Systematic Reuse and Ad Hoc Forking to Develop Software Variants

by

Ştefan Stănciulescu

Advisor: Andrzej Wąsowski

Co-Advisor: Kasper Østerbye

A dissertation submitted in partial satisfaction of the

requirements for the degree of Doctor of Philosophy

in Software Engineering

at

IT University of Copenhagen,

Rued Langgaards Vej 7, 2300

Copenhagen, Denmark.

August, 2017

Abstract

A common way to implement variability in C/C++ systems is to use preprocessor annotations to allow software to be configurable. Source code is annotated by developers, who can then use preprocessor tools to exclude or include code fragments at compilation time. Although annotations such as *#ifdef* and *#endif* increase efforts in maintenance, clutter the code, and challenge program comprehension, they are easy to use, scalable, and therefore heavily used in practice. An alternative for developing software variants is to reuse existing code by *forking* an existing project. In its simplest form, forking is the process of copying an existing project and then having complete freedom to modify the fresh copy. This is a quick, effective, and common method used in open source and industrial software projects because of its low costs. However, this method does not scale and increases maintenance efforts due to multiple copies of the same artifacts that need to be maintained in multiple places.

In this dissertation I explore the advantages and disadvantages of using these two techniques to develop software variants in practice. The main goal is to understand how to leverage their benefits while reducing the associated costs and issues. I analyze a complex and popular 3D printer firmware and its community, to understand how variants are developed using preprocessor annotations and forking. I explore the idea of combining the two techniques to have the advantages of both, while reducing some drawbacks. As a way to achieve this, I propose and show the feasibility of a variation control system that combines and unifies the usage of preprocessor annotations and forking, leveraging their benefits. Developers work on specific projections (variants) of the code, make changes and push their changes back to the repository. The variation control system also offers support for independent development through forking.

Often, forked variants need to be reintegrated into an integrated platform, though it is not a trivial process. I argue that the integration should be supported by specific tools that are not based on the classical diff tools. To this end, I propose to abstract from code and lift the process to more common *integration intentions*. To achieve this, I developed a set of generic integration intentions that guide the developer through the integration process. Developers have two views showing the differences between the mainline and the forked variant, can make changes that they can see in those two views (while the views are simultaneously synchronized), and can preview the result of their changes and intentions in the preview view. This allows them to explore different integration goals and quickly apply and undo intentions. Through a series of simulations and a controlled experiment, I show that users using a specialized tool do less mistakes compared to traditional merging tool, while they are using just a few operations to achieve their integration goal.

Resumé

Den almindelige måde at implementere variabilitet på i C/C++ er ved at bruge præprocessorannotationer, hvilket tillader software at være konfigurerbart. Kildekode er annoteret af udviklere, som kan derefter bruge værktøjer til præprocessering til at udelukke eller inkludere fragmenter af kode på oversættelsestidspunktet. Selvom annoteringer som #ifdef og #endif øger vedligeholdelsesarbejdet, gør koden mindre læsbar, og formindsker programforståelsen er de nemme at bruge og skalérbare, hvilket gør at de er stærkt anvendt i praksis. Et alternativ til udvikling af softwarevarianter er at genbruge eksisterende kode ved at forke et eksisterende projekt. Forking er forstået som kopiering af et eksisterende projekt, hvilket giver programmøren fuldstændig frihed til at modificere den friske kopi. Denne metode er hurtig og effektiv, og er udbredt i både open source og industrielle software projekter på grund af sine lave Omkostninger. Denne metode skalerer dog imidlertid ikke, og øger også vedligeholdelsesindsatsen fordi at der skal holdes styr på flere kopier af de samme genstande på samme tid. I denne afhandling undersøger jeg fordele og ulemper ved at bruge disse to teknikker til udvikling af softwarevarianter i praksis. Hovedformålet er at forstå hvordan man kan udnytte deres fordele samtidig med at de forbundne omkostninger og proble- mer reduceres. Jeg analyserer en kompleks og populær 3D-printer firmware og udviklermiljøet omkring det, for at forstå hvordan softwarevarianter er udviklet ved brug af præprocessorannoteringer og forking. Jeg undersøger hvordan man kombinerer de to teknikker for at opnå de fordele der findes i begge systemer, samtidig med at prøve at reducere nogle af ulemperne. For at opnå dette, foreslår jeg et softwarevariant kontrolsystem, som gør det muligt at kombinere og forene brugen af præprocessorannotationer og forking, og derved gør brug af deres fordele. Udviklere kan arbejde med specifikke projekteringer (varianter) af koden, foretage ændringer og gemme deres ændringer tilbage i lageret. Softwarevariant kontrolsystemet tilbyder også støtte til uafhængig udvikling gennem forking. Ofte skal forket varianter genintegreres i en integreret platform, selvom dette ikke er en triviel proces. Jeg argumenterer for at integrationen skal støttes af specifikke værktøjer, der ikke er baseret på de klassiske 'diff' værktøjer. Til dette formål foreslår jeg at abstrahere væk fra kode og i stedet benytte en højniveausproces der understøtter almen brugte intentioner til integrering af kode. For at opnå dette, Har jeg udviklet et sæt generiske integreringsintentioner, der styrer udvikleren gennem integrationsprocessen. Udviklere arbejder med to visninger der viser forskellene mellem hovedvarianten og varianten som er forket, de kan foretage ændringer som vises direkte i disse to visninger (som synkroniserer med hinanden på samme tid), og kan se resultatet af deres ændringer og intentioner i en forhåndsvisning. Dette giver dem mulighed for at udforske forskellige integrationsmål og hurtigt anvende og fortryde hensigter. Gennem en række simuleringer og et kontrolleret eksperiment viser jeg, at brugere der bruger et specialiseret værktøj laver færre fejl, når man sammenligner med traditionelle fusionsværktøjer, og samtidig bruger de færre operationer for at nå deres integrationsmål.

Acknowledgments

A Ph.D. is a difficult endeavor and this one is no exception. I would like to thank my advisors Andrzej Wąsowski and Kasper Østerbye for their wonderful guidance. Andrzej has always given me the freedom to choose my path, while offering suggestions and criticism in the most needed moments. Thank you for being patient with me, trusting me, and mentoring me throughout the past 4 years. Many things that I have learned, I owe them to you. Kasper, thank you for always showing and putting things in a different perspective. I tremendously enjoyed our Friday morning meetings, from which I gathered a lot of wisdom. Finally, many thanks to everyone I met at ITU (you know who you are). A special thanks goes to Thorsten. I enjoyed learning and collaborating with you.

I am extremely grateful to for the chance of visiting the Generative Software Development Lab at University of Waterloo in Canada, where I was hosted by prof. Krzysztof Czarnecki. This was the place where I learned many things about research and software engineering, and the most important of all: to question everything and ask questions. I would also like to thank the many wonderful people that I met there during my stay. A sincere thanks goes to Leo and Lorin; I have deeply enjoyed the evenings discussing interesting things and drinking beer. I hope we can do it again!

Life sometimes gives you the most unexpected gifts. I am fortunate to have been given the opportunity to visit Carnegie Mellon University for a longer period of time. This was possible through the EliteForsk Stipendium awarded by the Danish Ministry of Higher Education and Research, to which I am extremely grateful. During my time at CMU, I have met wonderful people with whom I have truly enjoyed spending time and working together. Christian, thank you for having me in your group. Your insights, critical thinking, and always pushing for excellent and insightful results have shaped me as a researcher. Shurui, Gabriel, Chu-Pan and Miguel, thank you for the hospitality and for making me feel welcome there. A big thanks goes also to everyone else that I met at CMU.

My stay in Pittsburgh would have not been the same without a few other people. Specifically, I am grateful for having met Martin, Ivan and Jens, and starting the Pilsner club together with them. I cherish all the moments that we spent together and I am happy that we were able to create such a strong friendship. It was awesome, thank you, and see you soon for a reunion!

The past few months would have been much harder without my amazing and lovely fiancée, Mei. Thank you for your patience, trust, and being with me in these difficult moments. Also thank you for introducing me to the world of clothes hangers. I promise that I will use them from now on, and I am looking forward to having lifetime adventures with you.

Finally, I would like to thank to all my family. My siblings and my parents have supported me and my crazy ideas, particularly when moving to five countries in less than six years. I could have not done it without you. This dissertation is for you.

Contents

List of Figures

List of Tables

1. Introduction

Today's software is becoming increasingly complex requiring even more substantial efforts to develop, maintain and test. Developers have to deliver a high quality product and to cope with increasing demands to provide sufficient security, performance, reliability and allow the software to be configured.

Configurability adds complexity and increases the efforts of development and maintenance. Companies often need to configure their products to the customer requirements. Developers need to build the software with the configurability aspect in mind, mainly to allow their products to reuse as much code as possible. However, this is more costly, harder to get correct and inherently challenging to maintain and develop.

In the past few decades, researchers and practitioners have developed a number of techniques and methods to mitigate this problem. Among these, systematic reuse and ad-hoc reuse techniques have been preferred for developing families of software programs.

1.1 Systematic Reuse

Techniques for systematic reuse have been proposed both in academia and industry to tackle many of the challenges that emerge from the need of developing families of software programs. These techniques include and are not limited to: component-based development [Czarnecki and Eisenecker, 2000], frameworks [Johnson and Foote, 1988], aspect-oriented programming [Kiczales et al., 1997], feature-oriented programming [Prehofer, 1997, Batory et al., 2004], language preprocessors [Kernighan and Ritchie, 1988] and software product lines [Kang et al., 1990, Clements and Northrop, 2002].

A particular successful model, the software product line paradigm, has been shown to decrease costs and efforts of developing families of programs [Dubinsky et al., 2013], while increasing return-on-investment [Hetrick et al., 2006] and the quality of the code [Pohl et al., 2005]. While software product lines deal with many aspects of the business (processes, requirements, feature models), there is one category of software systems that is closely related: highly configurable systems (HCS). These systems are simpler and usually have no explicit feature models, but are widespread

```
119 // LCD selection
120 #ifdef U8GLIB_ST7920
121   //U8GLIB_ST7920_128X64_RRD u8g(0,0,0);
122   U8GLIB_ST7920_128X64_RRD u8g(0);
123 #elif defined(MAKRPANEL)
124   // The MaKrPanel display, ST7565 controller as well
125   U8GLIB_NHD_C12864 u8g(DOGLCD_CS, DOGLCD_A0);
126 #elif defined(VIKI2) || defined(miniVIKI)
127   // Mini Viki and Viki 2.0 LCD, ST7565 controller as well
128   U8GLIB_NHD_C12864 u8g(DOGLCD_CS, DOGLCD_A0);
129 #elif defined(U8GLIB_LM6059_AF)
130   // Based on the Adafruit ST7565 (http://www.adafruit.com/products/250)
131   U8GLIB_LM6059 u8g(DOGLCD_CS, DOGLCD_A0);
132 #else
133   // for regular DOGM128 display with HW-SPI
134   U8GLIB_DOGM128 u8g(DOGLCD_CS, DOGLCD_A0);  // HW-SPI Com: CS, A0
135 #endif
```

Listing 1.1: Marlin excerpt (dogm_lcd_implementation.h at commit a83bf18)

both in open source and industrial environments. In this dissertation I will focus only on highly configurable systems implemented in C/C++.

A common technique used to implement highly configurable systems as well as industrial software product lines [Dubinsky et al., 2013] is the language preprocessor (e.g., C preprocessor (CPP)). In the case of preprocessor, source code is annotated using the tokens **`#ifdef-#endif`**. These annotations define configuration options that, in general, are strings or logical expressions given by developers (e.g., *CONFIG_WIN32 or CONFIG_WIN32 || CONFIG_UNIX*). The end result is an *integrated platform* that contains a single codebase from which different variants can be derived by enabling or disabling these configuration options. Examples of configurable systems include many diverse open source projects: Linux Kernel (operating system), Busybox (Unix tools), Berkeley DB (databases), Marlin (3D printer firmware), Vim (text editor), Apache (webserver), and others, as well as industrial embedded systems. For example, Danfoss's frequency converters contain over 200 features and have more than 1000 configurable parameters [Jepsen and Beuche, 2009], used to derive over 45 variants [Zhang et al., 2013]. HP's configurable software system for printers has been developed for more than 15 years, and can be configured to more than 30 product variants [Mebane and Ohta, 2007]. Other examples include the NASA flight software [Ganesan et al., 2009] and Wilkon's remote control systems [Pech et al., 2009].

In Listing 1.1, an example from the open source project Marlin shows the usage of preprocessor. Marlin is a 3D printer firmware developed in C++ that uses CPP to allow its users to configure the firmware according to their hardware components and preferences.

The code snippet is from the *dogm_lcd_implementation.h* header file, where many variables are defined to be used when the LCD configuration option is enabled. In lines 120, 123, 126, 129 several configuration options are used to annotate fragments of the code. There are in total five configuration options (*U8GLIB_ST7920, MAKR-*

PANEL, VIKI2, miniVIKI, U8GLIB_LM6059_AF) used in only 17 lines of code. For each configuration option, a different constructor is used to set the correct parameters. For example, in line 125 the U8GLIB_NHD_C12864 constructor and two variables, *DOGLCD_CS* and *DOGLCD_A0* (defined in another header file which represent specific pins of the ST7565 controller) are used to initialize the graphical library with the correct values for that particular hardware display. Concretely, from this code snippet five variants can be derived.

Advantages and disadvantages of an integrated platform using the preprocessor

In this section I describe in more details the advantages (+) and disadvantages (-) of using the preprocessor to develop an integrated platform for developing software variants.

(+) Known and used by developers Generally, developers are familiar with the preprocessor (CPP) language. It is used extensively in C/C++ languages, not only to implement variability but also to define macros (e.g., constants) that are later expanded before the code is compiled. This kind of annotaion mechanism usually exists in several other languages than C/C++ via extensions or other ways; e.g., Antenna preprocessor for Java, a subset of CPP for C#, Visual Basic, Adobe Flex, and others [Kästner, 2010].

(+) Easy to use One of the main benefits of using the preprocessor is that it has a low learning curve and it is easy to use, particularly in legacy systems where flexibility may be required. Consider the example where in order to introduce a variant we can annotate a line of code with an #if annotation for the configuration option. Removing the preprocessor annotation is also very simple, by deleting that particular annotation from code.

(+) Flexible Preprocessor annotations are simple to use. They allow developers to apply annotations at token or even character level without needing to respect the underlying language's structure. This makes it a flexible and powerful mechanism.

(+) Scalable Preprocessor annotations can be used to annotate code to be included or excluded at compilation. These annotations can be used in large codebases without having a formal limit on their usage. Therefore, many variants can be derived from the same codebase that uses preprocessor. One such example is the Linux Kernel system comprising several million lines of code and more than 15.000 configuration options that are used in #ifdef annotations throughout its codebase. This is one of the largest

open source projects that successfully uses the preprocessor to implement variability and support the development of a large number of software variants.

(-) Challenges program comprehension One inherent drawback of the preprocessor is that developers work with all the variant space at once. They need to work, understand and modify codebase from which a large number of variants can be derived. The increasing number of variants that need to be supported, managing and evolving configurable software systems using preprocessor annotations poses many challenges. Working on all possible variants of the system at once is known to negatively impact the comprehension of source code [Melo et al., 2016]. Moreover, it is inherently harder to follow the control flow of the program due to the usage of the preprocessor. Code is impacted negatively through the usage of preprocessor, which lessens code quality [Spencer and Geoff, 1992, Favre, 1996, Pohl et al., 2005].

(-) Scattered and tangled code When using preprocessor annotations, the code becomes quickly tangled and clogged by many fine-grained annotations. Although the preprocessor is versatile and flexible, features' implementation code likely gets scattered among files. Separation of concerns becomes difficult if the technique is not correctly managed, as the code becomes more scattered and tangled [Kästner and Apel, 2009, Kästner, 2010].

(-) Subtle errors and little tool support Type errors and syntax errors are also common due to its usage, particularly because of its flexibility that sometimes leads to undisciplined annotations (on tokens or characters and not on AST elements). Developers have a limited set of tools that are variability aware and can support easy development and maintenance of their integrated platform. A few academic tools have tried to address some of the problems, but none have been widely adopted in industry. Beyond syntax highlighting and code folding, no major IDE supports editing variant subsets while ensuring the consistency of the whole system.

Summary

Though there are considerable drawbacks in using CPP annotations, it is still the most common used mechanism to implement highly configurable systems mainly due to its simplicity, low learning curve and easy usage, and being highly flexible.

1.2 Ad-hoc Reuse

An alternative to using an integrated platform implemented via `#ifdef` annotations is to copy an existing project, modify it to comply with the new requirements and

continue to maintain and evolve it as a variant. This process is known as *clone-and-own*, or forking [Faust and Verhoef, 2003, Dubinsky et al., 2013, Rubin and Chechik, 2013a, Stănciulescu et al., 2015].

Advantages and disadvantages of forking

In this section I explain some of the advantages (+) and disadvantages (-) of this technique. There are many advantages and disadvantages that make forking an attractive technique to use on the short term, but that can be extremely costly in the long term.

(+) Reuse tested code In general, forking requires little effort for the initial development, making it an attractive technique. Forking is preferred when reusing existing tested code. One of the main reasons for reusing tested code is that it raises confidence in product reliability [Dubinsky et al., 2013] and potentially the cloned code is more stable [Krinke, 2008]. In some domains, i.e. finance, this is preferred for complex algorithms that deal with financial aspects. Not only that it increases confidence, but it does decrease the time and cost of delivery [Kapser and Godfrey, 2006a] as well.

(+) Easy to use It is a lightweight reuse method that does not require any specialized tools or processes. Cut-and-paste has existed as a method of reuse long before computers, particularly in the area of manuscript-editings. In the era of computers, copy-paste can be executed on almost any kind of artifact, from a line of code or text, to a file, folder or entire project. Software projects that are versioned using version control system can also be copy pasted and further developed as a variant, either by creating a branch or simply by copy pasting the entire versioned repository. Its simplicity is one of the main reasons is a popular mechanism among open source developers and industrial developers alike. It is clearly the right strategy if it is known upfront that the copied artifact will not be merged into its parent.

(+) Quick and cost-effective on short-term Developers use this mechanism even in industrial projects where a systematic reuse policy is generally enforced [Dubinsky et al., 2013], mainly because it is simple, fast and cheap. Forking allows quick prototyping and experimenting to develop a different variant of a software program, without affecting the stability of the main project [Kapser and Godfrey, 2006a, Stănciulescu et al., 2015]. The owner of the clone has now full control over the code, making any needed changes without requiring to not break the code from where it was cloned.

(-) Increases maintenance efforts Forking decreases code quality and increases the maintenance burden [Faust and Verhoef, 2003, Juergens et al., 2009]. For example, a bug found in one of the clones might exist in the system from which the clone originated. A developer must investigate if indeed the bug manifests in the original system,

Integrated platform via #ifdef annotations	Forking
(+) Known by developers	(+) Reuse tested code
(+) Easy to adopt	(+) Easy to use
(+) Flexible	(+) Quick and cost-effective on short term
(+) Scalable	
(-) Challenges program comprehension	(-) Increases maintenance efforts
(-) Scattered and tangled code	(-) Difficult to migrate to systematic reuse
(-) Subtle errors	(-) Does not scale

Table 1.1: Summary of the advantages and disadvantages of integrated platform using preprocessor annotations and forking

and developing a fix for that system as well. Clone traceability is usually not enforced and developers rely on their memory to find the parent of the clone [Dubinsky et al., 2013]. Finally, managing more than a few variants developed with this technique becomes unmanageable due to the increased efforts and costs.

(-) Difficult to migrate to systematic reuse Switching to a systematic reuse mechanism (e.g., an integrated platform) is very challenging. A first step to tackle this challenge this problem is to detect how different the codebases are, how much common code they share and understand how the products will evolve in the short and long term [Kapser and Godfrey, 2006a]. Re-engineering cloned variants poses itself several challenges, particularly as there is little tool support and most of the previous work is tailored for specific projects.

(-) Does not scale Variants developed using forking are usually kept in separate branches of a version control system. This technique does not scale if there are many variants of the system, each of them unique for a customer, as merging them to reduce redundancy is difficult. Version control systems such as CVS, SVN or Git, do not scale to developing many variants (e.g., merging variants, propagating features or bug-fixes). Only very recently commercial version control systems have started considering *'native'* support for product lines [McVoy, 2015].

Summary

Table 1.1 summarizes the advantages and disadvantages of developing software variants using preprocessor annotations and forking.

