchmarks for clone-and-own research, these benchmarks lack variant drift. Variant drift is the introduction of unintentional divergences of semantically equivalent software fragments. As most real variants today evolve independently, they inevitably drift away from each other over time. To provide more realistic variants, we are thus interested in simulating such variant drift too.

In this work, we investigate the feasibility to introduce variant drift in variants simulated with VEVOS. We do so by applying refactoring operations based on the abstract syntax tree (AST) of variants generated with VEVOS. After the necessary background knowledge was introduced in Section 2, we take a look at the tool in charge of performing the refactoring, called clang-tidy, in Section 3. Research goals introduced in Section 4.1 guide our exploration on the feasibility and challenges of such approach. Results are then discussed in Section 5.

2 Background

2.1 Software Reuse

Software reuse is the process of creating software systems from existing software rather than building them from scratch [Krueger, 1992]. While the idea exists since the beginning of software development, many of the methodologies and tools in use today emerged since the 1970s. There is a reason for that - the software crisis.

> *"[...] machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming had become an equally gigantic problem."*
>
> *Edsger W. Dijkstra [Dijkstra, 1972]*

The term *software crisis* originates from the first NATO Software Engineering Conference in Garmisch-Partenkirchen, Germany. It alludes to the fact that computers have become more and more performant, yet previous techniques could not keep up with this rapid progress. Nowadays, not only source code is reused but any reusable asset - be it requirements, design or documentation.
The advantages of reuse are obvious - something that can be reused reduces time-to-market and therefore costs; tremendously important characteristics for any company competing in today's software market.

There exist two commonly used paradigms for deriving new variants (i.e., products) in use today - Software Product Line Engineering (cf. Sec. 2.1.1) and Clone-and-Own Development (cf. Sec. 2.1.3). Both allow for mass customization; that is the large-scale production of goods tailored to individual customers' needs [Pohl et al., 2005] and requirements. Ideally, we would like the best of both (cf. Sec. 2.2). How they work and a weighing of the pros and cons of each is illustrated in the following subsections.

2.1.1 Software Product Line Engineering

Software product line (SPL) engineering is a paradigm for developing software applications using platforms and mass customization [Pohl et al., 2005]. In contrast to creating variants ad hoc like with clone-and-own, this approach takes an inherently different path and view of business. Its reuse strategy is based on the insight that most companies specialize within a particular problem domain. Furthermore, the software systems developed in that domains are not new. Rather they are new variants with ever increasing functionality and improvements; trying to keep up with customers needs, but also competition.

Software development using this approach is typically split into two consecutive phases: Domain Engineering and Application Engineering (see Fig. 1). One way to think about them is development *for reuse* and development *with reuse* [Van der Linden et al., 2007].

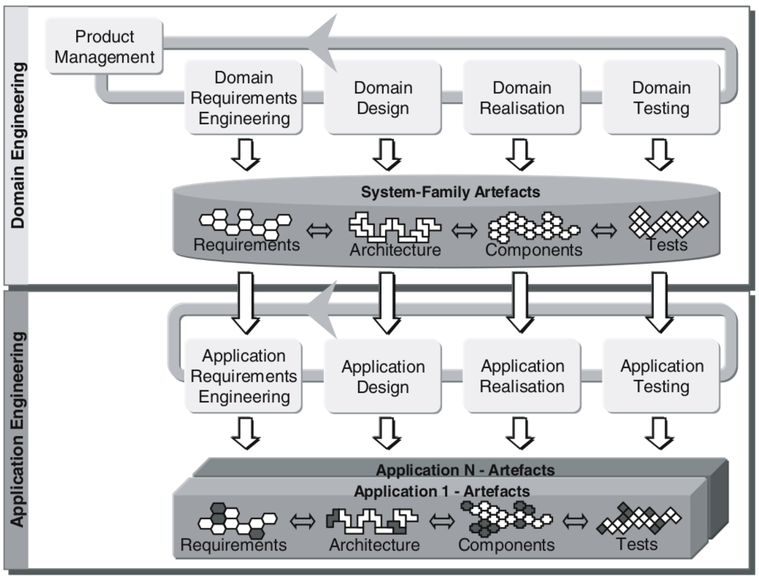


Figure 1: The two-life-cycle model of software product line engineering. Taken from [Van der Linden et al., 2007].

Domain Engineering. Software product line engineering begins with domain engineering. In this phase the reusable platform is defined. Compared to other reuse approaches, the platform encompasses every reusable artifact (e.g. requirements, design and tests) used throughout the software development [Van der Linden et al., 2007]. The platform thereby contains both commonality (i.e., the parts that are shared between different products) and variability (i.e., the parts that distinguish different products). This phase is crucial, as it explicitly defines the set of applications to be generated by the product line, their constraints and feasibility. Typically, a *feature model* describes the set of features[1] and their valid combinations (a.k.a. configurations) [Schultheiß et al., 2022a]. Employing traceability management allows reasoning, whether the properties of the SPL are reflected in a given product and linking between artifacts [Lago et al., 2009].

Application Engineering. Following domain engineering is application engineering. This phase is responsible for deriving (e.g. defining and developing) the product line variants. This is done by binding the variability according to the application needs from requirements over architecture, to components, and test cases [Pohl et al., 2005]. Specifically, new products are generated by gathering requirements, categorizing them

[1]Prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems [Kang et al., 1990].
Unit of functionality that satisfies a requirement, represents a design decision, and provides a potential configuration option [Apel et al., 2011].

as commonality or variability and instantiating artifacts [Van der Linden et al., 2007]. According to Van der Linden et al. up to 90% of the product may be available from reuse.

The cost of developing single systems and product lines is compared in Figure 2. Accumulated effort increases with the number of different systems (resp. variants); the rate of which differs significantly though. The break-even point - the point where costs of deploying single systems and product lines intersect - is reached after about three variants [Van der Linden et al., 2007].

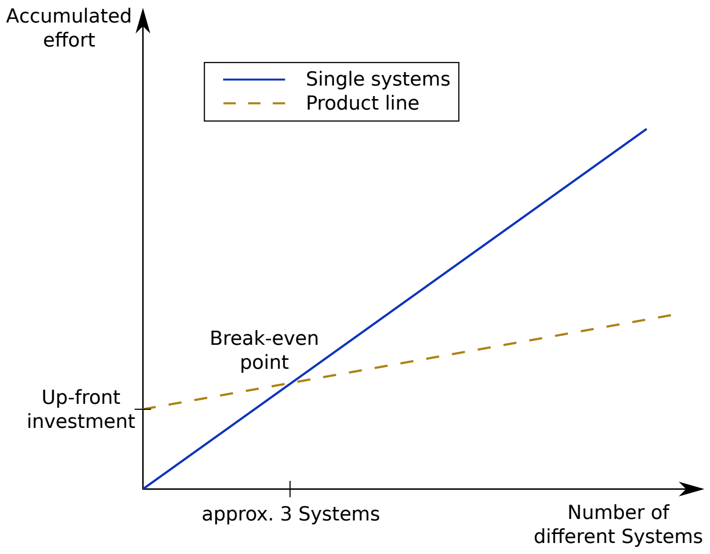


Figure 2: Economics of single system development versus using a software product line. Slightly modified version taken from [Van der Linden et al., 2007].

2.1.2 Annotation-Based Software Product Lines

In this study we focus on software product line projects exposing variability through annotations in implementation artifacts (i.e., its source code files). To illustrate this approach we show how this is done in the C/C++ programming language. Other variation mechanisms exist [Ye et al., 2009]. Examples include but are not limited to parametrization / templates, the use of dynamic link libraries (DLLs) [Gacek and Anastasopoules, 2001] or deploying a service-oriented architecture (SOA) via plug-ins [Cohen and Krut, 2010].

C and C++ allow for conditional inclusion, i.e., compilation by use of preprocessor directives like `#ifdef` or `#ifndef`. The preprocessor is a program that supports text macro replacements. It is executed before compilation and also resolves `#include`-directives; essentially copying that files content into the source file. We can use GCC[2] -

[2]https://gcc.gnu.org/

a compiler collection - to show the output of the preprocessor. Suppose we have the following C++ source code file:

```cpp
#include <cstdio>
void f() {
  #ifdef DEBUG
    dumpFullContext();
    calculate();
  #else
    calculate();
  #endif
}
```

We invoke `gcc`; instructing it to only run the preprocessor and not compile, assemble or link by passing the `-E` flag. Looking at the output for function `f()` only and stripping empty lines, we get:

```cpp
void f() {
  calculate();
}
```

Had we instead *#define*-d the flag `DEBUG` or passed it to gcc via the command line (by passing `-DDEBUG`), the output would be the following:

```cpp
void f() {
  dumpFullContext();
  calculate();
}
```

Note that all #-directives have been resolved and therefore removed.

By nesting these directives, thus forming propositional formulas, we can control which code is included, i.e., excluded. To illustrate, see Listing 1:

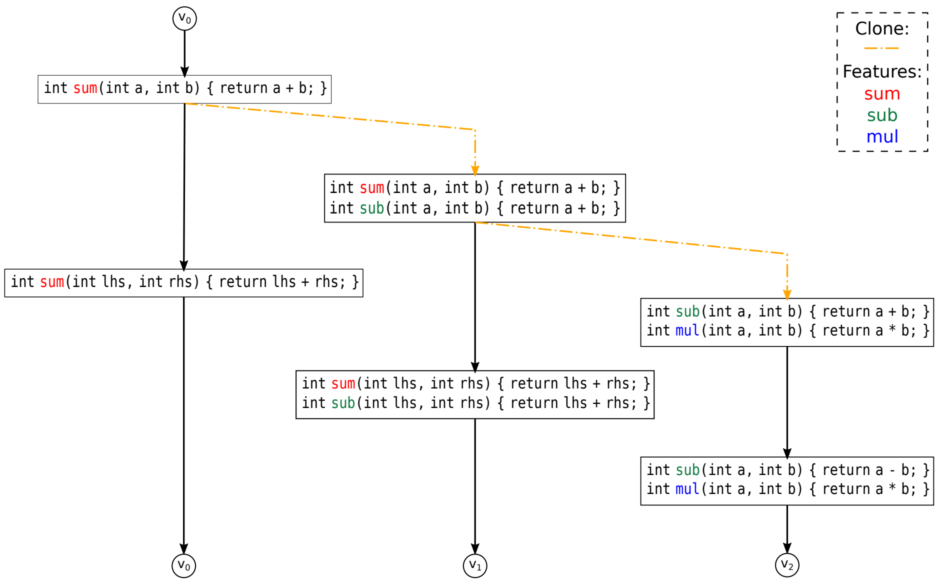
```
1 #ifdef USE_MODULE_A
2   #if FEATURE_B_SUPPORTED || FEATURE_C_SUPPORTED
3     // ...
4   #endif
5 #endif
```

Listing 1: C/C++ source code snippet containing preprocessor directives.

Line 3 in listing 1 is only processed (i.e., parsed and resolved) if both directives above evaluate to true, i.e., module A is used and at least one of features B or C is supported.

2.1.3 Clone-and-Own Development

The practice used most often for developing software systems is Clone-and-Own. You start by making an exact copy of an already existing project or product, i.e., using forking or branching in version control systems. This is the *clone* part. Modifications and changes necessary to derive a new variant, that satisfies certain requirements, are what is referred to by the *owning* part. Clone-and-own is straightforward. No up-front investment (cf. Fig. 2) like domain analysis is needed and you have a working prototype right away. There is also the independence of developers to make any necessary modifications that is stated [Dubinsky et al., 2013] as a result of using this approach. There is a major drawback however - it does not scale well with the number of variants.



```
int sum(int a, int b) { return a + b; }
```

```
int sum(int a, int b) { return a + b; }
int sub(int a, int b) { return a + b; }
```

```
int sum(int lhs, int rhs) { return lhs + rhs; }
```

```
int sub(int a, int b) { return a + b; }
int mul(int a, int b) { return a * b; }
```

```
int sum(int lhs, int rhs) { return lhs + rhs; }
int sub(int lhs, int rhs) { return lhs + rhs; }
```

```
int sub(int a, int b) { return a - b; }
int mul(int a, int b) { return a * b; }
```

Figure 3: Evolution of a clone-and-own project.

Figure 3 shows the history of a clone-and-own project implementing a heavily simplified math library. Altogether, it implements the three features sum, sub and mul. Lines going from one rectangle down to another depict the history of these variants; with orange dot-dashed ones indicating a *clone* taking place. Differences in the content of upper and lower rectangle constitute the changes introduced as part of this evolution step. Suppose that variant v_0 is some kind of basic model or library; whereas variants v_1 and v_2 are more sophisticated ones implementing additional, but different functionality.

At first, there is only variant v_0 implementing the feature sum. Later on the project was cloned and extended by feature sub; forming variant v_1. After that, variant v_0 was modified. Since $v_0 - v_2$ are developed independently (cf. Sec. 2.1.3), there is no propagation of bug-fixes or tracking of similar features already implemented in other variants of the code base. The copy-and-paste bug of feature sub in v_1 - returning the sum instead of the difference - was only spotted and fixed in the latest update to variant v_2.

The challenging part begins when bugs like the one seen in Figure 3 should also be fixed in other variants. This is hard because functions and variables might have been renamed or the code otherwise transformed; making it difficult to know if other variants are affected. If so, problems continue trying to locate the matching piece of code. Obviously this maintenance problem only gets worse, the more variants there are in total. This is in contrast to software product line engineering (cf. Section 2.1.1), where a common base and thereby all variants containing the refactored artifacts are changed.

2.1.4 Variant Drift

Variant Drift. The case study and hypotheses formulated as part of it presented by Schultheiß et al. indicate a difference in evaluation results between techniques that use SPL variants and those done on clone-and-own variants. The authors argue that the reason for this difference is a phenomenon called variant drift [Schultheiß et al., 2020].

Variant drift is a property of cloned variants. These drift apart from each other over time. The definition introduced by Schultheiß et al., whereby variant drift is the introduction of unintentional divergences in semantically equivalent software artifacts [Schultheiß et al., 2020] is adopted and extended here. It is useful to distinguish between the intentional and unintentional introduction of divergence.

Intentional divergences are all divergences which are introduced as part of what differentiates this variant from others. The implementation or modification of a specific feature is an example that falls under this category.

Unintentional divergences are changes to software artifacts, though they are not propagated to other variants. The modifications are usually intentional. Examples include refactoring efforts or the fixing of a bug in only a single variant.

2.2 SPL Variants for Clone-and-Own Research

2.2.1 Motivation

Managing the evolution of variant-rich systems is challenging. A more recent line of research thus focuses on assisting developers through the proposal of new methods, tools and better automation [Lapeña et al., 2016, Schmorleiz and Lämmel, 2014, Bittner et al., 2021]. A problem encountered in that field of research is the fact that there is a "*[...] substantial lack of publicly available clone-and-own projects which could be used as experimental subjects*" [Schultheiß et al., 2020]. Consequently, variants derived from software product lines are frequently used in studies and research instead. As the maturity of a research field depends on the availability of commonly accepted benchmarks [Strüber et al., 2019], Schultheiß et al. introduced the tool suite VEVOS (V*ariant* EVO*lution* S*imulation*) [Schultheiß et al., 2022a].

VEVOS is a benchmark generation framework which allows simulating the evolution of multi-variant software systems; specifically clone-and-own projects. It is part of the research project called VariantSync[3] that aims at bridging the gap between clone-and-own development and software product lines [Kehrer et al., 2021]. Given the history, i.e., version control of a software product line, VEVOS enables the user to generate clone-and-own variants for use in research. This works by first extracting a *ground truth* - knowledge about existing features, their relationships, and traces to their implementation - and using this information to generate clone-and-own variants. VEVOS is thus divided into two main components: the ground truth extraction, called VEVOS_Extraction (cf. Section 2.2.2) and the variant simulation called VEVOS_Simulation (cf. Section 2.2.3).

The benchmarks (i.e., variants) generated with VEVOS form the basis regarding our investigational study. To reiterate: The shortage of publicly available clone-and-own projects led to the quest of generating suitable benchmarks [Schultheiß et al., 2022a]. However, the generated variants do not expose characteristics introduced to real clone-and-owns variants like variant drift (cf. Section 2.1.4). They might be biased, not generalizable and pose threats to validity [Schultheiß et al., 2020]. It is this viewpoint which motivates us to apply automated refactoring in an effort towards simulating variant drift (cf. Section 4).

[3] `https://github.com/VariantSync`

2.2.2 VEVOS' Ground Truth Extraction

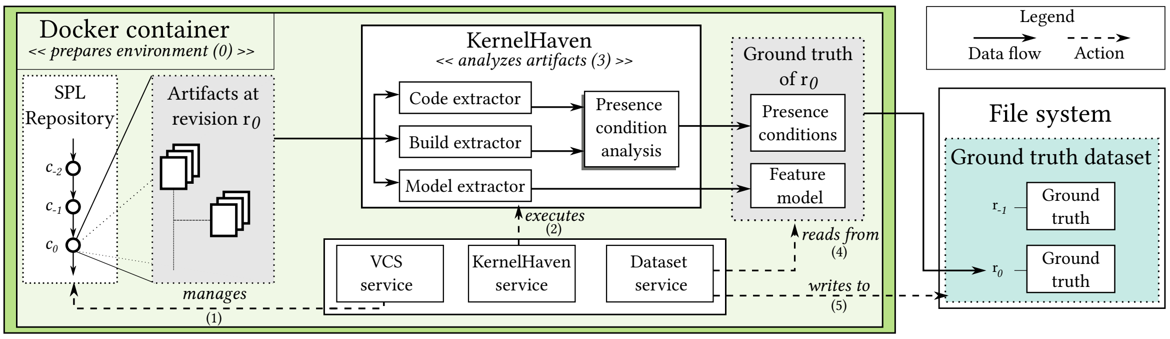


Figure 4: Overview of VEVOS' ground truth extraction library.
Taken from [Schultheiß et al., 2022a].