Preprocessor annotations are in general known and used, easy to adopt. They are flexible with respect to the underlying language and developing software variants is

highly scalable using this technique. On the other hand, they challenge program comprehension, lead for scattered and tangled code, and can introduce subtle errors that are hard to debug, decreasing the code quality. While the main advantages of forking are its simplicity, cost and trust of reusing tested code, it also suffers from many drawbacks: it increases maintenance effort, makes it difficult to migrate the cloned variants to a systematic reuse mechanism, existing tools and version control systems do not support well this method and usually do not maintain any traceability between the clones and the original systems.

1.3 Objectives and Contributions

In this dissertation I have the following two main objectives:

O1. Investigate challenges in the development (evolution and maintenance) of software variants, developed both independently and as a family of programs,

O2. Design, develop, and evaluate tools tailored for evolving, maintaining and integrating variants of software programs, regardless of how these are developed (independently as clones or as a family).

Following these two objectives, this dissertation's contributions are twofold: first, it provides an in-depth understanding of combining preprocessor annotations and forking techniques for developing software variants and the challenges associated with them, and second, it shows that a specialized version control system can leverage some of the advantages of the two techniques, while minimizing some of their drawbacks. Concretely, this dissertation contributes the following:

C1. A vision for developing software variants that proposes to use incremental efforts in preparing the software programs for systematic reuse. Developing software variants can be done either by using forking and incrementally engaging reuse efforts for preparing the system towards an integrated platform and finally to have a software product line. This vision consists of different levels of incremental reuse strategies, including traceability of assets between these levels. The two techniques can be freely used and combined to satisfy the current needs and variants which are to be developed.

C2. A list of empirically supported reasons for using an integrated platform with preprocessor annotations or forking. I analyze in-depth an open source firmware project to understand how preprocessor annotations and forking are combined to develop variants of the firmware. I conduct two interviews with maintainers of the project and survey over 50 developers and users from the ecosystem. We learn that both techniques are used due to their flexibility and simplistic usage, that preprocessor annotations is required for flexibility of including or excluding

code and prototyping small features, but that forking has major drawbacks with regards to propagating changes and keeping an overview of the ecosystem in terms of what functionality exists, where, and who is developing it.

C3. A variation control system prototype that addresses some of the known drawbacks of preprocessor annotations usage, such as working on the complete configuration space (all variants) at once. I present an extended variant of the projectional editing model developed by Walkingshaw [Walkingshaw and Ostermann, 2014], that allows developers to work on concrete configurations on the system, make changes to the code and propagate those changes to specific configurations. By design this editing model has several desirable properties, e.g., consistency of changes. I further show the feasibility of this variation control system by replaying a set of changes from Marlin, using the specific editing workflow.

C4. A tool (INCLINE) for integrating software variants developed from the same codebase into an integrated platform. The tool leverages the idea of projectional views and intentions. Projectional views allow the developer to work on specific variants and to see how one particular integration action will affect the end-result, in a preview view. Intentions are used to provide high level guidance for the developer, by offering straightforward choices (i.e., keep or discard code change) to be applied on the integration scenario. Five intentions were designed and applied on several integration scenarios. Furthermore, through a controlled experiment with graduate students I show that 1) INCLINE produces the correct result and it scales to large files, 2) that the five intentions suffice for most integrations scenarios, and 3) users do much fewer mistakes compared to users using a general diff tool, while being only slightly slower.

1.4 Outline

This dissertation consists of six chapters. Chapters 4-7 are each related to a paper that analyzes in depth the hypotheses and research questions, and provides empirical data to support the findings.

In Chapter 2, I introduce definitions and notations that are used throughout the dissertation.

In Chapter 3, I present a brief overview of research questions and hypotheses, and detail how the empirical evaluations were conducted throughout this work.

In Chapter 4 - paper A [Antkiewicz et al., 2014], I present the joint effort of several researchers, students and myself, of designing the *Virtual Platform* framework that allows to combine forking with other mechanisms for variability, providing several layers of development for increasing reuse and seamlessly advance to an integrated platform.

In Chapter 5 - paper B [Stănciulescu et al., 2015] I describe a study on how preprocessor annotations and forking are used and combined in a real-world open source project to develop variants. The study brings evidence as when developers prefer or are forced to use preprocessor annotations and when they fork to create a variant.

In Chapter 6 - paper C [Stănciulescu et al., 2016a], I present a background on variation control systems, their shortcomings and my variant of a variation control system based on the projectional editing idea by Walkingshaw [Walkingshaw and Ostermann, 2014]. I show the feasibility of using a variation control system to develop and maintain Marlin, by successfully replaying a set of changes from the project's repository.

In Chapter 7 - paper D [Lillack et al., 2017] – submitted and under review –, I describe the idea of using *projectional views* and *intentions* for variant integration. This was developed and implemented using JetBrains MPS IDE. In this chapter I explore how intentions can support integration tasks, and how do they compare against generic diff tools when executing integrations.

In Chapter 8, I present related work in the area of cloning, software product lines and re-engineering variants into a product line, and variation control systems.

In Chapter 9, I present the conclusion and outline future work.

Several papers that I authored or co-authored are not part of this thesis:

- *To connect or not to connect: experiences from modeling topological variability.* Thorsten Berger, Ştefan Stănciulescu, Ommund Øgård, Øystein Haugen, Bo Larsen, Andrzej Wąsowski. In 18th International Software Product Line Conference, Florence, Italy 2014 [Berger et al., 2014b]

- *A technology-neutral role-based collaboration model for software ecosystems.* Ştefan Stănciulescu, Daniela Rabiser, Christoph Seidl. In Proceedings of the 7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, Corfu, Greece, 2016 [Stănciulescu et al., 2016b]

- *Variability Bugs in Highly Configurable Systems: A Qualitative Analysis.* Iago Abal, Jean Melo, Ştefan Stănciulescu, Márcio Ribeiro, Claus Brabrand, Andrzej Wąsowski. In Transactions on Software Engineering and Methodology Journal, 2017 [Abal et al., 2017].

- –Submitted and under review– *INFOX: Identifying Features from Forks.* Shurui Zhou, Ştefan Stănciulescu, Olaf Lessenich, Yingfey Xiong, Christian Kästner, Andrzej Wąsowski [Zhou et al., 2018].

- –Work incomplete– *Detecting (and Preventing) Duplicates in Fork-Based Development Projects.* Ştefan Stănciulescu, Shurui Zhou, Christian Kästner, Andrzej Wąsowski. 2017.

2. Background

In this chapter I introduce the system that I used throughout this dissertation to pursue the objectives set previously. In addition, I shortly present Git and Github, two software systems that are widely used to work with code repositories and which offered strong support for the open source software movement. Finally, I briefly present a calculus that I use in later chapters to describe variational source code formally. This chapter can be read either now or can be referred to when reading the later chapters.

2.1 The Marlin Firmware

Throughout this dissertation, I use the Marlin open source software project as my main subject system to conduct experiments, analyze code, interview, and survey developers, and to answer the research questions formulated in Ch. 3.

Marlin is a 3D printer firmware designed to work with Atmega microcontrollers. It has been created by reusing parts of two existing firmwares, Sprinter and Grbl, to which new original code was added. The firmware computes and controls the movements of the printer, by interpreting a sliced model as seen in Fig. 2.1. The sliced model is designed using a CAD tool.

The project was initiated by one person, Erik Zalm, in August 2011. It gained attention and popularity due to several improvements over Sprinter, and due to the fact that it supports many hardware platforms for 3D printers. Over time, more than 100 developers contributed to the project. Several other firmware projects are inspired by,

Figure 2.1: Printing process in a 3D printer

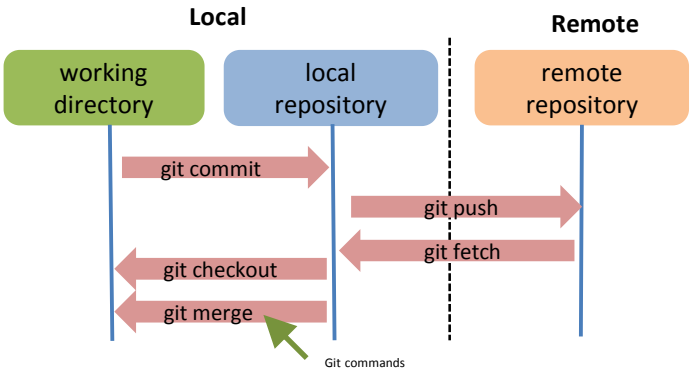


Figure 2.2: Git commands at local level and between local and remote repository

or forked from, Marlin.[1] Besides hobbyists, several manufacturers of 3D printers use the firmware in their products. The 3D printer community uses several forums and IRC channels devoted to Marlin, in which specific Marlin related issues are discussed.

Marlin is flexible enough to deal with different hardware and printer types. It has around 140 features which can be controlled using compile-time parameters. At the time of my initial data retrieval (November 2014), the main Marlin repository contained more than 1500 commits, and it has been forked 1588 times. The high number of forks and the fact that Marlin has explicit variability, makes it a good choice to study our objectives.

Marlin is released under the GPL license. The project and its formal forks are hosted on Github.[2] In early 2015 the repository was transferred to a Github organization MarlinFirmware. In the dissertation I point to both, the new repository MarlinFirmware/Marlin and to the old one ErikZalm/Marlin (presently listed just as a fork of *MarlinFirmware/Marlin*).

2.2 Git

Git is a distributed version control system that allows for creating local repositories, that can be set to point to a remote repository (Fig. 2.2). The local repository can be used at full capacity and capability that Git provides, independent of a network infrastructure or a remote server. Git uses a command-line interface and the user normally works in a directory. To store any changes made, the user has to *commit* the changes made. This results in recording changes that exist in the working directory to the local repository. A commit resembles a Unix patch, and adds a message with the description of the change. Commits are identified by computing an SHA-1 value of the change. Git

[1] `http://reprap.org/wiki/List_of_Firmware`
[2] `https://github.com/MarlinFirmware/Marlin`

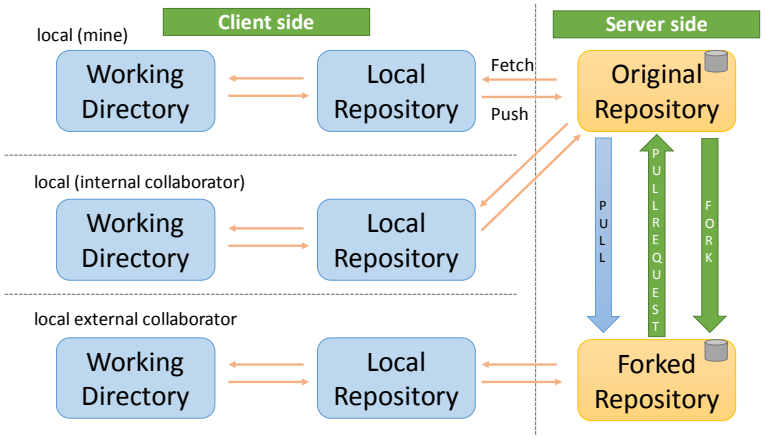


Figure 2.3: Client and server repositories. Internal and external collaborators, using original and forked repositories.

employs a lightweight and simple branching system, with each branch being nothing more than a label attached to a commit. Merging branches in Git is fast and has several heuristics to deal with conflicts. Heavily merging of many trees is done successfully, for example, in the Linux Kernel project, for which Git was originally developed.

When needed, changes can be pushed to the remote repository via the *git push* command. The remote repository resembles a copy of the local one; it stores commits and branch labels and contains all the data that exists in the local repository.

2.3 Github

Github is a hosting service for Git repositories, offering a platform for collaborative development. Unlike Git, which is a command line tool, Github is a web-based graphical interface for code management. It offers facilities such as wikis, task management, bug-tracking, and small scale social media activities such as following users or repositories. Most importantly, Github allows for copying repositories in a structured way. This mechanism, known as *forking*, creates a traceability link between the copied repository, *the fork*, and the original project. The usual workflow is presented in Fig. 2.3. An original repository resides on the server side (e.g. Github). The owner of that repository can retrieve the repository and store it on the local machine by using the *git clone* command shown in Fig. 2.2. From thereon, the owner of that repository can work locally on his computer without accessing the server repository. When working in a team the original repository is shared between multiple internal collaborators. If an external collaborator wants to contribute with patches, the collaborator must *fork* the original repository. For example, Fig. 2.4 shows a fork of the Marlin project. This operation creates a new repository on the collaborator's account, and links the forked

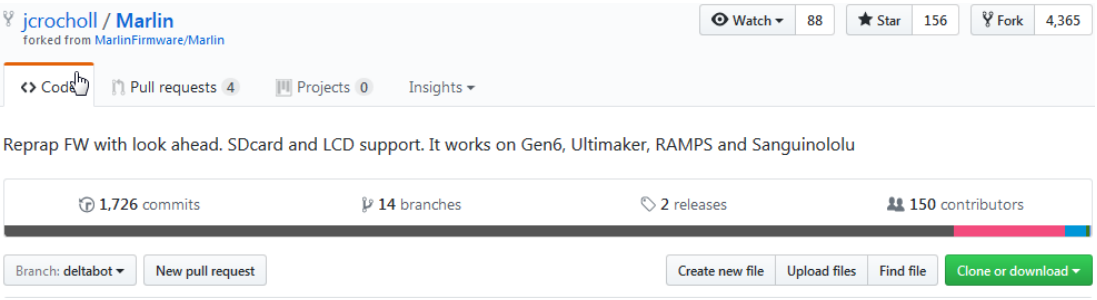


Figure 2.4: Fork of the Marlin project in which the Delta variant was developed. On top left corner there is the tag that this repository was forked from MarlinFirmware/-Marlin, which indicates that this is a formal fork. In comparison, in Fig. 2.5 this tag does not exist, but we can quickly deduct that the repository is an informal fork of Marlin.

repository to the original one. The forked repository has a forked-from relationship with the original repository. The owner of the fork can create a *pull request* from the fork to the original repository. A pull request is similar to a change request as it is known in software configuration management (SCM) [Babich, 1986]. In its simplest form, a pull request consists one or several commits, a comment describing the change. The pull request can be itself commented upon, updated, and accepted into the project's repository.

A pull request can be created either in the same repository, e.g. to allow a team to discuss the change, or from a fork to the original project. Each pull request consists of a description, possible comments from users and a set of commits. A pull request can be merged *automatically* if there are no merge conflicts detected by Github's pull request service, otherwise it must be merged manually. If a pull request cannot be merged automatically, it is often required by the maintainer of the repository to fix the conflict on the fork side by pulling the latest changes from the source and then updating the pull request.

Definitions

Repository. A repository is a structured storage for a project. The content is organized by a version control system.

Branch. We refer to a branch as being a line of development that has a label. In Git's terminology, a branch is just a pointer to a commit, and it has a label attached to the tip of the branch. In practice, a branch is used to separate lines of development, e.g., developing a variant of a software program. In this respect, forking and branching are usually used for the same purpose, but in this dissertation I focus on forks as defined in this chapter.

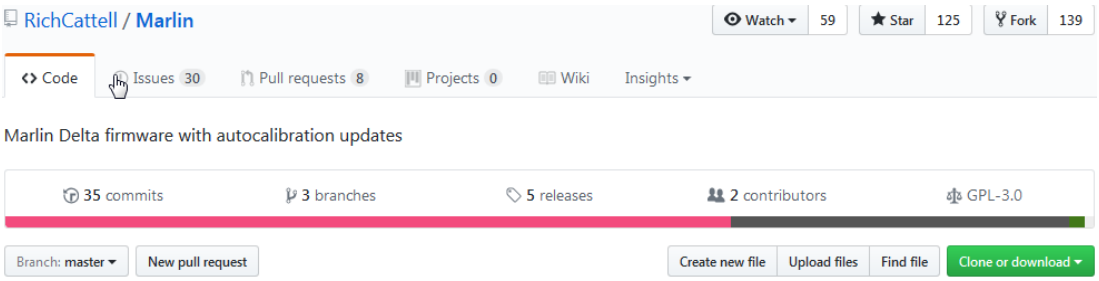


Figure 2.5: An informal fork of Marlin to support Delta printers with autocalibration. This was done by simply copy pasting the code and creating a repository from scratch, which was then pushed to Github.

Commit. A commit is an atomic change that was applied to a repository. It uses a similar syntax to UNIX patches, with a message attached that describes the change.
Fork. A fork is a copy of a project created by cloning in-the-large.
Formal fork. A formal fork is a fork that has been created using Github's forking mechanism.
Informal fork. An informal fork is a copy of an existing repository created simply by copying files elsewhere, without any automatic traceability links.
Active fork. An active fork is a fork that has either synchronized with the origin after its creation, or has had changes applied to it. An *inactive* fork displays no activity in the repository after the creation date.
Variant. A variant is a project that was cloned and modified to satisfy certain requirements. Variants can also be created by derivation from an integrated platform, given a configuration.
Pull request. A pull request is a change request that contains commits and information about the change. A pull request can exist in one of the following three states: *open*—when the pull request is created and awaits to be accepted, *merged*—the pull request is accepted into the target repository, and *closed*—the pull request has been rejected.

2.4 Feature Models

A feature model is a representation of the features that exist in a system, and the relations between them. Feature models were first introduced by Kang et al [Kang et al., 1990], and they are usually represented using feature diagrams. However, using feature diagrams for systems that have thousands of features is infeasible, thus a textual representation is usually used for such systems (e.g. Linux has its own variability language, Kconfig [She et al., 2010]). A feature model consists of a AND\OR tree and cross tree constraints. A simple feature model of a fictional phone product line is presented in Fig 2.6. The filled black dot means that the feature is mandatory, the unfilled dot

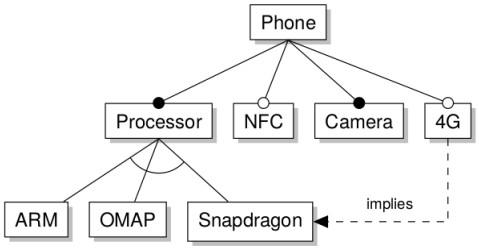


Figure 2.6: Feature model with cross tree constraints.

represents a feature that is optional, and the arc between the three subfeatures ARM, OMAP, and Snapdragon and their parent, Processor, represents an exclusive OR (only of the three subfeatures can be selected).

2.5 Choice Calculus

Choice calculus is a formal notation for specifying variability [Erwig and Walkingshaw, 2011a, Walkingshaw, 2013]. It is a simple and concise notation that I use to describe formally code with preprocessor annotations. I use the choice calculus notation in later sections to describe the Variation Control System and show examples.

For example, Fig. 2.7 shows a simplified snippet of the **`#ifdef`** block from Listing 1.1 (top) and its representation as a choice calculus expression (bottom). As the reader can observe, choice calculus is a more compact and concise representation. Similarly to **`#ifdef`**, choices can be nested. An important aspect is that using choice calculus I can specify meta-operations on it, for example rewriting or editing the tree, which are useful later in the dissertation. The code can be resolved as following: if $U8GLIB\_ST7920$ is set to $true$, the choice will resolve to `U8GLIB_ST7920_128X64_RRD u8g(0);` during configuration. If $U8GLIB\_ST7920$ is set to $false$, then it will resolve to `U8GLIB_DOGM128 u8g(DOGLCD_CS, DOGLCD_A0);`. Finally, complex presence conditions can also be expressed in choice calculus, as I will show next.

Figure 2.7: **`#ifdef`** block (top) and its corresponding choice expression (bottom), where the numbers 2 and 4 represent the strings at those lines in the top box.

```
1 #ifdef U8GLIB_ST7920
2   U8GLIB_ST7920_128X64_RRD u8g(0);
3 #else
4   U8GLIB_DOGM128 u8g(DOGLCD_CS, DOGLCD_A0);
5 #endif
```

$$A\langle 2, 4\rangle$$

Figure 2.8: Choice calculus syntax

$$
\begin{array}{rcll}
B & ::= & I \mid \mathit{true} \mid \mathit{false} & \\
F & ::= & B \mid \neg F \mid F \vee F \mid F \wedge F & \\
e & ::= & F\langle e, e\rangle & \textit{Choice} \\
 & \mid & e \cdot e & \textit{Append} \\
 & \mid & a & \textit{Token} \\
 & \mid & \iota & \textit{Identity} \\
I & ::= & \mathit{String} & \textit{Identifier}
\end{array}
$$

The choice calculus' syntax used in this dissertation is presented in Fig. 2.8. The metavariable e denotes a choice calculus expression (i.e., code). An expression can consist of a choice, the concatenation of one expression to another using the monoid append operation ($\cdot$), an arbitrary token, or the monoid identity element (ι). Usually, tokens are arbitrary strings, the append operation is string and line concatenation, and the identity element is empty string. Throughout this dissertation I will consider tokens to be arbitrary strings.

A choice represents a variation point in-place as a *choice between alternatives*, written $F\langle e_1, e_2\rangle$. The associated condition F is the choice's presence condition. When configuring a choice calculus expression, each feature is set to true or false, then each choice is replaced by either alternative e_1 if F evaluates to true, or alternative e_2 otherwise.

3. Methodology

In this dissertation, I followed a two step method. First, I studied how preprocessor annotations and forking are used to develop cost-efficient software variants. Second, I designed, developed and evaluated tools that were inspired based on insights obtained in the first step.

In the following part of this chapter, I will detail the research method, starting with the research questions and hypotheses.

Research Questions

The following two research questions drove my research forward and led to the contributions mentioned in Sec. 1.3.

RQ1 *What are the main challenges and advantages of preprocessor annotations and forking to develop software variants?*
The goal of this research question is to analyze in-depth how these two techniques are used for developing software variants, from the perspective of their advantages and disadvantages. Importantly, these two techniques are not used together usually, but are independent of each other. Understanding the interplay of these two techniques will lead to developing a set of requirements for designing and developing a system to support the development of software variants, maximizing the advantages of the two techniques. This is directly related with the first objective of this dissertation(O1.).

RQ2 *How can a specialized variation control system leverage the advantages of preprocessor annotations and forking to support the development, maintenance and integration of software variants, while minimizing some of their drawbacks?*
This question is aimed at providing evidence that a variation control system has the potential of improving on some drawbacks of the preprocessor usage and forking, while retaining the benefits of both. The second objective O2. of this dissertation relates to this research question.

Hypotheses

Previous work and industrial projects point us to the importance of having flexible, straightforward and cost-efficient mechanisms for developing software variants [Duc et al., 2014, Dubinsky et al., 2013]. To support and investigate the research questions defined previously, I formulate the following hypotheses.

H1 : The preprocessor and forking are used and combined by developers due to their flexibility, low cost, and straightforward usage.

H2 : A variation control system can 1) support the editing of configurable systems and combine the preprocessor and forking in a uniform way, and 2) with specialized integration support forked variants can be re-integrated into the integrated platform.

Method

To analyze and evaluate how the preprocessor and forking can be combined to develop, maintain and integrate software variants, and evaluate the tools developed in this dissertation, I use several techniques.

To test *H1*, I study the usage of the preprocessor and forking on an open source software system. I used the 3D printer firmware Marlin, that uses preprocessor and forking to develop variants. I execute the study using mixed methods. I combine artifact studies, studying history of the project and understanding how forks contribute to the project, as well as *surveying* developers from that community and *interviewing* maintainers of the project [Stănciulescu et al., 2015]. I run two surveys targeted at active and inactive fork owners of Marlin (see Appendix A.5 and Appendix A.6), after analyzing the forks of Marlin. For the interview part, I conduct two semi structured interviews with two maintainers of Marlin, in open writing.

To test *H2*, I design and implement a variation control system prototype with forking support. I conduct an empirical study to understand the edit patterns that occur when developers do variability-related edits. I then re-execute a number of variant edits using the prototype tool, and show that it is feasible to use the tool to maintain and develop a configurable system.

To test the integration part of *H2*, together with colleagues I designed and developed a dedicated integration tool with UI support (INCLINE). We run a series of simulations with real integration scenarios to test the completeness, correctness and the scalability of the tool. We then conduct a controlled experiment with students to assess the capabilities of INCLINE in a realistic setting. In the controlled experiment, each participant performs two tasks, using two treatments: Eclipse CDT, and INCLINE(INC) on two programs, in a random order to reduce learning effects.

4. Flexible Product Line Engineering Paper A

This chapter presents a vision idea that emerged during a workshop that I attended, held at University of Waterloo in 2013, soon after I started my Ph.D. studies. The outcome of this workshop was an idea paper that was a joint effort of several professors, researchers and students, including myself. Although I am not the main author of this work and my contribution is minor, this initial work has been the foundation stone upon which I built my research. Therefore, I think it is worth presenting it as a context for my research, while the results presented in this dissertation are one attempt at realizing this vision. Some parts of this chapter are borrowed from the following paper "Flexible Product Line Engineering with a Virtual Platform" [Antkiewicz et al., 2014], and I adapted them to align with the goals of this dissertation.

4.1 Introduction

Developing variants of software products is necessary to satisfy different requirements, to adapt products to different geographical areas, and to support different usage conditions. Very often, product variants are created using forking, to reuse existing code or variant. As I have described in Ch. 1, both forking and the integrated platform through the usage of preprocessor annotations have significant advantages. In this chapter, I present a vision on how to have the benefits of an integrated platform but without requiring the high-risk transition processes, while retaining the flexibility and benefits of forking. This vision strives to explore the two development models and the tools needed to reconcile both methods.

Virtual platform [Antkiewicz et al., 2014] is an idea of a framework for supporting the mixed development of variants, not unlike in the presented case study of Marlin firmware. Virtual platform intends of using integrated variability and forking as the mechanisms for reuse. The main idea is to use explicit traceability for reusable artifacts that can be shared between projects. Moreover, the framework should support automatic and manual propagation of changes in a controlled process. The frame-

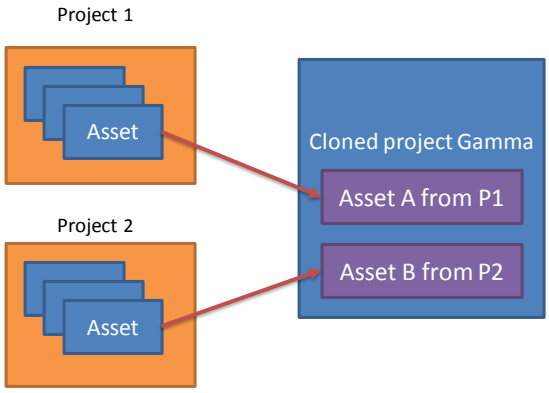


Figure 4.1: Recording traceability

work enforces traceability between clones, features and other artifacts using specific meta-data, the assets being, initially, distributed across projects.

The novelty of the virtual platform consists not only in supporting both development styles, but also supporting a transition from fork-based development to a software product line. The idea is to aggregate existing source code into one integrated platform from which different variants can be generated. The key point is that integrated platforms are usually mature architectures, while ad hoc forking may lead to an ad hoc organization of variability. We distinguish between several adoption levels, which are suitable for different phases (maturity levels) of a project's lifecycle with intensive variability.

4.2 Adoption Levels

The levels of adoption are meant to guide an organization on deciding when and how to invest more into reuse. We distinguish six adoption levels, and the standard forking mechanism.

Standard Forking. This is the level when organizations do not have a strategy for reuse, and develop related projects by freely copying and modifying existing ones. Consider an organization that has one initial project, Project 1 and Project 2. The two projects themselves are developed via forking. The organization has to deliver a new product, project Gamma, thus it forks Project1, and copies some assets from Project 2. This ad-hoc reuse mechanism does not enforce any specific reuse or traceability. The user is left with all the decision on what assets to copy and reuse. The advantages of forking are well understood [Dubinsky et al., 2013, Kapser and Godfrey, 2006a].

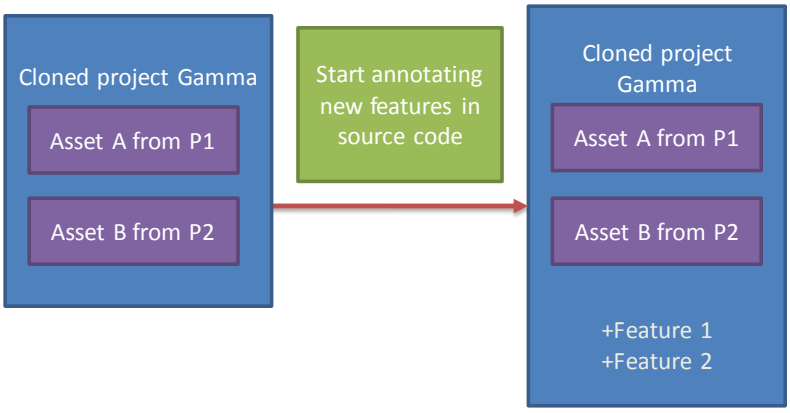


Figure 4.2: Annotating and adding new features in a clone

However, if not carefully managed, forking cannot scale and increases the costs of developing and maintaining the related projects. Ad hoc forking can be successfully used if the reuse frequency is low and maintaining consistency among assets is not important.

Level 1. Forking with traceability. This level is the state when an organization starts keeping traceability of assets that are being copied and their origins. Providing just simple traces of which assets were copied and their origin, can improve maintenance of the clones (see Fig.4.1). Using traceability links reduces efforts for propagating extensions and bug-fixes to cloned assets, as information about the origin of the clone is kept. Developers can notify cloned asset's owners about changes and modifications done to the artifacts. The notified teams can decide if they need the change or not, and act accordingly.

Level 2. Forking with features. This level is the state when organization are developing features with focus on reusability. An organization may need to add new features that will be later reused in other projects (see Fig.4.2). Now it should be the appropriate time to invest more efforts in developing reusable features. Development teams declare features and map them to source code fragments that implement them by using preprocessor annotations. Teams benefit from a better overview of the functionality, but also can propagate features easier as the relevant fragments can be located easily.

Level 3. Forking with configuration This level is the state when teams add the capability to enable or disable features, and derive a variant through feature selection (see Fig.4.3). Feature constraints can be added to exclude invalid combinations. This is suitable when frequent derivations of similar variations are important, together

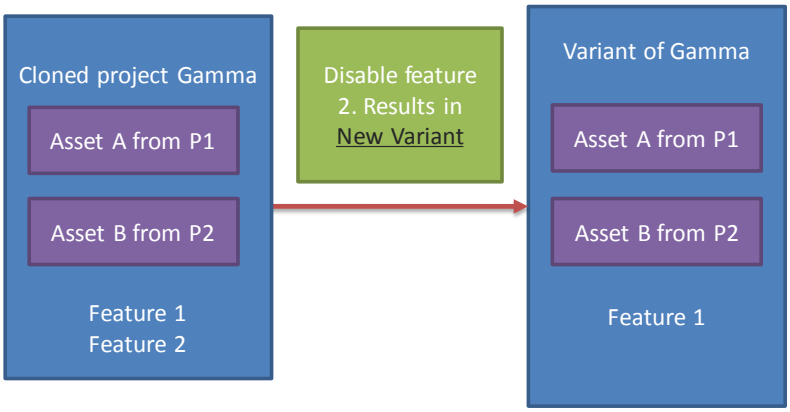


Figure 4.3: Derivation of variants by selecting/deselecting desired features

with maintaining consistency between projects. On the other hand, deriving variants reduces the need for forking, and encourages more systematic reuse.

For example, the Marlin ecosystem can be seen to exist at three levels: L1, L2 and partially L3. The traceability links that are enforced in L1 are present in Marlin, but at a much coarse granularity level (repository) than L1 suggests. Marlin is however much better integrated with L2 of the virtual platform, as it uses conditional compilation to annotate features in the source code. L3 refers to being able to create new variants by simply selecting or deselecting certain features. While this capability exists in the main Marlin project (e.g., derive a variant from the main codebase), it is not possible to *fork* a project with only some features selected. Therefore, users do copy paste features implemented in other repositories.

Level 4. Forking with feature model This level is the state when an organization creates a central feature model. The feature model covers all projects and all the implement features. New projects are created by reusing an existing project, and then propagating any needed features from other projects as allowed by the central feature model. The idea here is to have distributed assets among different projects, but to centralize the information regarding those assets.

This level is appropriate when frequency of reuse becomes high, thus facilitating the developers on creating new projects by taking an existing one as basis, and adding the needed features as allowed by the central feature model. The main disadvantage is that the feature model has to be maintained. However, this is an important step in realizing an overview of what functionality exists and specifically, features that are

shared between projects. This information is beneficial for the next step in the adoption ladder, to create an integrated platform and its respective feature model.

However, there are many systems that do not have a formal description of their features, for example a feature model. Marlin also does not have one, less alone a feature model that represents its forks. Although several solutions have been proposed to this problem [She, 2013, Bécan, 2016], extracting feature models from source code is not the main focus of this dissertation.

Level 5. Product line with integrated variability and forking This level is the state when a company uses both the integrated platform as well as forking, to maintain the core assets and respectively to customize features or functionality for specific users in forks. Need of frequent reuse accelerates the process of developing the platform. The platform increases scalability, reduces redundancy and improves change propagation, and allows to derive products from it. Moreover, forking is still used as in the previous levels or to support customer specific requirements. Both flexibility and systematic reuse is achieved at this level.

As a remark, migrating to a product line is a challenge that has been researched to some extent [Hetrick et al., 2006, Jepsen et al., 2007, Jepsen and Beuche, 2009]. In this context, as a project with a lot of functionality developed and maintained in forks, Marlin is an interesting project. Researchers can use the Marlin ecosystem to develop and test techniques, methods or tools for advancing the state of the art in re-engineering and migrating legacy systems to software product lines.

Level 6. Full product line This level is the state when an organization has a fully integrated platform, consisting of core assets and variable assets, as well as a feature model. These systems are called highly configurable systems [Kästner and Apel, 2009]. An example of a highly configurable system is the Linux Kernel.

While having a fully integrated platform is desired, in practice, there will often be forks of highly configurable systems where new functionality is developed, either for management reasons, for technical reasons or to experiment with the codebase. The added benefit of doing some development in forks of the system is that it maintains the integrated platform stable.

4.3 Marlin, Github and Virtual Platform

The Marlin firmware together with the hosting and usage of Github as a collaboration kit, resembles to some extent the idea of the virtual platform. However, there are a few improvements that virtual platform has in comparison.

1. Github's forking only allows to clone at the repository granularity. The virtual platform improves this by allowing to clone assets and imposing a traceability

link at a finer granularity than repository as in case of Github's forking. Improving traceability is a first step on having a better management of incoming and outgoing changes, and change impact analysis.

2. Decentralization of information is a problem in Github due to the large amount of forks created. The virtual platform proposes to improve decentralization by creating a central feature model of the existing projects. While this may be a costly investment, it is needed for successfully realizing the integrated platform, and it tackles the decentralization issue. Moreover, having an overview of the existing functionality at the moment, can reduce redundant development that is present both in open source and industrial projects [Berger et al., 2014a, Dubinsky et al., 2013, Duc et al., 2014].

3. Propagating changes and bug-fixes to Marlin's forks is low, as I will show in Ch. 5. The virtual platform addresses this issue first by having a finer granularity in traceability links between clones (and their assets). As such, these traceability links can be used by an automatic or semi-automatic change propagation system that automates the transfer of changes between clones. Schmorleiz et al. (initial collaborators of virtual platform) have created a tool that can synchronize fragments of code that are similar, using several types of annotations for specific use cases (maintaining clone in sync, allowing some differences between clones, restoring equality of one clone to its linked asset etc.) [Schmorleiz, 2015]. Second, by combining preprocessor annotations and forking in a unified system, we can avoid this synchronization of changes problem, as I will show in Ch. 6.

Summary

The virtual platform is an idea of a decentralized platform with distributed assets. One of the goals of the virtual platform is to incrementally increase reuse throughout the project's lifecycle.

Increased reuse can be achieved through the guidance of several adoption levels. These levels guide towards reaching an fully integrated platform, that consists of core and variable assets. Products can be derived from the integrated platform.

In this dissertation I explore two things: 1) combining the integrated platform and forking to enhance variant development (L1+L2), and 2) support the integration of forked variants or features into the integrated platform.

5. Practices of Systematic and Ad hoc Reuse Paper B

This chapter shares content with paper "Forked And Integrated Variants in an Open Source Firmware Project" published in ICSME 2015 [Stănciulescu et al., 2015]. For complete details, refer to the paper in Appendix A.2

In the previous chapter I introduced a framework for developing software variants using both an integrated platform and forking. These two techniques are often intermixed, both in open source systems and in industrial projects. In this chapter, I present a case study to understand 1) how these two techniques are combined, when one is used in detriment of the other, or why both of them are used by developers, and 2) to derive detailed requirements for developing tools to support this development method.

5.1 Introduction

Historically, forking had sometimes a negative antisocial connotation. It denoted sometimes a community schism, when a project is split and an independent development starts in a diverging direction [Raymond, 1999]. The term has acquired a less negative meaning since the arrival of distributed version control systems and, in particular, of collaborative and code hosting platform, Github. In Github's terms, a fork is a new repository that is traced as being a fork of an existing repository from some other user than the fork's owner. Github's forking mechanism introduced traceability and easy propagation of commits between the fork and fork's main parent. The fork's owner has full control over this duplicated repository (the fork). Any commits pushed to the fork can be further propagated via a pull request to the parent of that fork. This pull-based development mechanism allows developers to easily collaborate, review changes, and gives them full control over their own forks but also over the main project's roadmap.

The main goal of forking is to reuse code by creating a copy of an existing repository. Reusing code decreases costs and allows for customization, for example to create a specific variant for a specific customer. The fork is traced back to the repository

from which was copied, which is important for long-term documentation and traceability. Therefore, forking can be seen as a software reuse mechanism next to established examples such as object-oriented reuse patterns, aspect-oriented programming, or software product line architectures.

In this chapter I investigate how preprocessor annotations and forking are combined to develop software variants. Specifically, I am interested in validating *H1 — The preprocessor and forking are used and combined by developers due to their flexibility, low cost, and straightforward usage*; and answering *RQ1 – What are the main challenges and advantages of preprocessor annotations and forking to develop software variants?*

Due to the fact that *RQ1* is formulated as a broad research question, I split the question into several questions, each targeted to one concrete topic of interest:

- What are the criteria to introduce variants using preprocessor annotations instead of forking?
- What are the main reasons for creating forks?
- How do ongoing project development and maintenance benefit from existence of many forks?
- Under what circumstances is forking preferred over integrated variability for creating and maintaining variants?
- What other challenges arise in fork-based development projects?
- What are the criteria that lead to integrating a forked variant into an integrated platform using preprocessor annotations?

To answer these questions, I investigate forking practices in Marlin, an open source 3D printer firmware project. Marlin is an appropriate study subject due to an unprecedented amount of forks created in a very short period. The Marlin project has been forked over 4000 times (1588 times at the time of the original study, in the period of 3 years and 3 months). Also, the project itself was created by reusing parts of two existing firmware projects, Sprinter and Grbl, to which new code was added. The firmware computes and controls the movements of the printer, by interpreting a series of codes specific to CNC machines. It has about 140 features, which can be controlled using compile-time parameters.

The Marlin project is of interest in this context as it uses preprocessor annotations (in the role of the integrated platform) and intensive forking (for more ad hoc volatile changes). As such, Marlin and Github can be seen as a spartan prototype of the virtual platform idea.

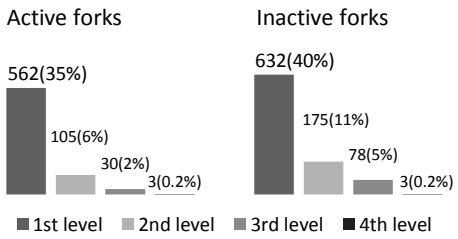


Figure 5.1: Marlin's active and inactive forks, and forks' levels (percentages show the relative size in the set of all forks). The 1st level forks were created by forking main Marlin repository, the 2nd level forks were created by forking 1st level forks of Marlin, and similarly for 3rd and 4th level.

5.2 Study Design

To get insights why users use preprocessor annotations and forking, I proceed in two steps. First, I classify the forks into two categories; purpose of the fork, and fork activity and nesting depth (Fig 5.1). Commit messages (using key word search) of all forks are used to classify the forks by their main purpose (i.e., why a particular fork has been created). To obtain qualitative data, I analyzed rejected pull requests to retrieve information about reasons for rejecting proposed changes from forks.

Second, I developed two short surveys directed towards active (made changes after forking) and inactive (no changes since forking) fork owners (see Appendix A.5 and respectively Appendix A.6 for the actual surveys). The surveys contained both closed and open questions. They were available for ten days, after which were closed. The survey specifically asked for the reasons of using preprocessor annotations and forking to develop Marlin; what are the challenges encountered in getting their patches accepted; why they do not retrieve changes from Marlin and if they do it, how often they synchronize. This gives a broader view of the development and maintenance practices in Marlin, from the developers' perspective.