VEVOS extends the functionality provided by KernelHaven; a tool for performing different analyses on product lines[4]. Additional I/O-functionality and a KernelHaven plugin that analyzes presence conditions form the extraction framework. The version control service (VCS) checks out commits of the SPL repository one by one (1); restoring that commits' artifacts. For each commit, the KernelHaven service configures and executes KernelHaven itself (2). A series of extractor plugins are now run; performing the actual extraction (3):

Extractor plugin	**Analyzes**	**Output**
Code extractor	Variability in source code files	Code blocks and their respective block conditions
Build extractor	Product line's build files	Presence condition of each source file
Model extractor	Build system	Features and constraints

Outputs of the code and build extractor plugins are fed into the presence condition analysis plugin; yielding the entire code's presence conditions. Together with the feature model they form the final ground truth for that commit. Eventually, the dataset service collects (4) and packages (5) the ground truth. The progress then starts at (1) with the next commit.

Part of the ground truth extracted by VEVOS are presence conditions of code. These are a combination of file and block conditions. The condition under which a file is included (or excluded) into compilation is known as file condition. This information is usually undocumented in projects and buried behind build systems such as Make[5]. The condition under which specific lines of code are included (or excluded) is called block condition.

[4]https://github.com/KernelHaven/KernelHaven
[5]https://www.gnu.org/software/make/

2.2.3 VEVOS' Variant Simulation

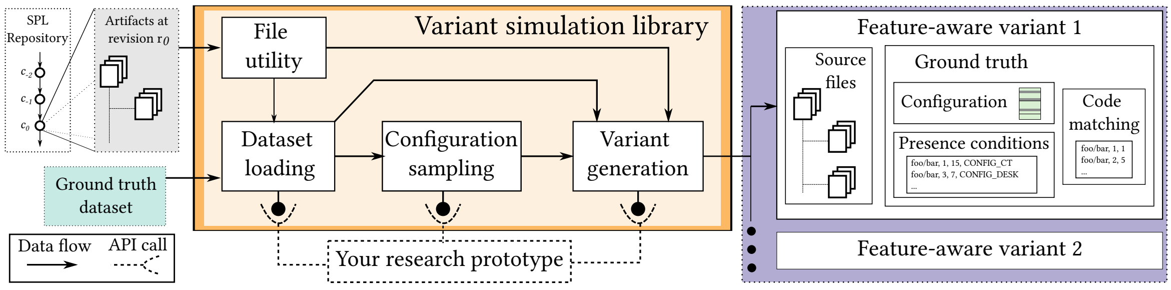


Figure 5: Overview of VEVOS' variant simulation library.
Taken from [Schultheiß et al., 2022a].

The second core module of VEVOS' tool suite is the variant simulation library. An overview of its components is shown in Figure 5. Once a ground truth has been extracted using VEVOS_Extraction, the resulting dataset may be loaded and parsed by the dataset loading component. The simulation module implements two sampling strategies within the configuration sampling component. *Constant sampling* uses a predefined set of configurations, while *random sampling* offers the creation of random, but valid configurations from the extracted feature model. Additionally, custom samplers may be added. Finally, the variant generation component is responsible for deriving *feature-aware* variants. Schultheiß et. al define this term to be a property of individual variants. Specifically, it refers to the availability of both configuration and presence conditions for the code [Schultheiß et al., 2022a]. Each feature-aware variant generated by VEVOS thus comprises source files, configuration, presence conditions mapped to code, block and file conditions and a code matching; mapping variant code to product line code via line numbers (cf. Figure 5).

2.3 Introduction to Compilers

Before taking a look at how refactoring can be applied technically (cf. Section 3.2), a rough understanding of the workings of a compiler is helpful. This section aims at providing the necessary overview while working towards the key component of AST-based refactoring - the AST (i.e., abstract syntax tree) itself.

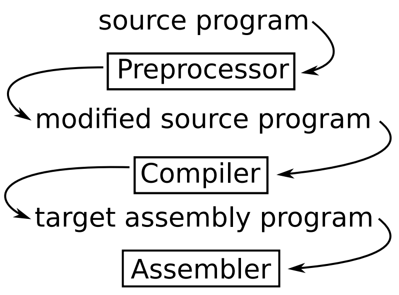


Figure 6: A language-processing system. Modified version taken from [Aho et al., 2007]

A compiler is a kind of language processor. It is "*[...] a program that can read a program in one language - the source language - and translate it into an equivalent program in another language - the target language.*"[Aho et al., 2007] As can be seen from Figure 6, a compiler sits right in between preprocessor and assembler. The preprocessor performs macro expansion and file inclusion; indicated by its output *modified source program*. The language most often associated with having a preprocessor is C. Other languages implement different technologies capable of performing similar functionality. In a pipeline fashion, the output of a previous phase is the input of the next. Once the compiler did its job, the target machine/assembly program is ready to be consumed by the assembler.

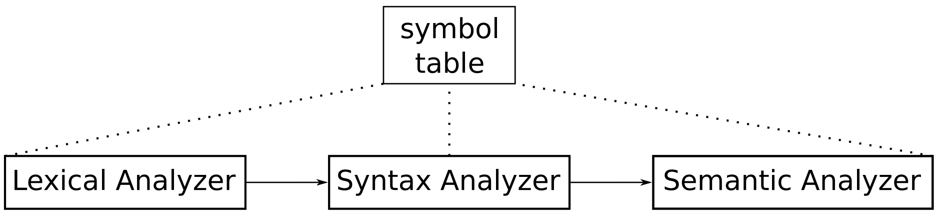


Figure 7: The first three phases of a compiler.

Providing an overview of every phase a compiler goes through from start to finish is beyond the scope of this study. Instead, we will focus only on the first three (depicted in Figure 7) by examining a running example. For an introduction or more information in general, the interested reader might check out the *Dragon Book*[6]. The symbol table (top box in Fig. 7) is a data structure in which each symbol (i.e., identifier) is mapped to information like the location it appears in the source code or its type. The symbol table is used by every phase.

[6]https://suif.stanford.edu/dragonbook/

Running Example[7]**.** We will look at how a typical compiler would parse the following assignment statement composed of two binary operators + and $*$:

```
position = initial + rate * 60 // comment
```

Listing 2: Our running example - An assignment statement.

Lexical Analysis. The first phase of a compiler is called lexical analysis and is responsible for tokenizing, i.e., splitting or grouping the input character stream into tokens. That is, for every meaningful sequence - also known as lexeme - the lexical analyzer transforms it into a token. What constitutes a token is usually defined by regular expressions. Extra white spaces separating the lexemes and comments are removed. For our example, the output would look like this:

```
<ident,1> <=> <ident,2> <+> <ident,3> <*> <60>
```

Constructs of the form `<token>` and `<token-name,token-value>` are tokens. For example, the equal sign (=) is a lexeme and is mapped to the token `<=>`. `position` is also a lexeme and is mapped to `<ident,1>`. This indicates that `position` is an identifier. Its second component (`1`) points to an entry representing it in the symbol table.

Syntax Analysis. The second phase of a compiler is syntax analysis. In this phase, the parser takes the tokens produced by the previous phase and constructs a parse-tree; that is, an intermediate representation in tree-like form. It does so using the formal grammar of the language. Most programming languages are defined both in terms of their semantics and syntax. The latter is defined by (context free) grammars. Syntactically correct language constructs can then be derived from this grammar, i.e., the production rules that represent it. A typical parse-tree representation is an *abstract syntax tree* (AST), or just syntax tree. In this tree, each interior node represents an operation. Children of a node represent the arguments to that operation [Aho et al., 2007]. A syntax tree for the statement in Listing 2 is shown below:

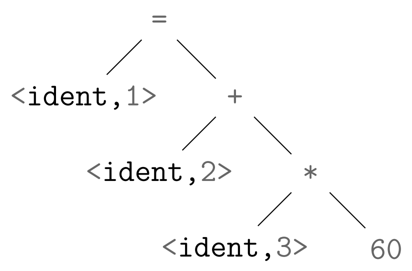


Figure 8: Abstract syntax tree for statement in Listing 2.

This tree also reveals the precedence of operators - nodes deeper in the tree have a higher precedence compared to nodes further up. Thus multiplication ($*$) is performed prior to addition (+).

[7]This example (the assignment statement) is taken from [Aho et al., 2007].