I distributed the survey invitation via email to 336 developers that had public email addresses (the others do not have an email address linked to their Github account), 185 belonging to owners of active forks and 151 to inactive ones. The response rate was 18.3% (34 respondents) for active fork owners, and 15.2% (23 respondents) for inactive forks. Finally, I interviewed two active maintainers of Marlin (in writing, open answers).

The database, survey questions, and other artifacts are available at `http://bitbucket.org/modelsteam/2015-marlin`.

5.3 Results and Analysis

What are the criteria to introduce variants using preprocessor annotations instead of forking?

This is a question I ask in the context of Marlin, as the community uses both the preprocessor and forked variants. The accumulated experience sheds light on the choice between the two mechanisms. I approach the question by qualitatively analyzing forks that developed changes involving preprocessor annotations, excluding those that modify only the configuration files (configuration changes involve changing preprocessor annotations, and are the most common type of changes among users). This data is supplemented with answers from the survey and interviews with the maintainers. The answer to this question is presented in the following observation box.

> *Marlin developers prefer preprocessor annotations over forking in following situations:*
>
> *T1. The flexibility to use several variants is needed.*
>
> *T2. Project maintainers require conditional compilation for new features submitted to main Marlin repository.*

In the following text, I present in detail each of the finding and the data that supports it. There are 261 forks (37% of active forks) that introduced preprocessor annotations in their commits. Preprocessor is used in these forks for the same reasons as in other system level software [Berger et al., 2013]: to reduce use of memory, to disable functionality that is incompatible with current hardware, or to control inclusion of experimental code. Seven out of 11 developers report use of the preprocessor for managing memory limitations (e.g., *Not all boards have enough space to run all the features so my feature was only compiled into larger chips*). Marlin supports printers based on 8-bit ATmega micro-controllers that have limited flash memory, usually 4–256kB. In general, developers use preprocessor to guard functionality that is optional, allowing it to be switched off either for themselves or for other users. Hence, flexibility of integrated variants (T1) is needed to meet memory requirements of different use cases and different hardware in the same fork. This aligns well with previous work [Kästner and Apel, 2009] that acknowledges the advantage of using preprocessor for its fine granularity.

Interestingly, also forks that do not contribute any changes to main Marlin do still use the preprocessor. Only 83 out of the 261 forks using this mechanism actually created pull requests (32%). Through an exploratory analysis of some of these changes, it is clear that the developers might have needed the flexibility for their own printers or to experiment with the code (T1). For example, malx122/Marlin - commit #69052359 added

support for a second serial communication using preprocessor annotations to experiment on a customized version of Marlin intended to be used on a RepRap printer.

Another interesting aspect of using preprocessor annotations is that if a developer plans to contribute a feature to the main repository, Marlin's coding guidelines must be followed. One of the main project maintainers states in the interview: *Every new feature contribution requires conditionals in some amount. New features that don't do this will be deferred until they do* (T2). For example, the pull request P594, was initially rejected because it did not properly consider the preprocessor annotations. Obviously, the Marlin maintainers enforce this rule, because the broad Marlin community needs the flexibility prescribed by criterion T1 and it is also a mechanism to protect existing users from installing on their printers unnecessary features (new features are disabled by default in the configuration files).

Although there is a general agreement among developers for the reasons of using preprocessor annotations, one mentions that this mechanism makes the code hard to read: *I didn't have time to re-architect their code*, thus the usage of preprocessor annotations. In this system, preprocessor annotations are heavily used throughout the code in an ad hoc manner, challenging program comprehension and increasing the overall complexity of the system. Recall the example from Listing 1.1, where there are more annotations than actual code.

What are the main reasons for creating forks?

This questions is important for understanding the main reasons for creating forks. The previous question looks at using the two main techniques from the technical perspective, while this question switches focus on understanding why forking is used. Due to the magnitude of hundreds of forks, classifying the forks' purposes is not possible through a manual analysis. Therefore, I establish the purpose of forks through a term-based classification of forks' commit messages. I initially group the forks into two categories; purpose of the fork, and fork activity and nesting depth (Fig 5.1). Commit messages (using key word search) of all forks are used to classify the forks by their main purpose (i.e., why a particular fork has been created). To verify the classification, I randomly sampled 40 active forks and checked if the changes in the commits correspond to the automatic classification. The results show over 94% match between the manual classification and the automatic one, which indicates that the data from Fig. 5.2 is reliable. In addition, I substantiate and complement the results with a quantitative questionnaire distributed to fork owners.

Heuristic classification of forks. The classifier was run on 700 active forks (Fig. 5.1), and I used the following terms for classifying forks in three categories:

- *Configuration forks* are forks that change the configuration of the firmware. Firmware users simply make changes to the configuration files of the firmware,

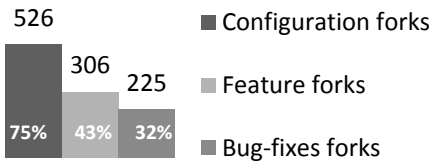


Figure 5.2: Number of active forks used for configuration, developing features or bug-fixes (overlapping categories, percentages show the relative size in the set of active forks)

according to their needs, and push their changes to their own fork. These are detected by checking if the main configuration files (*configuration.h* and *configuration_adv.h*) are modified.

- *Development forks* are used to add new functionality. These are detected by matching the following search terms: *add, support, feature, new, added, implemented*.
- *Bug-fix forks* are used to fix defects. These are identified using these terms: *corrected, fix, bugfix, bug, fixed, replace*.

Commit messages of a single fork may match search terms of more than one category. In such case the fork is classified as belonging to all matching categories.

Figure 5.2 summarizes the results of the heuristic categorization. As many as 75% of active forks do changes to their configuration files. This is needed to be able to run the firmware on each user's hardware and printer, according to their needs with respect to feature or parameters. Detailed inspection shows that some need to change the configuration to test a functionality that they are working on. Examples of configuration forks include: 3DPrintFIT #6343044c, jmil #04b8ef41, and Makers-Tool-Works #651b99d1. Using Github and the forking mechanism to store configurations of the system is somewhat surprising. Nevertheless, it is the principal reason for forking in Marlin. The other two categories are expected and cover development of new functionality (43%) and bug-fixes (32%).

Surveying Developers To validate the data from the heuristic classification, I asked fork owners what were the reasons for creating their forks. Among the active fork owners, 62% (21 responses) report that they had originally intended to configure the firmware. This fraction is smaller than 75% of the actual number of configuration forks (Fig. 5.2). The intention to just configure the firmware was even more dominating among inactive fork owners (74% or 17 responses). The situation is the opposite for the non-configuration forks: 68% of active fork owners report that their intention was to contribute new functionality or modify the existing one. This is more than the

actual number established heuristically: 54% of forks are used for new functionality or bug-fixes, which is the sum of the last two columns in Fig. 5.2, corrected for the intersection.

> *Forking is used to develop features and fix bugs, but also to store variant configuration data, which makes it a lightweight and effective configuration maintenance mechanism.*

This confirms partially *H1*, that is, forking is a simple, cheap and easy to use mechanism for starting the development on a software variant.

Maintaining variant configurations in forks of entire projects is a very simple and effective mechanism. It does not require specialized configuration management or variability management tools. The fork owner has a reliable backup copy of the configuration, and the configuration can always be easily reconciled with upstream, if that becomes desirable. The Marlin community is extremely successful using this mechanism for the purpose. As mentioned above, some developers end up following this practice, even though this is not what they initially expected. In general, storing configurations of systems as forks or in branches is not an established technique. Overall, this mechanism can be used with the purpose of maintaining configurations of the systems, as it is flexible, cheap, and does not require specialized tools.

How do ongoing project development and maintenance benefit from existence of many forks?

This question aims at investigating if the project benefits in any way from the existence of forks. For example, forks may fix bugs and introduce new features. If these these changes are propagated to the main project, then the whole ecosystem benefits from this fork and its development.

I investigate this question through two focused questions to understand to what extent there is a flow of contributions between the forks and main Marlin.

To what extent do forks retrieve changes from their origins? To answer this question, I use a script to verify if commits in Marlin that were created after the fork's creation date, do exist in forks. Only 238 forks (15% of all forks) have synchronized at least once with the main Marlin repository (Fig. 5.3), which amounts to only 34% of active forks.

> *Most forks do not retrieve new updates from the main Marlin repository.*

Marlin developers fork significantly more often than merge. This is a striking observation, given that merging of concurrent development is the key purpose of distributed version control systems, and in particular of Git. The forks in the Marlin ecosystem are characterized by a short maintenance lifetime (101 days on average). Once a fork achieves the desired functionality (the printer operates as expected) the incentives to

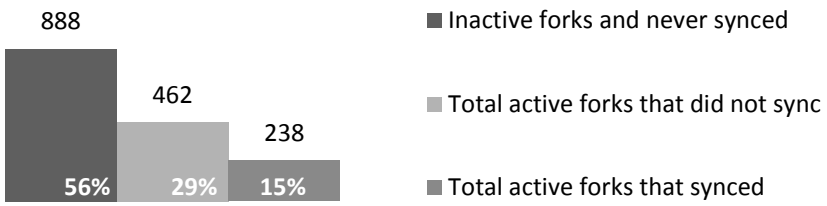


Figure 5.3: Quantitative data regarding synchronization of forks. The three columns represent the forks that 1) are inactive and never synchronized with the main Marlin repository, 2) are active (made changes) but did not synchronize, and 3) are active and synchronized with the main Marlin repository. These are non-overlapping groups, and percentages show the relative size in the set of all forks

maintain the fork decrease, and it becomes inactive (32% of active forks did not receive any commits between January 2014 and November 2014). Thus the period in which upstream changes are relevant for many developers is relatively short. The implication of not retrieving updates from the main project is twofold: first, users do not benefit from latest development and bug-fixes (I later show that this is an important aspect), and second, if changes are done in forks and might be pushed back into the main project at a later time, the integration process will be challenging (due to potential large merge conflicts) because of not synchronizing. Unfortunately, Github does not offer a simple mechanism to synchronize a fork with its main parent, or to selectively accept upstream changes. This can be done through Git, e.g., using cherry-picking, but the average user could benefit from a much simpler interface, without requiring to be a Git expert.

In comparison, in the survey responses, only 18% of active fork owners synchronize monthly, and 6% synchronize weekly. Others do not synchronize at all, or synchronize irregularly. When asked, they state that the upstream changes are uninteresting for them, or that they do not wish to take in new changes as integrating them costs additional work. Merging new changes from upstream can be difficult and time consuming. One inactive fork owner explained that *I fear that my settings/calibration could change, sometimes I stay 1–3 months without changing the firmware of my printer*.

At the same time, the *altruist* developers that want to contribute to the community synchronize more frequently. From 306 development forks, 142 have retrieved changes from the main Marlin repository. Moreover, 87% of pushed patches from development forks, come from those that synchronized. Being up-to-date with the main repository is key for producing clean up-to-date patches. Furthermore, it makes integration of changes and variants much easier due to smaller and less complex merge conflicts that may arise.

Forks		Number of pull requests			
fork level	#forks	total	open	merged	closed
1	197	389	56	245	88
2	5	7	3	3	1
3 and 4	0	0	0	0	0
unknown	51	92	2	51	39

Table 5.1: Pull request contributions from forks to the parent repository. The last row refers to deleted forks whose level is unknown

To what extent do forks contribute changes to their origins? In the set of active forks, only 202 forks (253 with forks that were deleted before our retrieval, and they are represented as unknown in pull requests on Github) contributed with patches to the main Marlin repository, so not even all feature development forks (306 in Fig. 5.2) have contributed pull requests upstream. Nevertheless, 714 commits have been integrated in main Marlin by merging pull requests, representing 58% of all commits in the main Marlin repository (excluding empty commits that acknowledge merges).

The ability to fork gives developers control over the code base, which encourages innovation. More than half of the commits in the main Marlin repository come from forks of Marlin. Although forking allows for isolated development, it is important that changes are frequently propagated in the ecosystem (in both directions). This brings visibility to new features and bug-fixes, and eases the overall maintenance of the system.

Another benefit of easy forking for developers is showing up in testing and debugging. Testing 3D printer firmware is difficult, because maintainers do not have access to all the supported hardware. Hence, changes that are related to new hardware are usually tested by users having the corresponding hardware (e.g. P335, P572). During the life of the project, many users debugged problems, reported bugs, and contributed fixes developed in their own forks (e.g. P335, P594).

Forking facilitates a gradual involvement of contributors. Fork owners gain experience working on their own forks. Once they gain reputation, they become committers in the main project. We have identified such cases both in our survey and in the interviews with the maintainers of the main project.

In summary, the development and maintenance of Marlin benefited from the multitude of forks in the following ways:

- Forks contribute new features and new hardware support.
- Fork owners test and improve the firmware on different hardware and configurations.

- Working on forks grooms new maintainers for the project.

Under what circumstances is forking preferred over integrated variability for creating and maintaining variants?

In order to understand why developers prefer forking over the preprocessor, I investigated scenarios in Marlin's history that are typical of fork-based development. This qualitative analysis included forks that are used to manage configurations and forks that develop features, but do not push changes upstream. I disregarded forks that push changes upstream as these have to wrap their changes using preprocessor annotations. Since Marlin itself was created as a fork, I investigated its origin and the initial rationale for creating Marlin. There are four main reasons as why forking is preferred over preprocessor annotations, presented in the following observation.

> *Marlin developers preferred forking over preprocessor annotations under the following reasons:*
>
> *S1. The maintenance time span for the developed code is expected to be short.*
>
> *S2. The external developer has no control over the upstream project.*
>
> *S3. A developer wants to create experimental code.*
>
> *S4. An active project provides a good skeleton for adding new functionality.*

Next, I detail the above reasons using concrete examples. Overall, there are 526 forks that did modifications to the configuration files. There are 316 forks (45% of active forks) that modified only the configuration files and made no other changes. For instance, 33d/marlin-i3 #abaec3b3 configures the firmware to comply with a specific hardware. Some other forks mix configuration changes with other development changes 0xPIT/Marlin #70c7dde7, which makes it difficult to create pull requests containing just the new code and no configuration noise. Both previous examples have not been updated afterward. Once the firmware is configured and running on the printer, new changes are not desired and no further maintenance is associated with these forks (S1).

Forks commonly develop features for their own use, which may be highly experimental. For instance, martinxyz/Marlin #a8d59b1a modifies the IRQ functionality of the software and even adds an alternative IRQ code (#2a1c0766) for the stepper motor control in the firmware (non-standard IRQs are unlikely to be used). In this case, experimenting with code and adding new features that the original project lacks are the main reasons for forking (S3).

The fork jcrocholl/Marlin is a first level fork of main Marlin. It adds support for a new type of printer, a so called *delta*, that works differently than the normal Cartesian printers. Delta printers use spherical geometry and compute the location of the

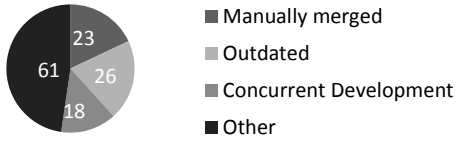


Figure 5.4: Reasons for not merging pull requests. The numbers represent how many pull requests were rejected in each category. Other includes: closed by the pull request author (no reason), bad patch, pull request created on wrong repository, or not fixing anything.

movements using trigonometric functions, such that the nozzle is not moved along the Cartesian planes. Before this extension, Marlin provided already support for some existing hardware and most of the needed software , which made it easy to introduce the extension. From user's perspective this was a large qualitative leap, almost a new project though, as it supported hardware with completely different design. The main Marlin project was not affected in any way by these changes. Moreover, the developer had complete control (S2) over his fork allowing him to progress fast.

Originally, Marlin itself was created by cloning and extending parts of klimen-t/Sprinter and grbl/grbl. The Sprinter firmware was itself a fork of tonokip/Tonokip-Firmware, which was based on Hydra-MMM firmware. So heavy forking in this community predates Marlin's time. These earlier projects provided a good skeleton, from which Marlin could evolve into a solid standalone variant (S4).

What other challenges arise in fork-based development projects?

I approach this question in the following way: (i) study the reasons for rejecting contributions, (ii) analyze if bug-fixes for important problems are propagated in the repositories and (iii) ask fork owners about importance and challenges of receiving and contributing bug-fixes from upstream. All of these aspects can potentially reveal information about frictions in project management on the boundary of forks. I organize the discussion along the identified challenges starting with redundant development and reasons for rejecting contributions, then moving to difficulties in propagating changes, and decentralization of information in forks.

Redundant development. The analysis of rejected pull requests showed that 18 pull requests (14% of all rejected pull requests) are rejected because of concurrent development (Fig. 5.4). The requested change either contained a feature or a bug-fix that already existed, or it was developed in parallel by two developers. This is a challenge not only for the contributors (e.g. P1087 or P223) but also for the maintainer who needs to have a good overview of all open pull requests to resolve the conflicts in the best

possible way (e.g. P594). Berger et al. [Berger et al., 2014a] confirm that concurrent development is a similar issue in industrial projects using clone-and-own.

Challenges in change propagation. The fact that forks do not retrieve changes from the main repository is problematic as fixes and new features are not propagated. To understand this phenomenon, I selected eight patches fixing important bugs (in an exploratory way using the gained knowledge of Marlin). The patches were created between January 2014 and November 2014. Thereafter, I checked if those patches were propagated to the forks. Figure 5.5 shows to what extent these patches have been adopted in forks. The light color part of the bar represents the number of active forks that have not pulled the patch, while the darker grey part represents active forks that have the patch (this only considers the active forks that existed before the patch was committed). For example, patch 1 in ErikZalm/Marlin #8a5eaa3c fixes a bug in a feature that may damage the printer. All the considered patch adoptions exhibit the same pattern.

Propagation of bug-fixes is a problem for forking, even though git offers facilities for selective download of patches from upstream (cherry picking).

At the same time, forks do not push changes back, so important fixes from the forks may never make it to the main repository. As many as 447 forks (63% of active forks) did not submit any patches to the main Marlin project. The survey data shows that one of the challenges is to prepare a robust pull request that does not break other features. A developer who works on his own fork, may find it difficult to take into account how his fix will affect configurations of all the other users. See for example a case of pull request P594 mentioned above, where a developer proposes to fix the problem for one hardware configuration, by removing the code that is necessary to make other configurations work. It is easier to maintain this general view on the variants, when integrated variability is used.

Decentralization of information. Decentralization of information is an issue that is specific to forking. Modifications and extensions to Marlin are not kept in one neatly organized repository but in several hundreds of forks. For example, malx122/Marlin #69052359 added support for a second serial communication, and also support for fast SD card transfer malx122/Marlin #326c59f6. These two features are not in the main Marlin repository, but they were actually taken from a fork (pipakin/Sprinter) of another firmware (kliment/Sprinter). Finding such features in the multitude of forks is extremely hard for the community members. This is consistent with an observation of Berger et al. [Berger et al., 2014a] that an excessive use of clone-and-own to create variants in industrial projects leads to loss of overview of the available functionality. The same work of [Berger et al., 2014a] mentions that centralized information is a key advantage of integrated variability management and feature modeling over fork-based

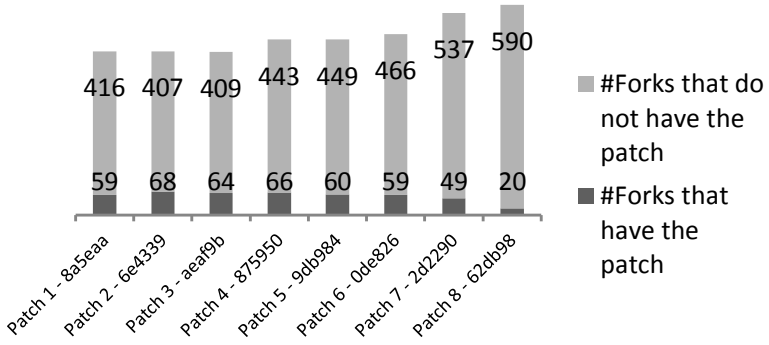


Figure 5.5: Synchronization of active forks for patches. The sum of the two represents existing active forks at the time creation of that patch.

management. Also diverged codebases make it hard for individual teams to know *who is doing what*, and what features exist elsewhere [Duc et al., 2014]. This is a problem in Marlin as well, even though, usually, individual forks do not depart far from the mainline. The sheer amount of forks makes it difficult to navigate and find interesting code.

Decentralization of information is a challenge in fork-based development.