Semantic Analysis. The third phase of a compiler (and last one we are going to look at) is the semantic analyzer. It uses the syntax tree constructed in the previous phase, plus information stored in the symbol table, to check the source programs compliance with the semantics imposed by the language. During its work, it might also add additional information to both syntax tree and symbol table. It is also this phase, in which type checking is performed. Up to this point, we might have a syntactically well-formed program, but that does not mean that it does what it should or does not have undefined behavior (UB). In C++, undefined behavior is defined as: "*Renders the entire program meaningless if certain rules of the language are violated.*"[8] which translates to "anything could happen". Examples of UB include signed overflow and null pointer dereference. Types help us to prevent many of these undefined behaviors. If objects and expressions have an associated type, we can enforce, i.e., check operations permitted for these entities.

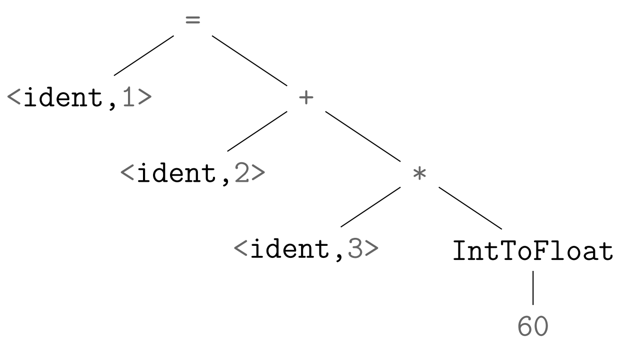


Figure 9: AST of statement in listing 2.
`IntToFloat` depicts an implicit type conversion.

Recall that `<ident,3>` represents identifier `rate` from Listing 2. If we assume, `rate` was declared as a floating-point type, say `double`, a language might perform an implicit conversion of the integer literal 60 to a floating-point type. The syntax tree in Figure 9 contains an additional node indicating such conversion.

[8] https://en.cppreference.com/w/cpp/language/ub

3 Software Refactoring

Refactoring is the process of improving existing code without creating new functionalities [Fowler, 2018, Vassallo et al., 2019].

3.1 Research on Software Refactoring

Research implies that refactoring is done for a multitude of reasons. Kataoka et al. showed that refactoring can lead to improved code metrics; specifically focusing on the aspect of maintainability, i.e., coupling, cohesion and complexity [Kataoka et al., 2002]. This aligns with "*[...] the major part of the total software development cost is devoted to software maintenance.*" [Mens and Tourwé, 2004]. Moser et al. seem to agree with the narrative that refactoring has "*long-term benefits on the quality of a software product [...]*" and that "*refactoring rather increases than decreases development productivity [...].*" [Moser et al., 2007] Wang et al. shine a different light on the matter by looking into the human behavior domain. They found that besides intrinsic motivators which include *Responsibility with Code Authorship* and *Unconscious Habit*, there are also external motivators like *Recognitions from Others* [Wang, 2009].

The large-scale empirical exploration on refactoring activities in open source software projects done by Vassallo et al. considers a list of 11 refactoring operations taken from [Fowler, 2018]. All of them aim at improving the design of the code from different perspectives [Vassallo et al., 2019]. This was done by choosing a variety of refactorings which move or rename parts of the code. Table 1 comprises a subset of these operations relevant later in this study:

Refactoring operation	**Description**
Extract-Method	Extract a fragment of a method into a new method whose name explains its purpose.
Move-Field	Create a new field in the target class, and change all its users.
Rename-Method	Replace the name of a method with a new one.

Table 1: Subset of refactoring operations used in [Vassallo et al., 2019].

Incidentally, these three refactoring operations are also among those used most often in the considered ecosystems in [Fowler, 2018]. Research conducted by Tsantalis and Chatzigeorgiou focuses solely on the Extract-Method operation. Specifically, they propose an approach for automatically identifying refactoring opportunities related with the complete computation of a variable [Tsantalis and Chatzigeorgiou, 2011]. Their work includes rules governing behavior preservation and usefulness of the extraction. It serves as a good example of the conceptual complexity and considerations involved when implementing a seemingly simple refactoring operation.

3.2 AST-Based Refactoring Using Clang-tidy

In Section 2.3, we ended up with an intermediate representation of the source program; the abstract syntax tree or AST for short. In this section we will see how we can use the AST. While Clang offers the ability to create standalone tools based on Clang's LibTooling[9], we deem the already existing tool clang-tidy capable of aiding us in simulating variant drift.

According to its online documentation[10], clang-tidy is a clang-based C++ "linter" tool.

The term *linter* dates back to the 1980s when Stephen C. Johnson, working at Bell Labs at the time, created the unix utility Lint. In a paper published in 1978, Johnson describes it as a command that "[...] examines C source programs, detecting a number of bugs and obscurities."[Johnson, 1977]. Today we think of a linter as a static analysis tool, assisting the programmer by flagging violations regarding formatting style, guidelines and error-prone or otherwise suspicious constructs. Examples include the use of undeclared identifiers, suggestions to use more modern/safer language features and performance considerations. Many of such tools, clang-tidy included, offer automatic fixes when possible.

Clang-based refers to Clang - a C/C++/Objective-C compiler. It is part of the Clang project, which provides a language front-end (left-most box in Figure 10) and tooling infrastructure for languages in the C language family. It is one of the primary subprojects of the LLVM project - a collection of compiler and toolchain technologies.

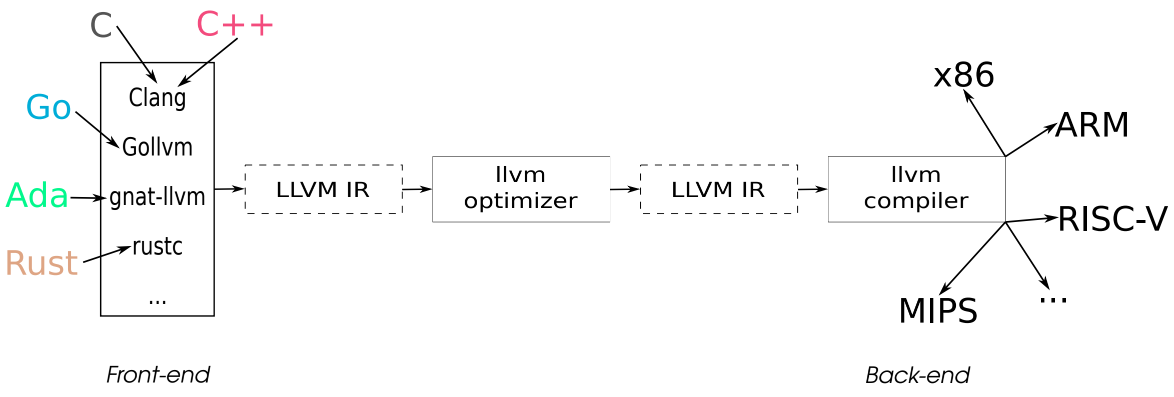


Figure 10: Overview of the LLVM compiler pipeline.

Fig. 10 is inspired by the graphic in this[11] blog article (Accessed 30 October 2022).

[9]https://clang.llvm.org/docs/LibTooling.html
[10]https://clang.llvm.org/extra/clang-tidy/
[11]https://blog.gopheracademy.com/advent-2018/llvm-ir-and-go/

LLVM supports multiple source languages and target architectures. While the front-end is concerned with parsing and error checking (cf. Sec. 2.3), the back-end (right-most box in Fig. 10) is responsible for the generation of target platform-specific code. The optimizers job is it to apply a multitude of different optimizations regarding loops or dataflow.
Using clang-tidy large-scale refactorings are possible. Since the tool has semantic understanding of the C/C++ code, it knows what a pointer is, when something is called or a variable is referenced. This works by using a technique called AST-matching, where AST matchers are predicates on AST-nodes. Nested matchers are used to make a matching more precise. Aforementioned technique is further explained in the following subsection.

3.2.1 Clang-tidy Demonstration

Clang-tidy calls the set of static analyses *checks*. Each of them focuses on a single suspicious construct like the violation of some guideline or use of bugprone code (a.k.a. code smell). These checks are organized in modules (i.e., categories of checks). Clang-tidy comes with a number of checks already implemented[12]. At the time of writing there exist 470 checks; although many of them are vendor specific or enforce developer policies of libraries. The command line options `--list-checks` and `--checks=<string>` can be used to determine the currently enabled checks:

```
# show checks enabled by default
$ clang-tidy --list-checks
# no checks enabled
$ clang-tidy --list-checks --checks=-*
# show only checks in module/category readability
$ clang-tidy --list-checks --checks=-*,readability-*
```

Let us assume, there exists the following C++ file named `main.cpp`:

```cpp
#include <iostream>
#include <vector>
#include <iostream>

int main() {
  int number;
  std::cin >> number;
  if (number % 2 == 0)
    std::cout << "number is even" << std::endl;

  return 0;
}
```

Listing 3: Sample code *before* using clang-tidy.

Running clang-tidy using the following command will flag problems with above code:

```
$ clang-tidy                                                \
    --checks=-*,readability-*,bugprone-*,cppcoreguidelines-*    \
    main.cpp --
```

Listing 4: Invocation of the clang-tidy tool.