What are the criteria that lead to integrating a forked variant into an integrated platform using preprocessor annotations?

Often forks and features developed in forks need to be integrated into the main project. However, this is a challenging, error-prone and time consuming process. I explore the question of when, why, and what are the criteria to integrate a variant into an integrated platform to understand how can integrations can be executed, and under what conditions. This step helps with designing the requirements for a tool that should support development and integration of variants. Analyzing how a few variants (e.g., alexborro/Marlin-BedAutoLev, jcrocholl/Marlin – deltabot) were integrated into the main Marlin, and how they evolved after the integration I find that:

Marlin fork owners consider integrating their forks into the main platform for the following reasons:

U1. Integrating widely used variants that need to be kept in sync with upstream reduces effort and evolution cost.

U2. Integrated features are more visible and attract more users.

Furthermore, through the interviews with the maintainers of Marlin I conclude that:

> *A fork-based variant is integrated into the main Marlin platform under the following conditions:*
>
> *U3. The quality of the feature is within standards. It has been tested and is known to work as expected.*
>
> *U4. Project maintainers accept to take over the maintenance, and the feature is aligned with project goals.*

The *Auto Bed Leveling* feature (#253dfc4b), was developed in a fork, later updated in another fork and finally integrated into the main repository. The integration took place after the feature has been tested and widely recognized[1] as well functioning (U3). On the other hand, if a feature is not working as expected and may introduce bugs or affect functionality, then it is not integrated. One such example is thinkyhead/Marlin #de725bd4, that adds support for SD card sorting functionality. This feature was developed by one of Marlin's maintainers, but was not accepted in the main project. The reason for not integrating this feature is that it causes problems in some specific cases (U4). Although it was not integrated at the time, the maintainer kept it in his fork because it is demanded by users.

The *deltabot* fork developed by jcrocholl/Marlin was integrated into the main Marlin repository (#c430906d, #6f4a6e53). Once integrated into the main project, visibility of new features and the variant was increased (U2) and it was easier to maintain it (U1). One of the maintainers stated that deltabot was merged *"because it was clear and we knew it had been well-tested"* (U3). Additionally, developers started to contribute changes to deltabot (P511,P568). During its existence as a standalone variant, in the deltabot repository there were only nine pull requests created and only one got accepted. Once it was integrated into Marlin, there were 20 pull requests related to it in ErikZalm/Marlin (55% more), out of which 14 were accepted (U2, U4). Interestingly, jcrocholl/Marlin remains a separate fork where the experimental development (S3) continues. This allows the owner to continue development outside the control of Marlin project maintainers (S2).

5.4 Summary

In this chapter I have shown through an empirical study the advantages and challenges of using preprocessor annotations and forking to develop software variants. It is clear that both mechanisms are flexible and preferred by developers under specific conditions. Using preprocessor annotations and forking is cheap (at least on short term),

[1]Forum discussion: "Bed Auto Leveling.. check this out": `http://forums.reprap.org/read.php?151,246132`

and does not require specialized tools. This confirms the first hypothesis (*H1*) and answers *RQ1*. This chapter provides insights on how to combine the two techniques to get the best of both worlds.

Forking is a good reuse mechanism for quick development and experimentation, but it is problematic with regards to maintenance and propagating changes if not appropriate measures are taken. Furthermore, forking leads to decentralized information, which is a challenge observed also in industrial systems [Duc et al., 2014, Berger et al., 2014a, Dubinsky et al., 2013]. Unfortunately, decentralization of information is more and more common [Gousios et al., 2014].

The ability to fork gives developers control over the code base, which encourages innovation. Although forking allows for isolated development, it is important that changes are frequently propagated in the ecosystem (in both directions). This brings visibility to new features and bug-fixes, and eases the overall maintenance of the system. As we have seen in the deltabot variant, synchronizing often with the main repository translates into small maintenance effort chunks instead of one large merge. This allowed the developer to quickly experiment with the deltabot variant, while not diverging too far from the main repository. The end result was a clean and effortless integration of this variant into the main project, which led to more pull requests and changes to the new variant. To this end, I believe that forking is a viable mechanism for reuse in the context of developing software variants, but it has to be carefully used and managed. In particular, synchronizing often and merging features or variants increases visibility and reduces maintenance costs.

Preprocessor annotations are also a cheap reuse mechanism and are heavily used in embedded systems. Unfortunately, an uncontrolled usage of preprocessor annotations inhibits program comprehension due to code cluttering and makes maintenance more difficult. Although their negative effects are known, these are broadly used in practice. The main reasons for using preprocessor annotations is their flexibility, fine granularity and that developers know how to use them. Using preprocessor annotations software systems with limited resources can be configured according to a specific configuration, thus giving flexibility for deriving many variants from one common codebase.

An important lesson is to be conscious of when and how often to use preprocessor annotations. When code becomes cluttered with annotations and maintenance is hindered by them, it is important to re-architect and refactor the code to improve comprehension and ease the maintenance. While this is usually a manual and error prone practice, in recent years more tools are supporting automated refactoring of code with preprocessor annotations. Transforming compile time variability into runtime variability [Liebig, 2015, Iosif-Lazar et al., 2017] is useful to use off the shelf static analysis tools. Detecting variability code smells [Fenske et al., 2015, Fenske and Schulze, 2015] can help in identifying potential problems due to usage of preprocessor annotations. Nevertheless, preprocessor annotations are a cheap, easy and fast reuse mechanism used in thousands of systems in open source and industry alike.

6. Variation Control Systems Paper C

This chapter shares content with paper "Concepts, Operations and Feasibility of a Variation Control System" published in ICSME 2016 [Stănciulescu et al., 2016a]. For complete details, refer to the paper in Appendix A.3.

In the previous chapter, I discussed the advantages and disadvantages of developing variants using preprocessor annotations and forking in the highly configurable Marlin system. In this chapter, I present a variation control system that is designed to handle the development of software variants. The advantage of using a variation control system is that it can alleviate some of the disadvantages of using preprocessor annotations. Through an empirical study, I identify what kind of edit operations are common when developing and maintaining highly configurable systems. Based on these identified edit operations, I analyze how and if these are supported by the proposed variation control system. Finally, this chapter presents how forking can be integrated in the variation control system to support quick and low cost initial reuse.

6.1 Introduction

Various methods and tools that allow working on dedicated subsets of all variants have been proposed in the literature [Walkingshaw and Ostermann, 2014, Le et al., 2011, Atkins et al., 2002, Kruskal, 1984, Kästner et al., 2008, Schwägerl et al., 2015]. To some degree, these systems can ease the engineering of highly configurable systems by providing views that only show the code related to specific variants or feature code, while hiding irrelevant code. However, none of them has found widespread adoption. In fact, little empirical data is available that shows *how* exactly they can be used to engineer real-world systems, and what their specific benefits and challenges are. Furthermore, it is needed to understand what edit operations users need and how a specialized variant development tool could support these operations.

In this chapter, I present a variation control system designed to cope with some of the challenges of preprocessor annotations. In an empirical study I extract a set of edit patterns that are related to variability changes.

```
1  // LCD selection
2  #ifdef U8GLIB_ST7920
3    //U8GLIB_ST7920_128X64_RRD u8g(0,0,0);
4    U8GLIB_ST7920_128X64_RRD u8g(0);
5  #elif defined(MAKRPANEL)
6    // The MaKrPanel display, ST7565 controller as well
7    U8GLIB_NHD_C12864 u8g(DOGLCD_CS, DOGLCD_A0);
8  #elif defined(VIKI2) || defined(miniVIKI)
9    // Mini Viki and Viki 2.0 LCD, ST7565 controller as well
10   U8GLIB_NHD_C12864 u8g(DOGLCD_CS, DOGLCD_A0);
11 #elif defined(U8GLIB_LM6059_AF)
12   // Based on the Adafruit ST7565 (http://www.adafruit.com/products/250)
13   U8GLIB_LM6059 u8g(DOGLCD_CS, DOGLCD_A0);
14 #else
15   // for regular DOGM128 display with HW-SPI
16   U8GLIB_DOGM128 u8g(DOGLCD_CS, DOGLCD_A0);  // HW-SPI Com: CS, A0
17 #endif
```

Listing 6.1: Marlin excerpt (dogm_lcd_implementation.h at commit a83bf18)

Then I assess the feasibility of using a variation control system in supporting the identified edit patterns. Specifically, the prototype is used to replay parts of the history of the highly configurable software, Marlin. To give an intuition of how a variation control system works, I use the the example shown in Listing 6.1.

A variation control system allows us to checkout a subset of all the variants from our code repository. That is, the source code that is generated by the variation control system is a subset of the initial code, according to a given projection. Let us use the projection *U8GLIB_ST7920*. In its simplest form, a projection is a propositional formula. The projection specifies a concrete variant to be visualized, yielding the following (temporary) code.

```
1  // LCD selection
2    //U8GLIB_ST7920_128X64_RRD u8g(0,0,0);
3    U8GLIB_ST7920_128X64_RRD u8g(0);
```

Listing 6.2: The view created by the variation control system that is specific for the *U8GLIB_ST7920* variant

The generated view, that is a *projection* of the code, can be edited with any tool a developer desires, e.g., a text editor or an IDE. In the view we can observe that the code has no preprocessor annotations and it is simpler to read and understand. Let us now delete line 2, and modify line 3 to call a different constructor, as following.

```
1  // LCD selection
2    U8GLIB_256X128_RRD u8g(0);
```

Listing 6.3: The edited view after we performed the desired changes

Once the code changes have been done, we can update the original code with the changes. This is specified through the checkin operation and uses an ambition.

```
1  // LCD selection
2  #ifdef U8GLIB_ST7920
3    U8GLIB_256X128_RRD u8g(0);
4  #elif defined(MAKRPANEL)
5    // The MaKrPanel display, ST7565 controller as well
6    U8GLIB_NHD_C12864 u8g(DOGLCD_CS, DOGLCD_A0);
7  #elif defined(VIKI2) || defined(miniVIKI)
8    // Mini Viki and Viki 2.0 LCD, ST7565 controller as well
9    U8GLIB_NHD_C12864 u8g(DOGLCD_CS, DOGLCD_A0);
10 #elif defined(U8GLIB_LM6059_AF)
11   // Based on the Adafruit ST7565 (http://www.adafruit.com/products/250)
12   U8GLIB_LM6059 u8g(DOGLCD_CS, DOGLCD_A0);
13 #else
14   // for regular DOGM128 display with HW-SPI
15   U8GLIB_DOGM128 u8g(DOGLCD_CS, DOGLCD_A0);  // HW-SPI Com: CS, A0
16 #endif
```

Listing 6.4: Updated variant code (shown in blue) after the checkin operation

The ambition is dual to the projection and specifies which variants should be affected by this change. For this example, we assume that we only wanted to modify the *U8GLIB_ST7920* variant, so that becomes our ambition. The variation control system now updates the platform and yields the code in Listing 6.4. The updated code does not contain line 3, and the previous line 4 now becomes the modified line 3 (highlighted with blue).

Through this sequence of operations, a variation control system supports the following workflow:

> *Checkout* the source code according to a given projection that yields a version of the code with less variability,
>
> *Edit* the source code shown in the view generated by the given projection,
>
> *Checkin* the edited code, which updates the underlying variational code.

The variation control system presented in this dissertation combines and extends concepts from prior work [Walkingshaw and Ostermann, 2014, Lie et al., 1989]. To describe it, I use choice calculus [Erwig and Walkingshaw, 2011b, Walkingshaw, 2013, Walkingshaw and Erwig, 2012], a concise and formal notation that avoids dealing with intricate preprocessor annotations syntax and semantics, and allows reasoning about highly configurable systems.

6.2 Background

In the beginning of the 1980s there has been some advancement on improving the comprehension of source code that uses preprocessor annotations to implement variability. One of the first tools was the P-Edit, developed by V. Kruskal, in 1984 [Kruskal,

1984]. A similar idea was introduced in the Change Oriented Versioning by [Lie et al., 1989] which led to an actual tool developed in Munch's Ph.D. [Munch, 1993]. Recently, Walkingshaw [Walkingshaw and Ostermann, 2014] developed the projectional editing model, that is as well similar to the previous two ones. A few other similar ideas exist [Blendinger, 2010, Kästner and Apel, 2009] as well, though none got any traction in practice.

In general, all these ideas and tools rely on variability realized using annotations embedded in code, similar to preprocessor annotations. These annotations carry a Boolean expression over *features*, called *presence condition* in the remainder of this chapter. The P-Edit system and the CoV system are not available anymore and there is hardly any evidence on their usefulness. The third one has not been empirically evaluated and no publicly available tool exists. The first two approaches unify versioning and variability. P-Edit is realized on top of an editor, while CoV is similar to a version-control system with control over which variants are checked out and in. In the rest of the section, I will describe in more detail the previous variation control systems.

P-Edit - 1984 Kruskal [Kruskal, 1984,Kruskal, 2000] presents an editor that realizes both concurrent versioning (variability) and sequential versioning (evolution in time) relying on variability annotations. Similar to conditional compilation, code lines are mapped to Boolean presence conditions, representing both the variant and the version to which the lines belong to. The editor creates views based on a partial configuration (a conjunction of features) called *mask*, supporting workflows where developers start with a relatively broad mask (e.g., projecting on just one feature), potentially restricting the mask by conjoining other features (e.g., "push" more masks on a stack), editing code in the views belonging to the more restricted masks, and then returning to more broad masks (e.g., "pop" masks from a stack). Code lines with presence conditions that do not contradict the mask are visible for editing. Several convenience commands are available to the user for iterating through variants and for manipulating presence conditions. The editor is not available anymore, and no empirical data on its use exists.

Change Oriented Versioning - CoV 1993 Lie, Munch and Westfechtel [Lie et al., 1989, Munch, 1993, Westfechtel et al., 2001] present and evaluate an alternative versioning model based on logical changes: change oriented versioning (CoV). It also unifies concurrent and sequential versioning, by attaching Boolean presence conditions to file fragments. It follows a classical checkout/checkin cycle, where a configuration (conjunction of features) determines both the version and the variant available in the view (e.g., the workspace), which can be edited. It decouples the projection (called "choice") from the expression used to checkin the edited view (called "ambition") to denote to which variants a change applies to. In an empirical study the authors translate existing C/C++ source code files of *gcc* into CoV representation in a database (EPOSDB-II [Gulla et al., 1991]), and compare the performance against RCS [Tichy,

1985] when doing a full checkin of version 2.4.0 of *gcc*. The experiment does not show how feasible it is to actually engineer a real-world system, and how exactly the checkout/checkin cycles using choices and ambitions can be used by developers.

Projectional Editing - 2014 Walkingshaw et al. [Walkingshaw and Ostermann, 2014] present a model for a variation control system called projectional editing.[1] They present a formal specification of the model with the get (create a view using a projection) and put (update the underlying program with changes done within the view) functions at its core. Examples are provided that show how to create the view and how an update executes the changes done to the view. However, in contrast to CoV, the definition of put is founded on an *edit isolation principle* that ensures that the only variants that change in the underlying program are those that can be reached from the view. In other words, when we use put to perform the update, the edits made on the view are guaranteed not to affect code that was hidden by the get function. Given this limitation, and since it was not evaluated on a real-world system, it is not clear whether this model can handle the engineering of real-world, highly configurable systems.

6.3 Variation Control System Design

The variation control system prototype presented next is a generalization of the one described before [Walkingshaw and Ostermann, 2014]. In addition to the original projectional editing, it uses the concept of *ambition* from CoV [Munch, 1993], which specifies what variants are affected by the change when updating the code.

Workflow

Figure 6.1 shows the intended workflow of the system. Symbols r and r' refer to the highly configurable software source code stored in a repository, and v and v' refer to working copies of the source code that are viewable and editable by the developer. The source code (r, r', v, v') is represented by a choice calculus expression (e in Fig. 2.8). The operation get is used to checkout a particular working copy from the repository, and put is used to checkin any changes back to the repository. The workflow is independent of the type of storage used to contain the source code (e.g., version control systems or just folders).

In step (1) the developer obtains a simplified view v from the initial repository r. The parameter p, the *projection*, defines how v is obtained from r. More specifically, p describes a partial or complete configuration of r, eliminating all of the variability that is irrelevant to the current editing task. In step (2) the developer edits v into v' using

[1] Not to be confused with projectional editing [Voelter et al., 2014, Berger et al., 2016] used in the Meta Programming System [mps,] or the Intentional Domain Workbench [Christerson and Kolk, 2009]

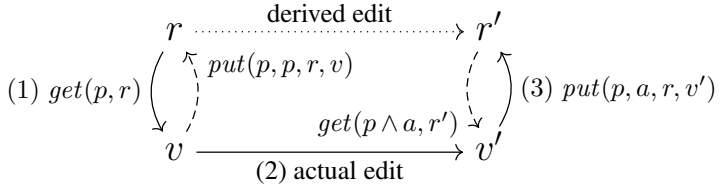


Figure 6.1: Projection-based variational editing workflow and relationships. Symbol r represents the repository that contains source code, p is the projection that specifies how to obtain the view v from r using the get function. The ambition a specifies how should the changes from the edited view v' be applied to the repository using the put function. Both p and a are Boolean expressions over features.

whatever standard tools they prefer. In step (3) an additional parameter a, the *ambition*, is introduced, which specifies how the developer's changes should be integrated into the repository. Note that the put operation takes into account the initial repository, the updated working copy, the projection, and the ambition when producing the updated repository r'.

The two dashed edges in the diagram describe some basic consistency principles that get and put should satisfy. These are derived from the *lens laws* developed in research on bidirectional transformations [Foster et al., 2007] and constrain the potential definitions of get and put.

The left dashed line requires that a get followed by a put is idempotent. Specifically, if a simplified version is retrieved v with some projection p and then immediately it is checked back in with the same ambition $a = p$, then the repository should remain unchanged. This enforces that the get operation is not effectful from the perspective of the repository.

The right dashed edge requires that a put followed by a get (with an appropriately structured projection) is idempotent. Specifically, immediately after applying the put function to checkin changes from the edited view v', that same working copy can be obtained by doing a checkout using the conjunction of p and a as the projection. This enforces that the put operation is always reversible.

Get and Put Function

Specifying and implementing get is straightforward (e.g., using partial or full preprocessing). We chose partial preprocessing to allow projections that are partial configurations. In practice, it might be unlikely that a developer has to work on a full configured product at once. The function get obtains a view using the following process. It iterates through all top-level choices in the AST. It takes the right alternative if the choice's presence condition contradicts the projection. It takes the left alternative if the negated presence condition contradicts the projection. Contradictions are checked using a SAT solver. If neither the presence condition nor its negation contradict the projection, get

keeps the choice as it is. For each not eliminated alternative, get repeats this process descending into each alternative's sub-tree.

In the previous projectional editing work [Walkingshaw and Ostermann, 2014], these functions relied on an *edit isolation principle*: when doing a put in (3), the edits made in (2) cannot affect code hidden by get in (1). Although this principle is somewhat restrictive, it leads naturally to a definition that satisfies the requirements derived from the lens laws. Generalizing the edit isolation principle, we obtain a put operation that is less restrictive than in the previous work, while still retaining the properties.

This generalized edit isolation principle can be defined as follows. Let $\mathcal{C}$ be the set of all configurations of r and r', and $r' = put(p, a, r, v')$ as defined in Fig. 6.1. The get function obtains all choices whose presence condition does not contradict the projection. If the presence condition contradicts the projection p, then get returns the projection from the original source.

$$\forall c \in \mathcal{C}.\, get(c, r') = \begin{cases} get(c, v') & \text{if } \mathit{SAT}(c \wedge p) \\ get(c, r) & \text{otherwise} \end{cases}$$

The put update function consists of constructing a new choice with the updated view v' in the left branch, and the original source r in the right branch:

$$put(p, a, r, v') = minimize(F\langle v', r\rangle)$$
$$F = (p \wedge a)$$

The next step is done by the *minimizing* function, which minimizes the choice expressions to a more compact representation. The goal is to reduce redundancy in the newly created choice. The minimize function uses several reduction rules (see Fig.3 from paper A.3 for full details). Note that they can change the syntax of a choice, but preserve its semantics.

Implementation

The variation control system is implemented in Scala, comprising of a parser, get and put function, minimization rules, and pretty printer. The prototype is programming-language independent (line-based), but the parser and pretty printer recognize and write variability annotations in C preprocessor syntax (e.g., `#if`, `#ifdef`, `#endif`). The tool is command-line based (see Fig. 6.2), similar to Git. It has a few simple commands such as

- checkout - used to get a view on the code through a given projection. One or multiple files can be checked out.
- checkin - used to put the changes back into the integrated platform with the specified ambition. One or multiple files that were checked out can be checked in.