Note the `--` after specifying the source file `main.cpp`. This option signals to clang-tidy, that no compiler-flags are passed.

[12] https://clang.llvm.org/extra/clang-tidy/checks/list.html

We get the following diagnostics:

```
/path/to/file/main.cpp:3:1: warning: duplicate include [readability-duplicate-include]
#include <iostream>
^~~~~~~~~~~~~~~~~~~
/path/to/file/main.cpp:6:7: warning: variable 'number' is not initialized [cppcoreguidelines-init-variables]
int number;
    ^
           = 0
/path/to/file/main.cpp:8:23: warning: statement should be inside braces [readability-braces-around-statements]
if (number % 2 == 0)
                    ^
                     {
```

These warnings are self-explanatory. When appending the `--fix` option to above command in listing 4, clang-tidy will apply the suggested fixes to the file - in our example resulting in:

```
1  #include <iostream>
2  #include <vector>
3
4  int main() {
5    int number = 0;
6    std::cin >> number;
7    if (number % 2 == 0) {
8      std::cout << "number is even" << std::endl;
9    }
10
11   return 0;
12 }
```

Listing 5: Sample code *after* using clang-tidy.

The duplicate inclusion of header `<iostream>` in line 3 in Listing 3 was removed. The variable `number` was initialized with a value of zero in line 5 in Listing 5 and braces were inserted around the body of the if-statement in line 7 in Listing 5.

3.2.2 Workflow for Extending Clang-tidy

Now that we understand what clang-tidy can do, let us take a look at a basic workflow presented by Stephen Kelly[13] to implement our own checks simulating variant drift. We are about to create a new check called `alternative-operator-and` inside the module or category of checks `readability`. The goal of this check is to detect uses of the binary logical operator `&&` (as in `true && false`) and to replace them with their alternative operator representation[14] `and` (as in `true and false`).

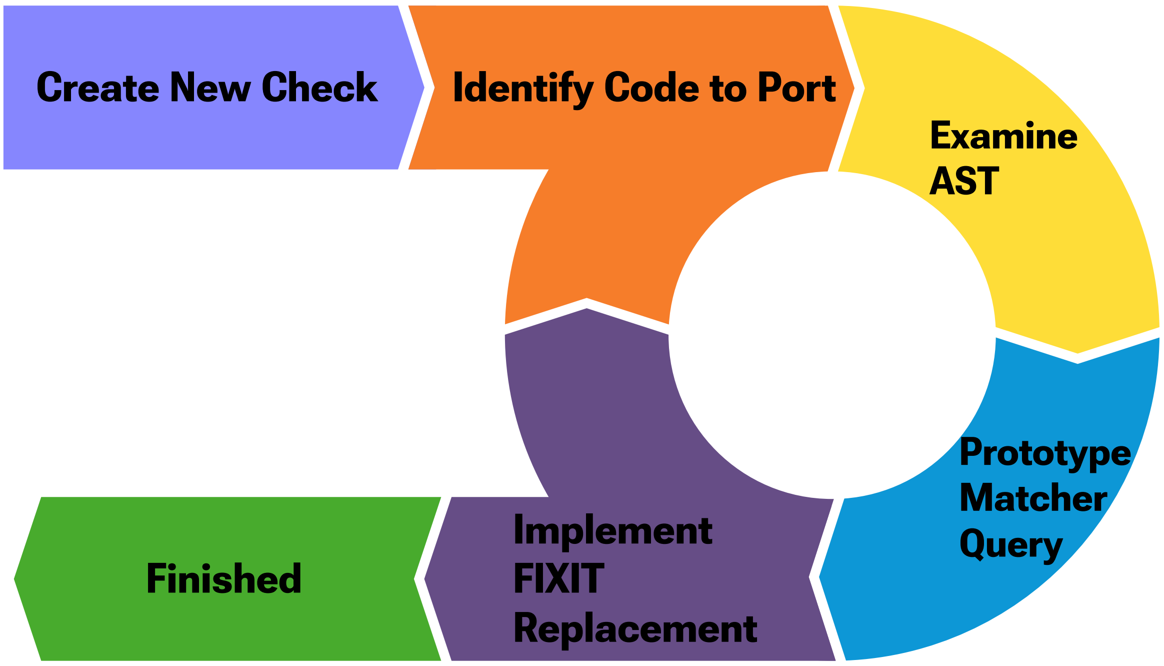


Figure 11: Basic workflow - the steps involved in creating a clang-tidy check. Recreated version of graphic in blog article referenced by Footnote 13.

We start by making sure a check like the one we want to add does not already exist. A typical way to do that would be to check the website (referenced by footnote 12 on page 17) listing all built-in checks or to use the `--list-checks` option with clang-tidy and piping its output to grep.

[13] `https://devblogs.microsoft.com/cppblog/exploring-clang-tooling-part-1-extending-clang-tidy/`
[14] `https://en.cppreference.com/w/cpp/language/operator_alternative`

Create New Check. There exists a python script `add_new_check.py` inside the *llvm-project/clang-tools-extra/clang-tidy/* directory to assist in creating the necessary new files and editing existing build files to include our check. We invoke this script by passing the name of the module under which the check will get included and the name of the check itself. Rebuilding and repeating the query described above inside LLVM's build directory should now list our check:

```
$ ./clang-tidy --list-checks --checks=* | grep alternative

    cppcoreguidelines-alternative-operator-and
```

Taking a look at Fig. 11 reveals, we enter the loop of Identify Code to Port, Examine AST, Prototype Matcher Query, and Implement FIXIT Replacement. All of these steps will be illustrated briefly.

Writing a clang-tidy check is similar to writing a regular expression - the goal is to match what you want and *only* what you want. With each iteration you might discover cases, which your current matcher does not handle yet, but which it should, and vice versa.

Identify Code to Port. We want to match uses of the binary logical `operator&&`. We do not want to match rvalue references and forwarding references. In C++ it is also possible to overload `operator&&` - we want to match such overloads as well.

In particular, we want to match on code like this:

```
void f() {
  bool b = true && false;                   // match
  bool lhs = true, rhs = false;
  bool res = lhs && rhs;                    // match
  if (false && false || true && true) {}    // match twice
}

struct BoolWrapper { bool b; };
bool operator&&(const BoolWrapper& lhs, const BoolWrapper& rhs) {
  return lhs.b && rhs.b;                    // match
}
```

Listing 6: Examples of code usages we want to match on.

We do not want to match on code like the following:

```
1 bool b = true and true;                    // no match
2 int&& rValueRef = 5;                       // no match
3 template<class T>
4 void my_func(T&& x) {}                     // no match
```

Listing 7: Examples of code usages we do not want to match on.

Examples like those in Listing 7 also demonstrate, how a naive search and replace approach - with the help of an IDE or the use of tools like sed[15] - would not produce a desired result a lot of times. That is, simply replacing the string "&&" of the statement `int&& rValueRef = 5;` in Listing 7 (line 2) would result in "`intand rValueRef = 5;`". This however is non-compiling C++ code because of an unknown type `intand`.

Examine AST. Once we have a better idea of the code we want to match, we can use Clang to dump the AST. We invoke the Clang compiler, passing it flags that control the behavior during compilation. In particular, we instruct Clang to print the AST using `-ast-dump` and to run only the preprocessor, parser and semantic analysis stages using `-fsyntax-only`.

```
|   |-DeclStmt 0x55faf25888b8 <line:2:3, col:25>
|   | `-VarDecl 0x55faf2588810 <col:3, col:20> col:8 b 'bool' cinit
|   |   `-BinaryOperator 0x55faf2588898 <col:12, col:20> 'bool' '&&'
|   |     |-CXXBoolLiteralExpr 0x55faf2588878 <col:12> 'bool' true
|   |     `-CXXBoolLiteralExpr 0x55faf2588888 <col:20> 'bool' false
```

Listing 8: AST output for statement `bool b = true && false;`.

From the output shown in Listing 8 we can see that variable `b` is represented in the AST as a `DeclStmt` (statement that declares something) or more precisely a `VarDecl` (declaration of a variable). Furthermore, we know the type of the variable being declared is `bool` and it is initialized using constant initialization (an initialization with assignment, `cinit`). This initialization is the result of a `BinaryOperator` called `&&`. The arguments of `BinaryOperator` are boolean literals which are decoded by `CXXBoolLiteralExpr` AST nodes and have values `true` and `false` respectively.

[15]A stream editor used for basic text transformations.
https://www.gnu.org/software/sed/manual/sed.html

Using Clang's AST Matcher Reference[16] and another tool which comes as part of LLVM called `clang-query`, we can write the matcher now. `clang-query` allows to quickly write, debug and iterate matchers, that can be easily transformed into a clang-tidy check later.

Prototype Matcher Query. From our observations made in the previous step, writing our matcher is not complicated. Matcher expressions on AST nodes are defined by a combination of basic matchers using a predicate-like language. This allows for the formulation almost in natural language:

```
binaryOperator(hasOperatorName("&&"))
```

Listing 9: Matcher query to match `operator&&`.