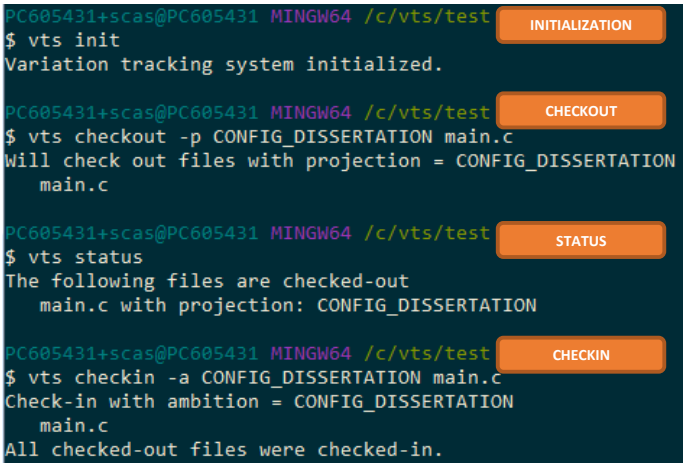


Figure 6.2: The Variation Control System commands from the command line interface.

- status - used to show which files are projected out and the projection used.

6.4 Study Design

Now that the variation control system has been described in detail, the question is how to test and evaluate the system. A first challenge that arises is that there is no understanding of what kind of changes can be performed with this variation control system and its dedicated workflow. Therefore, in this section I describe the steps taken and the design of the study to answer *RQ2 — How can a specialized variation control system leverage the advantages of preprocessor annotations and forking to support the development, maintenance and integration of software variants, while minimizing some of their drawbacks?*

I split the research question into two subquestions:

RQ2a *What edit operations should a variation control system support?* To answer this question, I analyze the commits of Marlin to identify variability-related editing patterns. These edit patterns are used to understand the kind of changes that are done to a configurable system, as well as verifying which of them can be executed with the variation control system.

RQ2b *Can a variation control system be used to maintain and evolve a highly configurable system?* In an experiment, parts of the history of Marlin are replayed using the variation control system prototype. The motivation for replaying changes is to understand if indeed a variation control system can support the development of software variants. The specific scope of the experiment is to determine how well editing cases are supported and which are not trivial to support.

RQ2a: Identification of Edit Operations

First step is to identify edit-related changes and extract edit patterns. To achieve this, 3747 commits (without merge commits) from Marlin's history are used. Each commit is split into a patch per changed file, excluding those files that were added, removed or renamed, resulting in 5640 patches. The *removed or renamed* files are ignored because they have already OS supported operations, and a variation control system does not require first class support for them. Adding new (large) files is too complicated to support with a variation control system if the file has specific code to many variants. Usually, a new file contains many code blocks that are added manually and then the whole file is added to the repository.

Patches are classified into patterns in three steps. First, manual inspection is done on 50 random commits that add or remove **`#ifdef`** directives in code using *grep*, to understand the change and recognize patterns. Second, several regular expressions that represent the patterns are created to automatically classify to which edit pattern a patch belongs to. Next, a check if all patches have been classified is performed. For patches that remain unclassified, the respective regular expressions are added, and the classifier is executed again. These steps are repeated until each patch is classified by at least one pattern (note that a patch can belong to multiple patterns).

To cross-validate the patterns the classifier is applied on Busybox project, a larger project with 175 KLOC. The project's git repository[2] was downloaded at commit a83e3ae, and contains 13,700 commits excluding merge commits. These commits are also split into a patch per file, yielding 34,018 patches. Running the classifier on this set yields a recall of 97%, meaning that 97% of the patches fall into one of the categories determined in the Marlin set. The high recall gives confidence that the classifier is working as expected.

RQ2b: Replaying a Sample of Marlin's History

To replay changes already executed in Marlin to understand if the edit patterns can be well supported, 2322 patches were considered for sampling. These patches only modify files containing only Boolean presence conditions. This is justified as analyzing the complexity of Marlin's presence conditions is not the main goal here. Other kinds of presence conditions could be handled using an SMT or CSP solver, without affecting the variation control system's main design features. From the 2322 patches, three patches are selected for each identified edit pattern. Some patterns did not have any purely Boolean representative in the selection. For these patches from the whole pool of 5640 (Boolean and non-Boolean) patches were selected randomly, and transformed their non-Boolean expressions into Boolean ones by introducing new variables for non-Boolean sub-expressions (a simple form of predicate abstraction).

[2] `https://git.busybox.net/busybox`

For each randomly selected patch, the projection and ambition are manually conceived by looking at the code and the patch, and respectively at the end result of the change. Each change is executed according to the workflow: S1 and S3 are done by our prototype, and S2 is done manually in a text editor.

S1 Checkout the original source code file using the projection,

S2 Edit the view to apply changes from the patch,

S3 Checkin changes using the ambition.

Metrics Two metrics are computed during the execution of the replay. The metrics are computed for: original source code (r), view on source code (v), and updated source code by our system (r'). To compare the latter to the updated original source code, metrics for the original update (r') from Marlin's Git repository are computed. The two metrics are:

- LOC: lines of code in a file, including comments but excluding blank lines. This metric is used to verify if the projection could improve the program comprehension by eliminating unnecessary code that is not needed when executing a given editing task.
- NVAR: number of variation points; more precisely: choices (represented by `#if`, `#ifdef`, `#ifndef`, etc.) in a file. A high NVAR challenges code comprehension. I hypothesize that using a projection, the number of code blocks wrapped by presence conditions is reduced in the view.

To get a better representation of how useful is the projection, a ***reduction factor*** is computed for LOC and NVAR. Its value is computed by dividing the value in the view to the value before checkout. The reduction factor is aimed to visualize the effect and positive or negative impact of views compared to the original source code.

6.5 Edit Operations

To identify the edit operations, the classifier ran on the set of 2322 patches several times until all patches were classified. This execution led to the identification of 14 edit patterns, shown in Tbl. 6.5.

RQ1a: There are 12 patterns representing edit operations that a projection-based variation control system needs to support.

I split the 14 edit patterns into three categories: code-adding patterns, code-removing patterns and other kind of patterns. I show how they can be turned into specific variation control system edits, that are executed with its workflow. For each category, I

Name	#Multi	#Only	Example
P1 AddIfdef	969	129	$\iota \to F\langle e, \iota\rangle$
P2 AddIfdef*	424	32	$(\iota \to F\langle e, \iota\rangle)^*$
P3 AddIfdefElse	271	4	$\iota \to F\langle e_1, e_2\rangle$
P4 AddIfdefWrapElse	43	17	$e_2 \to F\langle e_1, e_2\rangle$
P5 AddIfdefWrapThen	13	3	$e_1 \to F\langle e_1, e_2\rangle$
P6 AddNormalCode	4683	871	$\iota \to e$
P7 AddAnnotation	293	12	*not applicable*
P8 RemNormalCode	3932	209	$e \to \iota$
P9 RemIfdef	534	24	$F\langle e_1, e_2\rangle \to \iota$
P10 RemAnnotation	228	2	*not applicable*
P11 WrapCode	77	29	$e \to F\langle e, \iota\rangle$
P12 UnwrapCode	12	2	$F\langle e, \iota\rangle \to e$
P13 ChangePC	225	74	$F_1\langle e_1, e_2\rangle \to F_2\langle e_1, e_2\rangle$
P14 MoveElse	5	2	$F\langle e_1, e_2 \cdot e_3\rangle \to F\langle e_1 \cdot e_2, e_3\rangle$

Table 6.1: The 14 edit patterns identified by the classifier. The #Multi column indicates the number of patches that match the given pattern and also one or more other patterns. The #Only column indicates the number of patches that match only that pattern. The last column provides a brief illustration of the pattern using the choice calculus.

show one pattern in detail. The rest of the patterns are detailed in Appendix A.3, Sec. V.

To represent changes, I use a stripped notation of the *unified diff* program. A plus (+) in front of a line indicates that the line is to be added, a minus (−) that the line is to be removed. A line without plus or minus remains unmodified.

Code-Adding Patterns

P4 AddIfdefWrapElse. This pattern represents cases where some existing code becomes the **`#else`** branch of a new **`#ifdef`** block. The pattern is presented below.

```
1 + #ifdef ULTRA_LCD
2 +  lcd_setalertstatuspgm(lcd_msg);
3 + #else
4    alertstatuspgm(msg);
5 + #endif
```

Listing 6.5: Add #ifdef and wrap existing code in else branch

This pattern can be executed with the variation control system workflow, as illustrated in Fig. 6.3. In this figure, and the rest of the section, I simplify the code and

replace the actual code lines with the first word or characters of that line. For example, line 2 in this listing is represented as *lcd*, while line 4 is represented as *alert*. This makes it simpler to draw the examples through the workflow, and keeps it concise. The presence condition is represented using its initial character, e.g., ULTRA_LCD to U.

The first step is to perform a checkout with a trivial projection (e.g., true). The *true* is used as a projection because the existing previous code exists in all configurations (mandatory code). Once we have the view, we can replace the old code with new code that we want (e.g., line 2 from the patch). Finally, the changes are applied via a checkin with the ambition *ULTRA_LCD*, resulting in the same code as the change in the pattern.

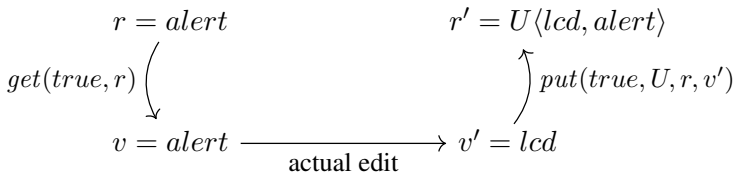


Figure 6.3: P4 AddIfdefWrapElse editing workflow.

Note that a checkout with projection U would yield the same result, but the advantage of this workflow is that we can decide after making the edits how they are be applied to the repository.

Code-Removing Patterns

The code-removing patterns are represented by P8, P9 and P10 patterns.

P9 RemIfdef. This pattern captures cases where code blocks guarded by presence conditions are removed. This pattern covers the removal of both simple **`#ifdef`** blocks and those containing an **`#else`** branch. The pattern is presented below:

```
1 - #ifdef ULTRA_LCD
2 -  lcd_setalertstatuspgm(lcd_msg);
3 - #else
4 -  alertstatuspgm(msg);
5 - #endif
```

Listing 6.6: Remove Ifdef code

This edit can be supported either by doing a trivial projection, where the code is simply deleted, or in a idiomatic way through a sequence of two edits as illustrated in Fig. 6.4.

Executing the change in two sequences is the idiomatic way of using the workflow, even though one extra step is required compared to a traditional text editing case.

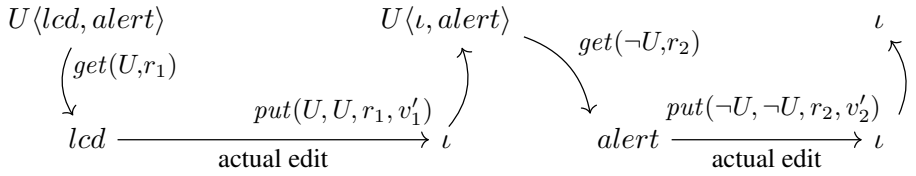


Figure 6.4: P9 RemIfdef editing workflow.

Other Patterns

The remaining edit patterns are P11 to P14.

P11 WrapCode. This pattern describes cases where an existing piece of code is made optional, as shown below:

```
1 + #ifdef ULTRA_LCD
2     lcd_setalertstatuspgm(lcd_msg);
3 + #endif
```

Listing 6.7: Wrapping existing code

This pattern can be supported by the following the workflow in Fig. 6.5. First, checkout with trivial projection. Then delete the code that should be conditionally wrapped, in this case line two. Finally, checkin with an ambition that describes the configurations in which the code should no longer appear (e.g., ¬*ULTRA_LCD*).

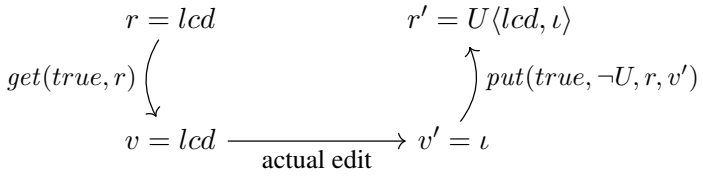


Figure 6.5: P11 WrapCode editing workflow.

As I mentioned before, the rest of the patterns are detailed in A.3, Sec. V.

RQ2a: From the total of 14 edit patterns identified, the variation control system should support 12 of them. The excluded two patterns (P7 and P10) are the ones in which annotations are added or removed as a single operation. The annotations are handled automatically by the variation control system through the usage of ambition, thus these cannot be directly supported.

6.6 Evaluation

The objective in this experiment is to verify if the edit patterns described in Sec. 6.5 are indeed supported, and what kind of projections and ambitions are used. Moreover, this also explores if there are any negative effects on the source code when using the prototype variation control system.

Applying the Changes

Following the methodology from Sec. 6.4, 33 patches were selected randomly to be applied with the variation control system. The criteria to this selection was that each patch should belong to only one edit pattern. The patches cover 12 edit patterns out of the 14 in total. Two edit patterns, (P7 AddAnnotation, P10 RemAnnotation), cannot be executed with the prototype because the tool manages annotations automatically.

All the selected patches were successfully applied using the variation control system. The actual changes on the view were performed with a simple text editor. Note that for all patches that add or remove `#ifdef` blocks, only the code between the annotations was touched, to realize the edit; the annotations are handled by the variation control system.

> *A projection-based variation control system can support all the presented edit patterns when no malformed variability annotations exist, to maintain and evolve a highly configurable system.*

This observation answers *RQ2b* with regards of maintaining and evolving a highly configurable system.

Complexity of Projections and Ambitions

Since some patches required multiple steps to execute the change, we performed a total of 37 *projections*. Of these, 14 use one feature and 11 the trivial condition *true*. The remaining 12 projections use two, three or four features in their expressions. In three cases the projection is the conjunction of four features, making these projections more difficult to understand and use.

Yet, it is not uncommon that a developer needs to consider two or more features (i.e., ≥ 4 system variants) when fixing bugs. In fact, Abal et al. [Abal et al., 2014] identify 30 bugs that occur when there is a combination of at least two configuration options. In such cases, using a projection-based editing tool could simplify the task, focusing only on the variants in which the bug appears.

In *ambitions*, the highest number of features is the same as in projections, four. But we see a decrease of trivial ambitions, which is expected, as for example P11 WrapCode edits may be performed on trivial projections, but require an ambition different

	LOC		NVAR	
	Our *put* function	Repository update	Our *put* function	Repository update
MIN	65	72	1	1
MAX	2448	2368	193	147
MEDIAN	447	449	20	21

Table 6.2: loc and nvar metrics with the min, max, and median values for the 33 changes for our put function. Repository update represents the change done by the developer in the original git repository of the project.

than $true$. In one case the expression used for both projection and ambition is a conjunction of a feature and a disjunction, $p = A \wedge (B \vee C)$. Finally, for 18 changes the ambition equals the projection.

Metrics and Reduction Factors

Table 6.6 shows the aggregate values (min, max, and median) of our metrics on the source code resulted from the update done by the prototype, and the original update from the Git repository. While our goal is not to improve code with regards to LOC or NVAR, Table 6.6 shows that the prototype does not perform worse than the original update in almost all cases.

The boxplot in Fig. 6.6 shows the reduction factor for LOC and NVAR after the projection. For the LOC reduction factor when doing projections, we would expect it to be zero, in the case of using a trivial projection, or larger than zero when a non-trivial projection is used. Table 6.6 and the boxplot confirm this hypothesis. The average number of LOC after projection is smaller than before projection. In one outlier case the LOC in the view was reduced by half, compared to the original file. The reason is that the source code has a sequence of `#if-#elif-#else-#endif` directives with many `#elif` branches, which naturally contradicted the projection. In such cases, the benefit of projecting views can be high, especially for code comprehension (e.g., to understand the control flow). Although a small improvement exists on average, in specific cases, a variation control system can help with program comprehension. Furthermore, these changes have been performed through a partial configuration. By

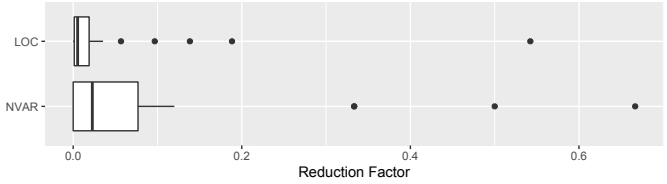


Figure 6.6: Reduction factors for LOC and NVAR for the view

design, if the presence condition is not contradicting the projection, but it is also not implied by it, then the prototype will keep that choice as it exists. I discuss more of this result in the discussion section of this chapter.

As expected, NVAR is reduced when projecting the code, although this reduction is minimal in most cases. In this experiment, many changes are done on features that wrap an entire file's source code or use the trivial projection $true$. Nevertheless, in three outlier cases there is a high decrease in NVAR when many `#ifdef` blocks are projected away.

RQ2b: A projection-based variation control system can be used to engineer a highly configurable system. The prototype did not negatively impact the code in terms of LOC, NVAR. In some cases it even improved the code.

6.7 Variation Control System with Forking Support

Forking is a mechanism used for quick development and experimentation, without affecting the stability of the code. Previously, in Ch. 5 I have shown that the main drawbacks of forking are 1) expensive maintenance and difficult to propagate changes, and 2) it leads to decentralized information. The key idea is to *unify* the preprocessor annotations and forking, and use a common codebase for both. Combining the two into one codebase mitigates the limited change propagation and decentralized information problems.

The main difference between an integrated platform and developing a variant via forking is that in the former, all the source code assets are usually stored in one repository. However, once a fork is created, a new branch or repository is created. These artifacts are now duplicated and spread across multiple repositories. Storing all the assets in one repository (monolithic codebase) makes it easier to maintain, improves the visibility of new features and bug-fixes, and centralizes the information in one place.

To add forking to the variation control system, the developer can specify during a check-in operation if the changes belong to a fork or not. By default, the system will wrap these changes with a *CONFIG_FORK_NAME* annotation. The code changes will be incorporated into the same codebase as the rest of the code. There are two

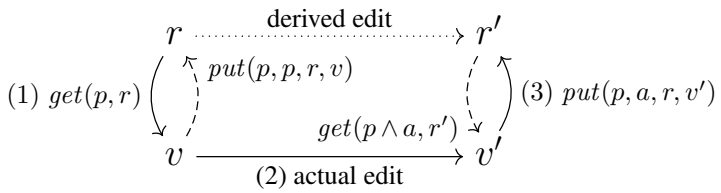


Figure 6.7: Variation control system editing workflow.

```
1 // LCD selection
2 #ifdef U8GLIB_ST7920
3   //U8GLIB_ST7920_128X64_RRD u8g(0,0,0);
4   U8GLIB_ST7920_128X64_RRD u8g(0);
5 #endif
6
7 static char currentfont;
```

Listing 6.8: Marlin excerpt (dogm_lcd_implementation.h at commit a83bf18)

advantages of keeping it in one codebase: 1) the tool can show what changes are done in the forks (centralized information), and 2) any changes to the mandatory code (visible to all configurations) are also present when working on the forks. Thus, the propagation problem is partially solved.

Demonstration

Recall the workflow of the variation control system (see Fig. 6.7). I will reuse the example with the LCD in a shorter form. To the existing code I add a line in the end, that is common to all variants as illustrated in Listing 6.8. Let's assume that we want to develop on the feature *U8GLIB_ST7920* and we do a checkout with that projection. We get the following code in a view.

```
1 // LCD selection
2   //U8GLIB_ST7920_128X64_RRD u8g(0,0,0);
3   U8GLIB_ST7920_128X64_RRD u8g(0);
4
5 static char currentfont;
```

Listing 6.9: The view created by the variation control system that is specific for the *U8GLIB_ST7920* variant

Now, we make some modifications as in Listing 6.10, and we want to create a new fork. We use the ambition *U8GLIB_ST7920* and specify the fork's name *HW_LCD_TEST* via the *-c* option for the checkin command.

```
1 // LCD selection
2   DISPLAY_128X64_RRD init(0);
3
4 static char currentfont;
```

Listing 6.10: Changes done to prepare for a new variant via forking

The end result is shown in Listing 6.11. A first difference when using forking with the tool is that by default it excludes the forks from any projections. Therefore, a simple projection on *U8GLIB_ST7920* will yield the same code as before (Listing 6.9, where as if we include the fork we will get the new code corresponding to the fork.

```
1  // LCD selection
2  #if U8GLIB_ST7920 && CONFIG_FORK_HW_LCD_TEST
3    DISPLAY_128X64_RRD init(0);
4  #else
5  #ifdef U8GLIB_ST7920
6    //U8GLIB_ST7920_128X64_RRD u8g(0,0,0);
7    U8GLIB_ST7920_128X64_RRD u8g(0);
8  #endif
9
10 static char currentfont;
```

Listing 6.11: Result after checkin for creating a new fork

However, the main difference is that if we now fix the bug in line 10, by initializing the `currentfont` variable, this change will still be included when we do a projection on the fork code. This decreases the chance of missing important fixes and solves partially the decentralization of information problem. For example, the variation control system could easily output the code introduced by forks and compare it to the mainline.

There are two main differences when using forking support. First, the *get* function will by default exclude any forks from the view. In practice, it uses the projection given by the user and negates all existing forks, which are tracked by the system. To work on a fork, the user needs to specify which fork should be projected. Let $\mathcal{K}$ be the set of all forks defined in the variation control system.

Therefore, the definition of get becomes:

$$\forall c \in \mathcal{C}.\, get(c, r') = \begin{cases} get(c, v'), & \text{if } \textit{SAT}(c \wedge p \wedge \neg f) \text{ when no fork is specified and} \\ & f = f_1 \wedge ... \wedge f_n,\ f \in \mathcal{K} \\ get(c, v'), & \text{if } \textit{SAT}(c \wedge p \wedge \neg f) \text{ where} \\ & f = f_1 \wedge ... \wedge f_n,\ f \in \mathcal{K} - f_i \text{ and} \\ & f_i \text{ is the fork specified by the user} \\ get(c, r) & \text{otherwise} \end{cases}$$

Second, when a user specifies that the changes belong to a fork (e.g., using parameter $-c\,[fork\_name]$), then the ambition a' will be the conjunction of the fork's name given by the developer and the specified ambition. The put update function is the same as before, except of the new ambition a'.

$$\begin{aligned} a' &= a \wedge fork\_name \\ put(p, a', r, v') &= minimize(F\langle v', r\rangle) \\ F &= (p \wedge a') \end{aligned}$$

Adding forking capabilities in a unified way with preprocessor annotations has several advantages. Internally, the same technique (preprocessor annotations) is used to add first class support for forking. Having centralized code, the tool can show more

information about forks and their changes. In addition, propagation of changes from the integrated platform to the fork is not needed. Combining preprocessor annotations and forking in a unified way makes it possible to support the development of software variants with two of the most used techniques. This unification is a step forward for validating *H1* and answering *RQ2*.

> *RQ2: A variation control system can unify and combine preprocessor annotations and forking, leveraging their advantages and reducing some of their known drawbacks.*

6.8 Discussion

Edit Operations for a Variation Control System. Some of the identified edit patterns were difficult to replay using the variation control system workflow and the get/put functions. However, the edit patterns should not be seen as the edit operations a developer would use when using a variation control system. The edit patterns are used to derive, where needed, the *edit operations* for a variation control system. Most patterns can be used in a straightforward manner and do not require specialized operations. However, a variation control system would require a specialized edit operation for renaming and changing a presence condition. Better support is also needed for P12 UnwrapCode and P14 MoveElse patterns, as an extra copy-paste editing step is required. These would require more specialized primitive operations, ideally in a text editor or IDE.

Finally, the variation control system has one limitation that is not solved by any of the existing ones. The generalized edit isolation principle (cf. Sec. 6.3) raises the following problem: How to handle the cases when an ambition is *weaker* than the projection? An example scenario could be fixing a bug in a particular variant, where the fix might affect other variants as well. So instead of fixing the bug in all variants, we would like to have a specific projection, but then perform the change with a weaker ambition. The definition of the put function (which conjoins projection and ambition) cannot handle this case. Solving this problem in a sound way is complex and is subject to future work.

Challenges of Using the Variation Control System. One of the biggest challenges of this prototype is that the editing workflow is different. This requires some mental effort in understanding what projection and ambition to use. However, in my experience, projections and ambitions were sufficiently simple to be specified. In general, choosing a projection and ambition is straightforward, but I experienced difficulties for changes that required two or three checkout/checkin cycles.

An interesting case to consider is choosing the ambition when making code optional, that is, wrapping existing code with a presence condition. Both P11 WrapCode

and P5 AddIfdefWrapThen required the ambition to be the negated desired presence condition. The intuition is that the we have to choose an ambition that describes in which configurations should the code not appear. This may seem unintuitive at first, but it is easy to see why this is necessary. In a text editor or IDE, the user could select the code that should be under a presence condition with the mouse, right click and select an option from the context menu through which the new presence condition is entered.

> *Three identified editing operations are not well supported with this variation control system. Specialized editing primitives are needed for those operations.*

To this end, I believe that more concrete support through a text editor plugin or an IDE is desired. It will support the editing operations which require first hand support.

Metrics and results The boxplots in Fig. 6.6 show that on average there is not much of improvement in terms of LOC and NVAR. The reduction factor is highly dependent on the way `#ifdef` annotations are used in the code and the usage of partial configurations. However, if a complete configuration of the tool would be used, e.g., the tool could accept a partial configuration but transform into a complete configuration on the user's request and eliminating all the variation points that are not related to the change, then it is likely that these metrics would change considerably. The challenge though, is to have a workflow and tool that does not create overly complex presence conditions and preserves the semantics of the code, but also the properties of the current variation control system.

6.9 Summary

In this chapter I have presented the design of a tool that combines preprocessor annotations and forking to offer enhanced flexibility, lower adoption costs, and decrease maintenance. This is one of the first systems that tries to use forking in a combination with preprocessor annotations to maximize flexibility for developing software variants. As shown in the previous chapter, forking is one available reuse mechanism for quick development and experimentation. However, it is widely used as a way to branch new projects from existing ones, creating new variants. The variation control system deals with two main issues that forking has: increased maintenance costs and decentralization of information.

First, a main challenge of forking is that changes are not propagated back and forth in the repositories. However, using the variation control system to create a fork, common code that is updated does not have to specifically be propagated to the fork.

Second, instead of having artifacts split across different repositories, the variation control system centralizes all the assets into one monolithic repository. Special commands can allow for creating forks or editing the code in those forks. Having all the

assets centralized in once place, it should be easier to find and manage assets, reducing the risk of re-implementing existing functionality. This is important as it reduces development costs.

Through an empirical study I have shown that a variation control system can support most general editing operations that developers perform in highly configurable systems. Although we can execute most operations, experience with such a system is limited at the moment. Therefore, more studies could be invaluable to understand more in-depth the intricacies of developing software variants with a variation control system. Nevertheless, this is a first step towards providing first class support for variant development at the code level, to reduce complexity, separate concerns, and modularize code through the usage of projections.

Until now, in this chapter I have only shown how an existing system can be developed and maintained, and also how can we use forking for quick development. One important challenge is to integrate forks into the integrated platform, to reduce costs and improve the platform. Similarly, it is also challenging for independently developed variants to be integrated into a common platform. This challenge I address in the following chapter, where I present a prototype for performing variant integrations using high level intentions.

7. Variant Integration using Intentions Paper D

This chapter shares content with the paper "Intention-Based Integration of Variants" that is currently under review. This work was done in close collaboration with Max Lillack and a few other researchers, thus I will use the first person plural form to present the work we have done. For complete details, refer to the paper in Appendix A.4.

In the previous chapter, I have presented a variation control system that leverages the advantages of preprocessor annotations and forking, and improves some of their drawbacks. One remaining challenge is to re-engineer either existing forked variants into an integrated platform, or to integrate back a fork into the main platform. In this chapter, I present how we can ease the developer's integration tasks. Using specialized integration tool support combined with specific integration-based intentions (abstractions of task goals), I show how developers can integrate forked variants with less manual work and fewer edits.

7.1 Introduction

Integrating forked variants or features back into an integrated platform is challenging. If the variants have not been synchronized and have evolved separately, then the differences between the codebases make the integration process non trivial. In the previous chapter, I have partially answered *RQ2* In this chapter, I answer the part of *RQ2* that refers to the integration of variants into an integrated platform. The key idea is to ease the integration of software variants into an integrated platform with specialized tool support. Integrating variants improves visibility, reduces maintenance costs and centralizes assets into one location.

This chapter has the following contributions: 1) a set of intentions to support general integration tasks of developers, and 2) it presents a series of simulations of real world integrations, showing that users using the tool do less mistakes and require less operations to achieve the tasks.

```
1  #ifdef ULTIPANEL
2   uint8_t lastEncoderBits;
3   uint32_t encoderPosition;
4   #if PIN_EXISTS(SD_DETECT)
5    uint8_t lcd_sd_status;
6    #endif
7  #endif  // ULTIPANEL
8
9  menu_t cM = lcd_status_scrn;
10  bool ignore_click = false;
```

```
1  #ifdef ULTIPANEL
2   uint8_t lastEncoderBits;
3   int8_t encoderDiff;
4   uint32_t encoderPosition;
5   #if (SDCARDDETECT > 0)
6   bool lcd_oldcardstatus;
7   #endif
8  #endif //ULTIPANEL
9
10 menu_t cM = lcd_status_scrn;
```

Figure 7.1: Code excerpts from Marlin's main codebase (left) and the corresponding fork (right). Colors indicate differences.

Variant Integration Challenges

Integrating forked variants into a configurable platform is a difficult, time consuming, and error-prone process. Consider the code in Fig. 7.1 split into two; on the left hand side it is an excerpt from the Marlin mainline repository, while on the right hand side there are changes done in a fork. Both variants have evolved after forking; the highlighted lines were either changed or added.

Let us consider how the two codebases from Fig. 7.1 could be integrated. The fork's line 3 (added variable `int8_t encoderDiff`) could be copied and kept as it is. It can also be made into an optional feature. For example if the line is annotated with an expression that specifies that the line belongs to a feature. The developer may also decide if this feature is the same as for the fork's lines 5-7 based on his expertise. Another interesting aspect in this example is the ordering of code blocks. From an integration point of view, in this case, the order of these variable declarations does not matter. However, this is not generally valid and ordering does matter in most cases. If these were statements, it might be necessary to enforce a certain order.

There are several steps that need to be completed to successfully execute an integration task. First, domain knowledge is important and usually tools can offer little help in this first step. A developer has to understand the changes and differences between the codebases to correctly execute the task. Second, the developer needs to take decisions about the code and its structure. Having two or more codebases that need to be integrated, there are several possibilities of merging the code. For example, the developer could choose to enclose the differences between the two codebases using **`#if`** annotations [Jepsen et al., 2007]. Another possibility is to integrate modifications from either codebases via copy-pasting, while using a diff tool. However, this method might quickly become complex particularly when errors have been made and a developer would want to undo the changes. The undo operations from text editor are at character level, and do not align well with the intentions of developers who deal with integrating tasks. When complex changes are involved, diff tools are not suffi-

cient to reason about the task, and do not make the task much easier for developers. A developer would still have to copy paste, add or delete code, or to introduce annotations manually to allow the code to be configurable. This is a time consuming and error-prone task.

Integrating software variants is challenging due to the following reasons:

- Manual editing is an error prone process
- Difficult to undo changes due limited undo capabilities of general text editors and IDEs
- Working at code level instead of an abstraction for the integration goal

In this dissertation, I will focus on integrating two codebases. Performing n-way integrations (regardless if models or code is merged) is a known and difficult problem [Rubin and Chechik, 2013b, Mens, 2002], and is left for future work.

7.2 The Integration Process

In this section I present what steps are needed to perform an intention-based integration. We consider two variants, a mainline and a fork. The process is realized in two steps.

1. Merging two codebases into an integrated codebase. We automatically create an integrated codebase from the mainline and the fork variant. The integrated codebase captures the commonalities and differences between two variants. When there exists a difference between the two variants, this difference is introduced as in `#ifdef` block with the `FORK` presence condition.

Figure 7.2 illustrates the creation of the integrated codebase (in concrete syntax) for the running example from Fig. 7.1. We can derive, compile and execute individual variants at any time during the integration process, e.g., to run a test suite. Since this integrated codebase is likely hard to understand and maintain, the developer can adjust it to obtain a well-structured platform in subsequent steps.

2. Exploring and editing the integrated codebase The developer explores the integrated codebase and navigates through it using different *views* that ease comprehension of the variants and their differences. We provide five different views, four of them being illustrated in Fig. 7.3. Using these views, the developer can see how a change affects the individual variants, and the resulting integrated platform.

The *integrated side-by-side view* (bottom left of Fig. 7.3) renders the common integrated codebase, but with the differences between mainline and fork arranged next to each other without `#ifdef` directives. The *mainline view* and the *fork view* (top left

```
1  #ifdef ULTIPANEL
2   uint8_t lastEncoderBits;
3  #ifdef FORK
4    int8_t encoderDiff;
5  #endif
6   uint32_t encoderPosition;
7  #ifdef FORK
8   #if SDCARDDETECT > 0
9     bool lcd_oldcardstatus;
10  #endif
11 #else
12  #if PIN_EXISTS(SD_DETECT)
13   uint8_t lcd_sd_status;
14  #endif
15 #endif
16 #endif
17
18 menu_t cM = lcd_status_scrn;
19 #ifndef FORK
20  bool ignore_click = false;
21 #endif
```

Figure 7.2: The common integrated codebase for the mainline/fork of Fig. 7.1

and right of Fig. 7.3) render a *projection* of the integrated codebase where only a subset of the derivable variants are shown. The *mainline view* shows all variants with feature `FORK` disabled, and *fork view* with this feature enabled. In these views, the developer can define intentions (explained shortly), and explore and edit code. The *result view* (bottom right of Fig. 7.3) renders a preview of the final result, after all intentions are resolved and any manual edits applied.

The developer edits the integrated codebase (using one of the *integrated views*, the *mainline* or the *fork view*) and declares intentions on code blocks (individual lines, multi-lines or entire blocks of code). Whenever a change is made, all views are updated synchronously. Developers decide how to deal with variant differences and how to prepare code for evolving it as a configurable platform (e.g., creating features that can be included or excluded at compilation time or renaming a variable, and other kinds of edits). The user interface supports navigating the variant differences, and declaring intention on selected code. Declaring intentions relieves developers from creating or adjusting **`#ifdef`** structures along with the associated presence conditions. The *result view* is updated when an intention is added or removed on a node. Finally, the developer commits the intentions and the integrated codebase is updated to incorporate all the selected intentions.

7.3 Intentions

Intentions are a way to model generic integrations goals as abstractions from code. When a developer proceeds to execute an integration task, the developer should have a goal or an *intent* for each difference between the codebases. Intentions are better suited

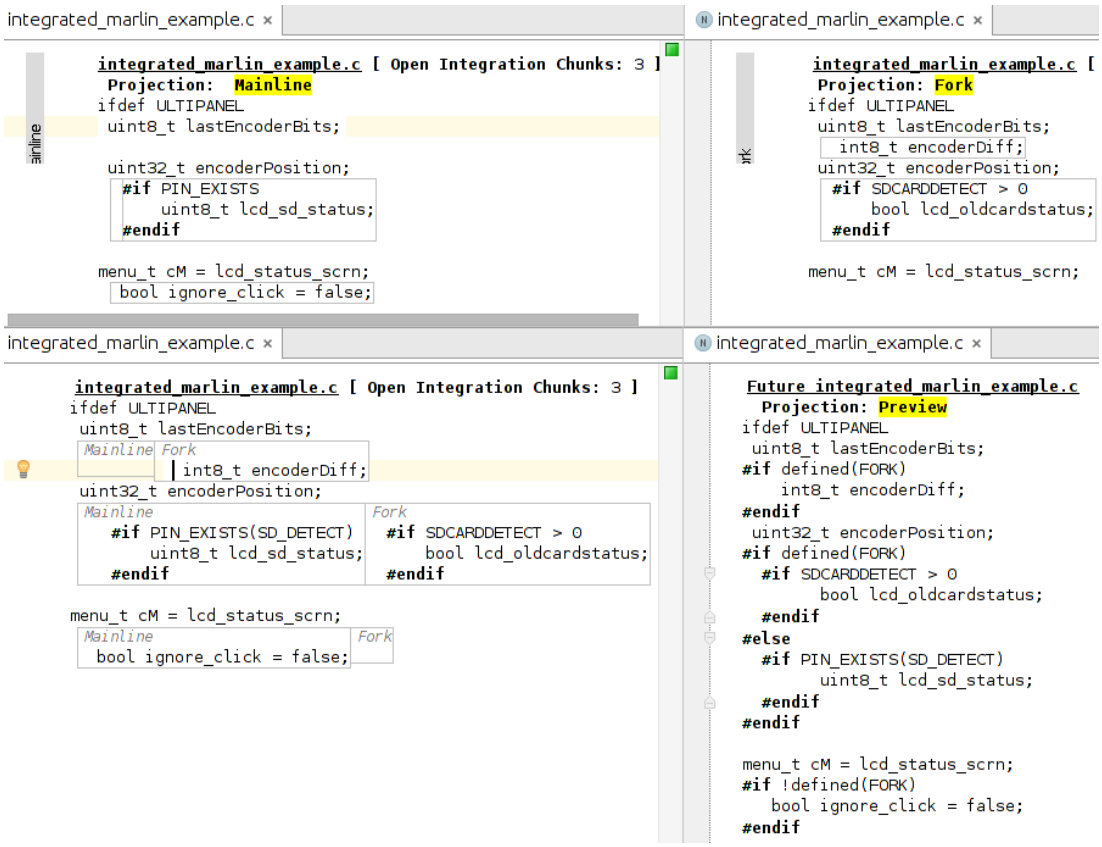


Figure 7.3: INCLINE showing the common integrated codebase of Fig. 7.1. Four views are shown. The top two views are projections of the integrated codebase; the top left is the code that exists in the mainline variant, and the top right is the code from the fork variant. The bottom left view shows the common integrated codebase, with side-by-side differences (gray boxes). The bottom right is a pre-view, showing how the future common integrated codebase will be affected by the intentions applied.

for performing different integration tasks, as they provide general common goals of integrating code. The key aspects of tool-based integration intentions are:

- Intentions abstract from the code level with the goal of providing a skeleton for the integration goal
- Tool supported integrations with intentions let the developer focus on the actual goal and not only on the code
- With an intention-based integration tool, developers need to do less editing and have better undo facilities. For example, a developer can undo a single intention even after other several intentions and changes have been done, which is much more difficult to achieve in a character-based undo tool.

```
#ifndef FORK // block_not_fork
 int servo_e1[] = SE
 int servo_e2[] = SEA
#else // block_fork
 int16_t servo_e1 = SE
 int16_t servo_e2[] = SEA
#endif
```

```
 int servo_e1[] = SE
 int servo_e2[] = SEA
#ifdef FORK
 int16_t servo_e1 = SE
 int16_t servo_e2 = SEA
#endif
```

Figure 7.4: *Keep* intention (left) and result (right)

Exploratory study To design and understand the type of intentions needed to integrate real world software project variants, I have analyzed a number of scenarios from Marlin. These scenarios are extracted either from pull requests that were merged, or from forks that were integrated based on previous knowledge of the project [Stănciulescu et al., 2015]. Once I identified a number of similar scenarios, I proceeded in understanding the developer's intention and how this could be extracted. After several iterations of this process, I have determined seven intentions that are specific for the integration process: *Keep*, *KeepAsFeature*, *Exclusive*, *Remove*, *AssignFeature*, *Postpone*, *Order*. I describe in detail the most common three intentions in the next section. The rest of them and the formalization of their semantics is fully described in A.4.

Intentions

Intentions are partial functions transforming an abstract syntax tree (AST). For instance, Fig. 7.4 shows the integrated AST in a textual representation on the left, selected nodes on which an intention is declared in gray, and the desired result on the right. In this section, the examples show all variants at once as it is more suitable to explain how intentions work directly on source code. I will show three most used intentions *Keep*, *Remove*, and *KeepAsFeature*. The other intentions, formalization of all intentions, and their semantics are presented in detail in Appendix A.4. In Appendix A.7, four intentions have complete examples extracted from real code.

Keep The *Keep* intention includes a block as it appears in mainline or fork in an unconditional manner, without guarding it with any additional feature.