There are three categories of matchers in Clang:

- Node Matcher - matches a specific type of AST node
- Narrowing Matcher - matches attributes on AST nodes
- Traversal Matcher - allows traversal between AST nodes

In above matcher (cf. List. 9) we combine the node matcher `binaryOperator()`, which matches binary operator expressions and the narrowing matcher `hasOperatorName()` passing it the string `"&&"` we found out about from the AST representation of the code (cf. List. 8).

More complex matchers like the following can be built:

```
callExpr(callee(
  functionDecl(hasName("foo"))),
  argumentCountIs(1)
)
```

This matcher reads: "Match calls to a function named `foo` taking a single argument."

Implement FIXIT Replacement. All that is left to do is to take our matcher from the previous step and transform it into a clang-tidy check. We do that by subclassing `ClangTidyCheck` - the base class for all clang-tidy checks - and overriding some of its member functions. Specifically, we are providing overrides for functions `registerMatchers()` and `check()`. In the first function we add one or more AST matchers, which will be used to find the pattern we specify:

[16] https://clang.llvm.org/docs/LibASTMatchersReference.html

```cpp
1  void AlternativeOperatorAndCheck::registerMatchers(MatchFinder *Finder) {
2    Finder->addMatcher(
3      binaryOperator(hasOperatorName("&&")).bind("logical_and"),
4      this
5    );
6  }
```

Listing 10: Override of function `registerMatchers()`.

Note the `.bind("logical_and")` in line 3 in listing 10. Node matchers are actually the only category that supports the `bind("id")` call. It binds the matched node to the given string `"id"`, which allows to retrieve it later (see line 3 in listing 11):

```cpp
1  void AlternativeOperatorAndCheck::check(const MatchFinder::MatchResult &Result) {
2    const auto *LogicalAnd
3      = Result.Nodes.getNodeAs<BinaryOperator>("logical_and");
4    if (LogicalAnd) {
5      SourceLocation opLoc = LogicalAnd->getOperatorLoc();
6      SourceManager &sm = *Result.SourceManager;
7      SourceLocation opEndLoc
8        = clang::Lexer::getLocForEndOfToken(opLoc, 0, sm, clang::LangOptions());
9      if (opEndLoc.isValid()) {
10       SourceRange opRange(opLoc, opEndLoc);
11       std::string opText = get_source_text_raw(opRange, sm);
12       if (opText == "&&") {
13         diag(opLoc, "consider using 'and' instead of '&&' for better readability")
14         << FixItHint::CreateReplacement(opRange, "and");
15       }
16     }
17   }
18 }
```

Listing 11: Override of function `check()`.

On each match to one of the registered matchers, clang-tidy will call-back our overriden `check()`-function. Details of the exact workings of this function are outside the scope of this work. In essence, we first retrieve the matched node, i.e., its location using the `"logical_and"` identifier that we bound earlier. If the location is valid (the operator itself is not the result of a macro expansion) we can use clang-tidys `diag`-API to build a diagnostic that can be emitted. Additionally we annotate this diagnostic with some code of interest. In particular, `FixItHint` allows for insertion, removal, or replacement of code at a specified location or source code range.

4 Method

The goal of our study is to assess the feasibility and technical challenges of simulating variant drift (cf. Sec. 2.1.4) in SPL-variants generated with VEVOS. More specifically, we are interested in combining VEVOS with automated refactoring based on clang-tidy. To guide our investigations, we formulate four research goals that focus on the availability of suitable subject systems, the technical difficulties of simulating variant drift, and the refactoring strategy with respect to VEVOS' benchmark generation.

4.1 Research Goals

RG1: Find suitable subject systems.

VEVOS is designed to be applicable to software product lines and has recently been extended to support any kind of C-preprocessor-based product line. As pointed out in Section 3.2, clang-tidy is a C++-based linter tool. The purpose of this research goal is to shine light on the challenges of finding a suitable subject system and combining VEVOS with C/C++-targeted refactoring.

RG2: Investigate technical difficulties of simulating variant drift with clang-tidy.

With this goal we want to quantify if clang-tidy is suitable for introducing variant drift in product line variants. This is in question as it is not a common use-case. With this in mind, we want to explore requirements towards a good refactoring and possible challenges of connecting tools written in different languages (VEVOS in Java, clang-tidy in C++).

RG3: Evaluate the feasibility of simulating variant drift by refactoring generated variants directly.

Intuitively, refactoring variants generated by VEVOS directly seems feasible as each variant represents a complete piece of software that could have been implemented independently. However, the build files of product lines are written/configured for that product line. Generated variants might therefore not mirror their dependencies. It is unclear, whether build files remain intact after extraction, if the generated variants are compilable and to which extent this impacts the feasibility of automated refactoring.

RG4: Evaluate the feasibility of simulating variant drift by refactoring a product line's source code.

An alternative approach towards simulating variant drift might be to refactor the product line's source code instead. Here, the idea is that variants are refactored indirectly by generating them from the refactored source code of a product line. By generating variants in steps, with each step applying new refactorings to the product line, it might be possible to attain variants with inherent variant drift. However, refactoring the product line could invalidate the ground truth. Thus, we investigate the technical feasibility of this approach.

4.2 Undertakings

Besides understanding the workings of VEVOS itself, including publications leading up to its introduction [Schultheiß et al., 2022a] as well as reports on using it [Schultheiß et al., 2022b], a lot of time went into getting to know the LLVM infrastructure. Trough testing and using many of Clang's tools like clang-query, clang-rename or clang-reorder-fields, we investigated possibilities of achieving the goal of simulating variant drift. Despite the option to create a standalone refactoring tool we eventually decided that the built-in functionality offered by clang-tidy and its possibility to be extended by custom checks will suffice.

Furthermore, we have successfully managed to extend clang-tidy by a couple of custom checks. This includes a check, which transforms upper-case enumerators into their lower-case counterparts. Refactorings like this might emerge when developers decide to adhere to the corresponding C++ Core Guideline[17]: "*Don't use ALL_CAPS for enumerators*". Another check, besides the one described in Section 3.2.2, detects catch-blocks not catching by const reference and subsequently inserts the const to the catch-variable. Others were tried but quit when facing unresolved challenges described in Section 5.2.

During development and testing the project `libxml2` was used. It is a XML parser and toolkit written in C and was originally developed for the GNOME Project. Its history[18] encompasses roughly 5,5 thousand commits and its source code exposes variability through the use of the C preprocessor.

Finally, we began the quest of finding a suitable subject system.

[17]A living document providing guidelines to write simpler, more efficient and more maintainable code. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
[18]https://github.com/GNOME/libxml2

4.3 Threats to Validity

4.3.1 Internal Threats

Build instructions are sometimes not well documented, neglected or outdated. The third column on used build systems in Table 2 depicts our best efforts in determining this information.

4.3.2 External Threats

The set of analyzed subject systems (cf. Section 4.4) is only a subset of existing software product lines. Therefore other results might be obtained if a different set of subject systems is used. However, it is the largest set known to us, contains a considerable number of subject systems (44), and has been used in previous studies.
Our study focuses only on clang-tidy. We did not use other AST-based approaches. However, as clang-tidy has been around for many years, we deemed it applicable to reach our intent.

4.4 Subject Systems

As subject systems, we selected the 44 open-source preprocessor-based software product lines that were provided as part of the replication package for the paper *Classifying Edits to Variability in Source Code*[19]. Instead of further information about where these projects are hosted, we list used programming language and build system relevant to our discussion (see Table 2).
Hyphens in the third column of this table either represent projects using no build system at all or this information was not apparent.

[19] https://github.com/VariantSync/DiffDetective

Project Name	Domain	Build System	Language	#Commits
apache-httpd	web server	CMake, Make	C	33,272
berkeley-db-libdb	database system	Make	C, Tcl	7
busybox	embedded systems	Make	C	17,447
cherokee-webserver	web server	Make	C, Python	5,855
clamav	antivirus program	CMake	C	10,880
dia	diagramming software	Meson	C	6,673
emacs	text editor	Make	C, Emacs Lisp, Roff	161,404
freebsd	operating system	Make	C, C++	278,919
gcc	compiler framework	Make	Ada, C, C++	196,181
ghostscript	postscript interpreter	Make	C	22,900
gimp	graphics editor	Make, Meson	C	49,025
glibc	programming library	Make	C	38,318
gnumeric	spreadsheet application	Make	C	24,247
gnuplot	plotting tool	Make	C	11,984
Godot	game engine	SCons	C++	48,095
irssi	IRC client	Meson, Ninja	C	6,657
libssh	network	CMake	C	5,556
libxml2	XML library	CMake, Make	C	5,540
lighttpd	web server	CMake, Meson	C	4,658
linux	operating system	Make	C	1,136,447
lynx	web browser	-	PHP	125
Marlin	3d printing	-	C, C++	18,880
minix	operating system	Make	C, Roff	7,153
mplayer-svn	media player	Make	C, Roff	37,992
MPSolve	mathematical software	Make	ReScript, C	1,775
openldap	LDAP directory service	Make	C	24,096
opensolaris	operating system	Make	C	11,422
openvpn	security application	Make	C	3,387
parrot	virtual machine	Make	C, Perl	49,989
php	program interpreter	Make	C, PHP	130,281
Pidgin	instant messenger	Meson, Ninja	C	40,097
postgresql	database system	Make, Meson	C, PL/pgSQL	54,821
privoxy	proxy server	Make	C	7,558
cpython	program interpreter	Make	C, Python	114,979
sendmail	mail transfer agent	-	JavaScript	86
sqlite	databases	Make	C	8,664
subversion	revision control system	Make	C, Python	60,211
sylpheed	e-mail client	Make	C, HTML	2,682
tcl	program interpreter	Make	C, Tcl	26,131
vim	text editor	Make	C, Vimscript	17,109
xfig	vector graphics editor	CMake, Make	C, HTML	9
xine-lib	media library	Make	Shell	133
xorg-server	X server	Meson	C	17,918
xterm	terminal emulator	Make	C, HTML, Roff	112

Table 2: The subject systems considered in this study.

5 Results

In this section, we discuss the research goals posed as part of Section 4. Boxes at the end of each topic outline the key take-aways.

5.1 Finding a Suitable Subject System

To run clang-tidy over entire projects a compilation database is needed. This is necessary since full information about how to parse a translation unit is required by tools based on the AST. A compilation database is a JSON file, which consist of an array of command objects, where each command object specifies one way a translation unit is compiled in the project[20]. The database file can be generated automatically by some build systems, including CMake or Bazel on Linux.
However, as not all projects use any of these build systems, this restricts the number of applicable subject systems.

Excerpt of the file `compile_commands.json` generated for `libxml2`:

```
[  {
     "directory": "/home/seb/libxml2/cmake-build-debug",
     "command": "/usr/bin/cc -DLibXml2_EXPORTS ...",
     "file": "/home/seb/libxml2/buf.c"                    },
    ...
]
```

In this file, the following command objects are used:

- directory - working directory of the compilation
- command - compile command as a single shell-escaped string
- file - main translation unit source processed by this compilation step

We establish two requirements for a suitable subject system: First, they have to be mainly written in C/C++. And second: They have to use the CMake build system. We define *mainly written in C/C++* by a project having more than 10% of its files being either C or C++ files[21]. Finding such a project turned out to be not an easy task. While 41 out of the 44 projects listed in Table 2 were mainly written in C/C++, only six of them used CMake. As a subject system should fulfill both requirements, this restricts the proportion of applicable subject system to 6/44. This corresponds to a proportion of 13.6% (cf. right pie chart in Fig. 12). As build files are notoriously hard to work with, transforming Make files into CMake files was not an option.

[20] https://clang.llvm.org/docs/JSONCompilationDatabase.html
[21] As is displayed for projects hosted on GitLab or GitHub.

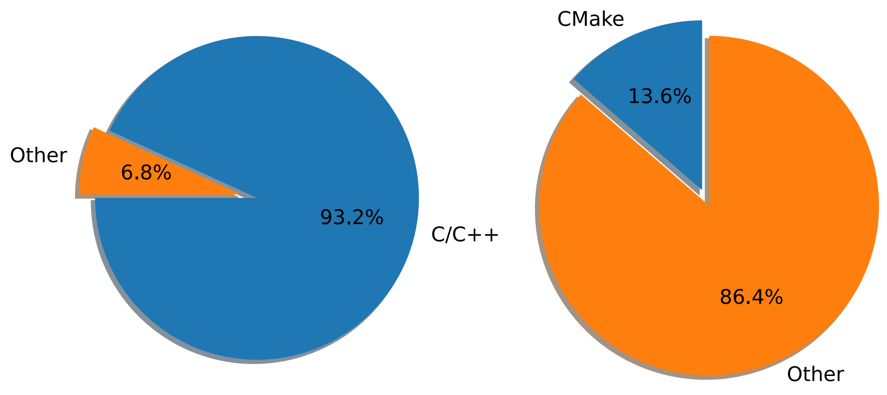


Figure 12: Proportion of subject systems fulfilling our requirements.
Left: Proportion of subject systems being mainly written in C/C++.
Right: Proportion of subject systems using the CMake build system.

- Only a small proportion of considered systems are applicable.
- The main restrictions are:
  - System is a C-preprocessor-based product line (VEVOS).
  - A sufficient portion of the project is written in C/C++ and uses a build system supporting automatic generation of a compilation database (clang-tidy).

5.2 Technical Challenges of Automated Refactoring

As mentioned in Section 3.1, finding refactoring opportunities, not to mention an optimal design solution is not easy. When done correctly, the source code modifications have to not only preserve the behavior, but also lead to improved code. Be it by lowering cohesion and coupling [Du Bois et al., 2004, Meyers and Binkley, 2007] or aiding in comprehension; ultimately resulting in easier to read, more maintainable and testable code. That being said, it does not come as a surprise, that automating the task of any non-trivial refactoring operation can be tricky.

Let us take the Extract-Method refactoring[22] to outline these challenges. According to its description (cf. Section 3), a developer might extract lines 2-5 in Listing 12 into a new method called `sum()`.

```
1 double average(std::vector<double> const& vec) {
2   double sum = 0.0;
3   for (const auto elem : vec) {
4     sum += elem;
5   }
6   return sum / static_cast<double>(vec.size());
7 }
```

Listing 12: Use case of the Extract-Method refactoring operation.

When strategizing how to best utilize clang-tidy, we found that clang-tidy on its ownClang-tidy on its own is not designed to recognize these refactoring opportunities. Trying to simulate variant drift using clang-tidy without additional techniques like program dependence graphs [Tsantalis and Chatzigeorgiou, 2011], we would have to select a random method and a random number of lines to extract within that method. While refactorings like these are technically possible with clang-tidy, it is the checks implementers' responsibility to guarantee that existing code is not breaking. By employing this random selection strategy we lose any realism and aspirations we have towards a good refactoring - in a sense almost contradicting its core principles. Note however, that existing checks built into clang-tidy and the custom checks we implemented are sufficient for simulating variant drift. They could provide new insight, even if the integration has to be further investigated in future work.

Moreover, VEVOS is written in the Java programming language. Clang-tidy on the other hand is C++ based. Knowing both languages is thus a requirement when trying to extend the behavior of either tool or combining results produced by both. We also found that in its current state, users of VEVOS are pretty constrained to the library itself. Output produced by either extraction or simulation are hard to work with on their own. Fortunately, clang-tidy is compiled into a single executable that can be invoked using its the command line interface. This simplifies the integration in VEVOS.

[22]Described problems also occur with other refactoring operations like Move-Field which change the structure of the code.

Lastly, we experienced that getting to know the LLVM infrastructure and its tools requires considerable time. Implementing a check that handles every corner case is hard. It requires a good understanding of the C/C++ programming langue. Additionally, knowledge about other LLVM tools - or at least its existence - is very useful, if not necessary. Even LLVM/Clang compiler developer V. Bridgers talks about an 80% rule regarding the covering of checks in his talk at the 2020 LLVM Developers' Meeting[23]. He is referring the fact that developers are usually content if the check handles 80% of the work. The rest is done by hand or a different tool.

- Large-scale refactoring using clang-tidy is possible but getting to know the LLVM infrastructure, its tools to be able to implement a good check is very time-consuming.
- More satisfactory results might be attainable when combining clang-tidy with additional techniques such as program dependence graphs.
- Results produced by either VEVOS' extraction or simulation are hard to work with outside of VEVOS.
- Offering a rudimental command line interface for manipulation of results could ease of use.

[23]https://llvm.org/devmtg/2020-09/slides/Clang-tidy_for_Customized_Checkers_and_Large_Scale.pdf

5.3 Refactoring Variants Directly

We began by exploring options of integrating clang-tidy into a workflow similar to the one presented in [Schultheiß et al., 2022b]. In this study the authors simulated change synchronization through the use of techniques based on document patching. They used the well known tools *diff* and *patch* to determine the evolution between a pair of consecutive commits, propagating changes to other variants and evaluating their applicability and correctness. As this study was already completed, conducting a similar one investigating differences in results to applicability and correctness seemed feasible at first. Because of the way VEVOS is implemented (i.e., being split into modules for extraction and simulation), as well as its internal workings (e.g., regarding the presence conditions), we faced unresolved challenges.

Refactoring operations changing the *structure* of the code (i.e., by moving, inserting or removing code) are interesting and challenging at the same time. That is because such refactoring might invalidate presence conditions, which reference line numbers. Recall that in order to generate feature-aware variants, VEVOS maps presence conditions to code, block and file conditions. Invalidated presence conditions thus also render the feature mappings derived by VEVOS' as useless. Unfortunately this implies that extraction has to be performed *after* applying each or a set of similar refactoring operations. However, this would result in long simulation times. We are facing restrictions imposed by the architecture of VEVOS.
There also exist refactoring operations like Rename-Method, which do not change the structure of the code. Simply changing the name of a method would not lead to an alteration of the presence conditions. Nevertheless, all of these operations are interesting as they change the *context* of the code and thereby represented features.

- The most interesting refactoring operations (changing the structure of the code) also invalidate extracted presence conditions and feature mappings.
- Generally, every refactoring is worthwhile as it changes the context.
- Performing extraction after applying refactoring operations results in long simulation times.

5.4 Refactoring Product Line's Source Code

VEVOS is split into two modules performing extraction (cf. Section 2.2.2) and simulation (cf. Section 2.2.3). While this makes sense from a separation of concerns perspective, especially since extraction takes considerably longer than simulation and is performed only once, this imposes some challenges. In particular: The generation of variants depends on prior extraction. Thus, the only way to reflect refactoring operations inside the ground truth extracted by VEVOS is if we inject them directly into the source code's history. Such approach could look like the following:

```
(1) Create empty folder F
(2) Initialize VCS repository R inside F
(3) For each commit in original SPL history:
    (4) Refactor code
    (5) Copy and overwrite content in F
    (6) Commit refactored code to R
```

Listing 13: Procedure of injecting refactoring into product line's source code.

Using these steps, we end up with a new history containing changes introduced by the refactoring while at the same time preserving the original software product lines' evolution. Which is a problem in its own right: Now the differentiation between changes introduced by the refactoring versus changes introduced by the natural evolution of the variant becomes infeasible.
To investigate various kinds of refactoring operations or different granularity thereof, we would have to create and handle multiple of these copies (emerging from step `(5)` in Listing 13). Furthermore, there exists no clear way to map between commits of both histories. Such a mapping only exists under the assumptions that refactoring operations affect these copies in the same way or that evolution and modifications introduced as part of the refactoring would end up at the same location. This remains to be proven.

- To incorporate refactoring into generated variants, refactoring operations have to be applied before extraction.
- We proposed a procedure to directly inject refactoring (i.e., variant drift) into the product line's source code.
- The feasibility of this approach requires further investigation:
  - It is unclear, whether a mapping between commits of distinct histories can be constructed.
  - It is not guaranteed that applying the same refactoring operation to different variants affects them equally.

6 Conclusion

In this study, we investigated the potential to simulate variant drift in software product line variants using AST-based refactoring. Schultheiß et al. introduced the tool VEVOS, which enables researchers to generate benchmarks and simulate the evolution of cloned variants. However, these benchmarks (i.e., variants) lack variant drift. To counteract this deficiency, our vision was to simulate variant drift by applying AST-refactoring operations using the C++ linter tool clang-tidy.
Besides a demonstration of how clang-tidy can be extended by custom checks, we examined requirements and challenges faced when trying to combine both tools. Our results show the intricacy of the matter. Specifically, we found that only a small proportion of the considered systems, six out of 44, are applicable at all. The main reason is clang-tidy's necessity to generate a compilation database in order to run it over entire projects. While we argue that large-scale refactoring using clang-tidy is possible, integrating clang-tidy into VEVOS posed additional challenges. These have to do with the fact that results produced by either VEVOS' extraction or simulation are hard to work with outside of VEVOS. To this end, we proposed two approaches regarding possibilities to integrate AST-based refactoring. The first one aimed at refactoring variants generated by VEVOS directly. The second targeted refactoring the product line's source code instead. Lastly, we outlined the reasons for AST-based refactoring using VEVOS being infeasible for the moment. In conclusion, we suggest that future research should focus on the development of new refactoring methods and tools that explicitly target the simulation of variant drift.

References

[Aho et al., 2007] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compilers: principles, techniques, & tools*. Pearson Education India.

[Apel et al., 2011] Apel, S., Heidenreich, F., Kastner, C., and Rosenmuller, M. (2011). Third international workshop on feature-oriented software development (fosd 2011). In *2011 15th International Software Product Line Conference*, pages 337–338. IEEE.

[Bittner et al., 2021] Bittner, P. M., Schultheiß, A., Thüm, T., Kehrer, T., Young, J. M., and Linsbauer, L. (2021). Feature trace recording. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1007–1020.

[Cohen and Krut, 2010] Cohen, S. and Krut, R. (2010). Managing variation in services in a software product line context. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

[Dijkstra, 1972] Dijkstra, E. W. (1972). The humble programmer. *Commun. ACM*, 15(10):859–866.

[Du Bois et al., 2004] Du Bois, B., Demeyer, S., and Verelst, J. (2004). Refactoring-improving coupling and cohesion of existing code. In *11th working conference on reverse engineering*, pages 144–151. IEEE.

[Dubinsky et al., 2013] Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., and Czarnecki, K. (2013). An exploratory study of cloning in industrial software product lines. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 25–34.

[Fischer et al., 2014] Fischer, S., Linsbauer, L., Lopez-Herrejon, R. E., and Egyed, A. (2014). Enhancing clone-and-own with systematic reuse for developing software variants. In *2014 IEEE International conference on software maintenance and evolution*, pages 391–400. IEEE.

[Fowler, 2018] Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[Gacek and Anastasopoules, 2001] Gacek, C. and Anastasopoules, M. (2001). Implementing product line variabilities. In *Proceedings of the 2001 symposium on Software reusability: putting software reuse in context*, pages 109–117.

[Johnson, 1977] Johnson, S. C. (1977). *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill.

[Kang et al., 1990] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

[Kataoka et al., 2002] Kataoka, Y., Imai, T., Andou, H., and Fukaya, T. (2002). A quantitative evaluation of maintainability enhancement by refactoring. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 576–585. IEEE.

[Kehrer et al., 2021] Kehrer, T., Thüm, T., Schultheiß, A., and Bittner, P. M. (2021). Bridging the Gap Between Clone-and-Own and Software Product Lines.

[Krueger, 1992] Krueger, C. W. (1992). Software reuse. *ACM Comput. Surv.*, 24(2):131–183.

[Lago et al., 2009] Lago, P., Muccini, H., and Van Vliet, H. (2009). A scoped approach to traceability management. *Journal of Systems and Software*, 82(1):168–182.

[Lapeña et al., 2016] Lapeña, R., Ballarin, M., and Cetina, C. (2016). Towards clone-and-own support: locating relevant methods in legacy products. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 194–203.

[Mens and Tourwé, 2004] Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139.

[Meyers and Binkley, 2007] Meyers, T. M. and Binkley, D. (2007). An empirical study of slice-based cohesion and coupling metrics. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(1):1–27.

[Moser et al., 2007] Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A., and Succi, G. (2007). A case study on the impact of refactoring on quality and productivity in an agile team. In *IFIP Central and East European Conference on Software Engineering Techniques*, pages 252–266. Springer.

[Pohl et al., 2005] Pohl, K., Böckle, G., and Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles, and Techniques.*

[Schmorleiz and Lämmel, 2014] Schmorleiz, T. and Lämmel, R. (2014). Similarity management via history annotation. *SATToSE 2014—Pre-proceedings*, page 45.

[Schultheiß et al., 2022a] Schultheiß, A., Bittner, P. M., El-Sharkawy, S., Thüm, T., and Kehrer, T. (2022a). Simulating the evolution of clone-and-own projects with vevos. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022*, pages 231–236.

[Schultheiß et al., 2020] Schultheiß, A., Bittner, P. M., Kehrer, T., and Thüm, T. (2020). On the use of product-line variants as experimental subjects for clone-and-own research: A case study. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*, SPLC '20, New York, NY, USA. Association for Computing Machinery.

[Schultheiß et al., 2022b] Schultheiß, A., Bittner, P. M., Thüm, T., and Kehrer, T. (2022b). Quantifying the Potential to Automate the Synchronization of Variants in Clone-and-Own. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME).*

[Strüber et al., 2019] Strüber, D., Mukelabai, M., Krüger, J., Fischer, S., Linsbauer, L., Martinez, J., and Berger, T. (2019). Facing the truth: benchmarking the techniques for the evolution of variant-rich systems. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*, pages 177–188.

[Tsantalis and Chatzigeorgiou, 2011] Tsantalis, N. and Chatzigeorgiou, A. (2011). Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782.

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

[Vassallo et al., 2019] Vassallo, C., Grano, G., Palomba, F., Gall, H. C., and Bacchelli, A. (2019). A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming*, 180:1–15.

[Wang, 2009] Wang, Y. (2009). What motivate software engineers to refactor source code? evidences from professional developers. In *2009 ieee international conference on software maintenance*, pages 413–416. IEEE.

[Ye et al., 2009] Ye, P., Peng, X., Xue, Y., and Jarzabek, S. (2009). A case study of variation mechanism in an industrial product line. In *International Conference on Software Reuse*, pages 126–136. Springer.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 31. Oktober 2022 Sebastian Wilke