Consider the example in Fig. 7.4 where we define *block_not_fork* to represent the set of nodes in the $\neg FORK$ branch highlighted with gray, and the *block_fork* represents the set of nodes in the *FORK* branch. The fork changes the type of the *servo* variables to be 16-bit signed integers, because different hardware and compiler are used for this variant. During the integration process, it is decided that the hardware used in the fork should no longer be supported, and only the code from mainline is kept. We apply the *Keep* intention on *block_not_fork* set of nodes. The right side shows the result of applying *Keep* on the selected *block_not_fork*. The nodes from the fork should be removed (which can be done with the dual of *Keep*, *Remove*, described below). Note

```
 int servo_e1[] = SE
 int servo_e2[] = SEA
#ifdef FORK
  int16_t servo_e1 = SE
  int16_t servo_e2 = SEA
#endif
```

```
int servo_e1[] = SE
int servo_e2[] = SEA
```

Figure 7.5: *Remove* intention (left) and result (right)

that the integration is not completed, as there is still a block from the forked variant, which should be resolved later.

The nodes for which *Keep* was declared should no longer be under the constraint created by the **#ifdef** that directly wraps those nodes (in the example we drop `!FORK`). Their new presence condition is the conjunction of all but the last constraint that directly wrapped the nodes. All nodes that are not part of the intention are unchanged.

Remove This intention removes the selected nodes from the AST. Its definition is simple; we ensure that the selected nodes do not exist in the updated AST. This can be seen in Fig. 7.5.

KeepAsFeature The *KeepAsFeature* intention preserves a block from one of the variants, but makes it conditionally present, only linked to a certain feature or combination of features. It wraps the block with a new presence condition given with the intention.

In the example of Fig. 7.6, a fork developer added functionality to pause a 3D-print from an SD card. Not concerned with other devices than the one for which the fork was developed, the developer included the new behavior unconditionally. However, in the integration process, it became clear that this functionality only makes sense in variants supporting SD cards, thus, it needs to be included conditionally. The desired result is shown on the right side of the figure.

Tool Implementation

We use the language workbench JetBrains MPS [mps,] on top of which we realize our tool INCLINE. MPS relies on projectional editing, which is well-suited for creating editable views. Projectional editing (a.k.a., syntax-directed editing or structural

Figure 7.6: *KeepAsFeature* intention (left) and result (right)

```
#ifdef FORK // block_fork
  card.pauseSDPrint();
#endif
```

```
#ifdef SDSUPPORT
 card.pauseSDPrint();
#endif
```

editing) [Völter et al., 2014, Berger et al., 2016] is conceptually different from traditional parser-based editing, since the user's program-editing gestures directly change the underlying AST, which is still rendered into concrete syntax the user sees. Projectional editing eases language composition and allows flexible notations (e.g., the *integrated side-by-side view*).

We make use of MPS' meta-modeling facilities and implement our own stripped version of the C preprocessor language, including only the **`#ifdef`**, **`#else`**, **`#if`**, **`#elif`**, **`#endif`** macros. For example, we define **`#ifdef`** as a language concept that defines a node containing a condition and having three child nodes for the nodes in the *true*, *else if*, and *else* branches. In addition, we add a `Text` concept that represents a line of actual source code (C/C++ in our case) in the AST. Each of the concepts can have different properties, e.g., the **`#ifdef`** concept has a *condition* attribute that holds the presence condition associated with this **`#ifdef`**. These language concepts suffice to represent source code files that use preprocessor annotations to implement variability.

We specify how the user can interact with our language, that is, edit the AST using the *editor* functionality of MPS. For each concept we create a visualization definition, which controls the rendering of it. The intentions are implemented using MPS *actions*, user-invoked commands to change the AST, selectable via the UI.

We implement three additional components. PPParse is a parser for C preprocessor directives that creates the initial AST, based on the clang compiler infrastructure. PPMerge is a tool to create the integrated AST of two files using the C preprocessor. It first parses the input files to create XML-based ASTs, then constructs the integrated AST using JNDiff [Di Iorio et al., 2009], which diffs the ASTs, followed by transforming the diff into optional changes in the input file. PPConstraintSolver is a tool to perform operations on preprocessor conditions using the SMT solver Z3 [Moura and Bjørner, 2008]. Specifically, we use it to calculate projections, i.e., to decide *if* and *how* conditions are shown in the views.

Figure 7.3 shows a screenshot of the tool. The user can select a four-view window for the integration process. On the top left part of the window, there is the view for the initial integrated platform created from two files. In this view, we can also see two boxes: one that has a tag *mainline* and one that has a tag *clone*. These two boxes represent the different lines of code that exist in the mainline and respectively in the fork, similarly to a diff tool. On the top right part of the window, we have a projection on the fork code. That is, we use the projection *FORK* to get a view for the fork. On bottom right part of the window, there is the view corresponding to the projection $\neg$ *FORK*, showing the mainline code. Finally, in the bottom left part of the window, we have the view responsible for previewing the result when applying different intentions.

7.4 Evaluation

To study how well INCLINE supports developers during variant integration, we conduct a series of simulations of real development. We conceive integration tasks from real evolution steps of the three often forked open-source systems Marlin, Vim, and BusyBox, as opposed to artificially creating tasks—a reasonable trade-off between complexity and real-world cases.

Subject Systems

All subjects use preprocessor annotations to realize variability. We enhance external validity by sampling over their source files and forks (>4,000 Marlin forks exist). Our selection of forks (three Marlin forks, one fork each from BusyBox and Vim) is based on the fork's activity, viability (i.e., has variability-related changes), and popularity (stars on Github). This way, we avoid bias towards a particular usage of the preprocessor. In addition, a study including Vim and BusyBox confirms that the preprocessor is used similarly among open-source and industrial systems [Hunsen et al., 2015].

1. Marlin (>40 KLOC of C++ code) has many forks that evolved separately or independently added new functionality. Given this richness of changes and new functionality, and existing re-integration efforts of the community, Marlin is an ideal subject for our evaluation. We mined from 1,715 approved pull requests and three forks based on my knowledge and understanding of the project. The three forks are as following: jcrochollMarlin, a variant for the deltabot printer; esenapajMarlin is a variant to support the Arduino Due board, based on a 32-bit ARM microcontroller; Marlin_STM32Marlin is a also variant that attempts to port Marlin to work on ARM microcontrollers. Using more powerful boards with more computation power (such as the ARM microcontrollers), has been one of the requests from the Marlin developers, thus these fork constitute a good selection for our purpose of integrating variants.s

2. BusyBox (>160 KLOC of C code) is a tool suite of common shell programs (e.g., grep, cut). We use a fork tailored for Android. [1]

3. Vim (>300 KLOC of C code) is a popular console-based text editor for Unix-like systems. We use a fork that adds support for OS2. [2]

Study Design

Evaluating an interactive tool with a rich UI such as INCLINE is challenging, since users do not only face the inherent complexity of the integration task, but also have to

[1] `https://github.com/jcadduono/android_external_busybox`
[2] `https://github.com/h-east/vim/tree/clpum`

learn the tool and handle potential usability issues of a research prototype. We conduct a series of simulations, going from an internal evaluation via a preliminary user study with MSc students to a controlled experiment with experienced PhD students in a realistic integration setting.

First, we verify the completeness of our intentions. We replay a set of real-world merge commits from Marlin by applying intentions and checking that the commits are realizable. We also check that the result is semantically identical (i.e., code lines obtain the correct presence condition) to the original merge commit.

Second, we validate the correctness of the implemented intention resolutions and investigate the usability and scalability of INCLINE. Three authors simulate ten Marlin integration tasks and thoroughly cross-check the correctness by reviewing the resulting integrated files. While this does not allow comparing the efficiency, it validates that INCLINE produces correctly integrated files if intentions are assigned correctly, and that it scales. It also provides valuable experiences to improve INCLINE and to fix bugs.

Third, we investigate the INCLINE approach with 16 MSc students to learn how they perceive the intentions, the views, and the editing efficiency and usability of our tool. We conduct a user study where we create realistic, but reasonably small tasks from Vim and BusyBox and let participants solve them using INCLINE and Eclipse. Since the participants lack domain knowledge for the integration tasks, we give the final result. As such, this experiment also lets us obtain information about the pure editing efficiency with INCLINE (recall its underlying projectional editor, which can cause editing challenges [Berger et al., 2016]), potential improvements, and bugs.

Fourth, after improving INCLINE with insights from the steps above, we validate INCLINE in a realistic setting. This validation is aimed at testing the integration part of *H2*. We conduct a controlled experiment with 12 experienced PhD students. We reuse the Vim and BusyBox tasks, but instead of providing the final result (which, as we learned from the user study, lets developer just apply low-level edits without understanding what they are doing), we provide detailed domain knowledge about the variants to be integrated. Tasks are solved with INCLINE and Eclipse using a 2x2 Latin square design. This controlled experiment lets us obtain information about the efficiency of integrating smaller files.

In summary, this set of consecutive experiments lets us obtain information about the benefits and limitations of our approach. As we will show, the scalability of INCLINE to files of up to 4K LOC, together with the benefits measured for integrating smaller files, evidences the applicability of INCLINE to smaller and larger integration project. Yet, we believe that training and experience can further increase the efficiency of developers working with INCLINE, which needs to be validated in a longitudinal study. Such a study is beyond the scope of this dissertation, but part of possible future work.

Completeness, correctness and scalability

Completeness To show that the defined intentions are sufficient to handle real-world integration tasks, we replay non-trivial (conflicting) merges from Marlin history. We retrieve all 2,065 merge commits of the mainline, and extract those that had conflicts, to identify complex merge tasks, yielding 49 merges. We discard two merges that had conflicts only in documentation files, two that conflicted in whitespace, three that conflicted due to configuration changes. Another three merges are discarded because some related artifact had syntax errors and could not be compiled. Additionally, four merges are discarded because they simply accepted the mainline changes as evolution (empty changeset). We use the remaining 35 merge commits as tasks.

We successfully simulate all 35 commits using intentions.

Examples mined from Vim and BusyBox can be handled by our intentions as well, as I will show shortly in the next section.

> The set of intentions suffices for real-world variant integration.

Correctness and Scalability With an internal study we validate that INCLINE produces correct results when intentions are assigned correctly, and that we can use it on large files without scalability problems.

We simulate ten integrations by randomly sampling seven commits tasks from the 35 merge commits in the previous experiment, and conceive three tasks simulating the integration of files from Marlin forks.[3] [4] [5]. The selected forks contain significant changes to the mainline, covering both evolution and new features.

Most tasks comprise only a single file, some two or three, but each file can be very large (up to nearly 4,000 LOC and hundreds of `#ifdef` blocks in a single file). The first three authors serve as evaluators, among whom we distribute the ten tasks to execute with INCLINE. We then manually peer-review the integration results to detect any errors that INCLINE might have introduced. For the seven tasks based on merge commits, we compare to the actual merge result. For the three tasks based on the fork integration, the correct results was determined during the peer review.

All of the observed errors could be explained by errors done by the user or errors introduced by the tool. We fixed errors in the implementation and analyzed mistakes done by us to improve usability. This shows that the intention resolution, as defined in Sec. 7.3, works as expected. Furthermore, the broad range of file sizes (from tens up to thousands of lines of code) evidences the scalability of INCLINE.

[3] `https://github.com/jcrocholl/Marlin`
[4] `https://github.com/esenapaj/Marlin`
[5] `https://github.com/Marlin_STM32`

INCLINE produces the correct output when correct intentions are applied and it scales to files up to 4K lines of code.

Using INCLINE with students

We continue with a user study to get first-hand experience how users deal with INCLINE and its UI. We recruit 16 MSc students to execute two newly developed integration tasks with INCLINE and Eclipse.

The two new integration tasks are derived from BusyBox (P1) and Vim (P2), by using files from the main project variant as well as a fork of each project (see Sec. 7.4). We select chunks of code based on our understanding and experience with the systems, as well as code blocks that involve integrating variability (`#ifdef` blocks). We merge these chunks into one condensed file for brevity and comprehension. The end result is an integrated file consisting of 74 LOC (P1), and 50 LOC (P2). The P1 file has 8 `#ifdef` blocks, with 37 LOC within these blocks; the P2 file contains 5 `#ifdef` blocks, with 32 LOC within these blocks. These tasks represent a good trade-off between complexity and real-world integration scenarios.

The participants are given the (correct) target solution and brief description of the integration goal. Giving the target solution reduces the influence of (a lack of) domain knowledge because all participants work towards the same goal. On the other hand, it is less realistic because the participants do not need to understand the example code or think on the level of integration goals. We observe their usage of the tools (through screen recordings), and at the end we ask them to take part in an exit questionnaire.

Results We observe that INCLINE users only need to declare a few intentions to reach their desired result, whereas Eclipse users use the keyboard much more heavily. INCLINE users mostly used the intentions *Keep* and *Remove*, since they are the most obvious intentions for new users and they were sufficient for the selected tasks. The user's behavior and their feedback suggest they miss more advanced functionality, e.g., the side-by-side view which was disabled by default. We also learn that the UI needs better highlighting of the intentions that were applied, that keyboard shortcuts would help to quickly apply an intention (the users needed to use a menu button to reach the intentions), that the multiple views help exploring the code to reach consensus on how to integrate it, and that the arrangement of views shown in Fig. 7.3 is most intuitive.

Not surprisingly, the Eclipse tasks are solved faster, since we provided the final result, and as such, solving the integration task amounts to straightforward, low-level editing. The users hardly need to take high level decisions and map them to low level edits. Analyzing the recordings of the Eclipse tasks in fact shows that the participants seem to thrive with the direct low-level editing of the code. They quickly start copy-pasting code, and introduce preprocessor annotations to match their solution to the

developers		*programs*	
		P1	P2
	D1	CDT	INC
	D2	INC	CDT

Table 7.1: Latin square design. Each developer executes in a random order the P1 or P2 task, with the two treatments: Eclipse CDT or INCLINE (INC).

actual target solution. INCLINE users are slower as they need to understand the integration solution and map it back to intentions first, which is more demanding. In summary, this shortcoming in plain editing efficiency illustrates an important limitation of our approach. Yet, even in INCLINE, developers could resort to writing preprocessor annotations by hand, making up for this limitation, a possibility about which we did not instruct them.

Controlled Experiment

We conduct a controlled experiment with 12 experienced PhD students who are familiar with the C preprocessor. We adapt the situation where a developer who has domain knowledge about the variants shall integrate them to simulate a realistic setting. To this end, we provide a detailed, but abstract explanation of the purpose of the variants' individual parts and how they should be integrated.

We use a 2x2 within-subjects counterbalanced Latin square design and reuse the Vim and BusyBox tasks. That is, each participant performs two tasks, using two treatments: Eclipse, and INCLINE on P1 or P2, in a random order to reduce learning effects (see Table 7.1). Using a within-subjects design, we can have a lower number of participants, while every subject participates in each task. Furthermore, we mitigate learning effects by randomizing the order of the tasks (counterbalanced part of the design).

Participants are trained through a video tutorial on how to use both tools, as well as being instructed on preprocessor usage (they only needed to use `#ifdef`, `#else`, and `#endif`). Then, we asked the participants to solve a warmup task extracted from Marlin, to get familiar with the tools. We record the screens and log information about keystrokes (in Eclipse) and intentions (in INCLINE).

We compare the performance of participants with both tools by measuring the mistakes done per task, the time to complete each task, and the number of edit operations (and number of intentions) applied per task as a proxy measure of effort.

We count mistakes done by the participants as follows. For Eclipse, we check if the end result is the same as the expected result. A mistake can be a missing preprocessor annotation, missing code or extra code. For INCLINE, we check for wrong intentions or no intentions applied by the participant that leads to errors in the resulting file.

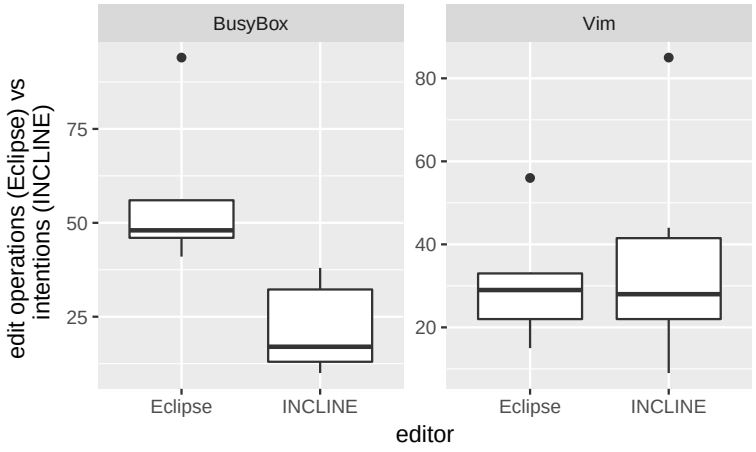


Figure 7.7: The edit operations/intentions needed to execute the BusyBox and Vim tasks using Eclipse and respectively INCLINE

For both tools, errors concerning comments are counted as a half mistake, errors in formatting of code is ignored.

Results INCLINE participants made fewer mistakes than participants using Eclipse (7 vs 17.5). Furthermore, only four participants did mistakes in INCLINE compared to 11 participants in Eclipse. This is no surprise, as INCLINE has better support for keeping or removing code without needing to copy&paste or create `#ifdef` structures (a good source of errors). Users made mistakes in INCLINE when they missed relevant nodes in the declared intentions, declared incorrect intentions, or declared different intentions for the same node with an unexpected result for the user. Common mistakes with Eclipse included failures in the `#ifdef` structure, leaving code that should be removed or removing too much code.

INCLINE is also better in terms of intentions needed to execute the task (see Fig. 7.7). In BusyBox, INCLINE users need only a handful of intentions to integrate the two variants, whereas BusyBox requires almost 50 edit operations for achieving the same goal. In perspective, Vim is less demanding for Eclipse users, and INCLINE users use a very similar number of intentions to execute the task.

If we compare the times to execute the task, INCLINE integrations are almost as fast as the ones in Eclipse (see Fig. 7.8). The reason for being slower is twofold. First, participants spent a lot of time (which we count in the result) reading back and forth through the descriptions to understand their integration goal. Second, some participants were always verifying the preview view after applying an intention. A possible explanation for the latter case is that users are not very familiar with the tool and intentions, and thus either do not trust the tool or are not sure if they applied the right intention. However, this is exactly where INCLINE shines as it facilitates exploration,

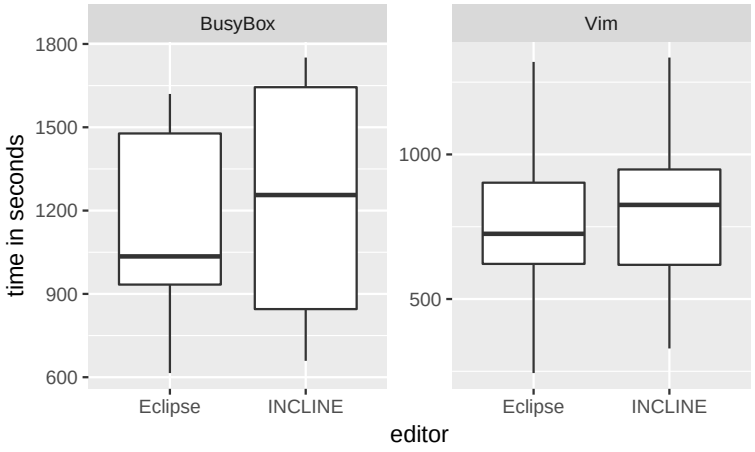


Figure 7.8: The time needed to execute the BusyBox and Vim tasks using Eclipse and respectively INCLINE

quick undo, and offers multiple views for manipulating and understanding the different variants and the integration result. Along these lines, one of the participants mentions that *"It was really useful to declare all the intentions while still having the original files in sight and previewing the result."*

INCLINE users are almost as fast as Eclipse users, but perform much fewer mistakes.

Post-experiment survey

After both user studies, we asked our participants to fill in a survey questionnaire. We received in total 25 responses.

Figure 7.9 shows the survey questions and the results on a Likert scale. The heatmap representation quickly shows that 56% of the participants consider that integration with INCLINE is faster than with Eclipse. One potential reason is that by not doing many copy pasting operations or editing text, INCLINE feels and sometimes is faster through the usage of intentions. To this end, more than 90% of the participants agreed that the *Keep* and *Remove* intentions are intuitive. Moreover, intention-based integration is not complex, according over 70% of the participants. This is an encouraging result that suggests the potential of using intentions to abstract from code and offer guidance for integration tasks. On the downside, one third of the participants believe that INCLINE is not mature enough, which is to be expected for a first fully working research prototype.

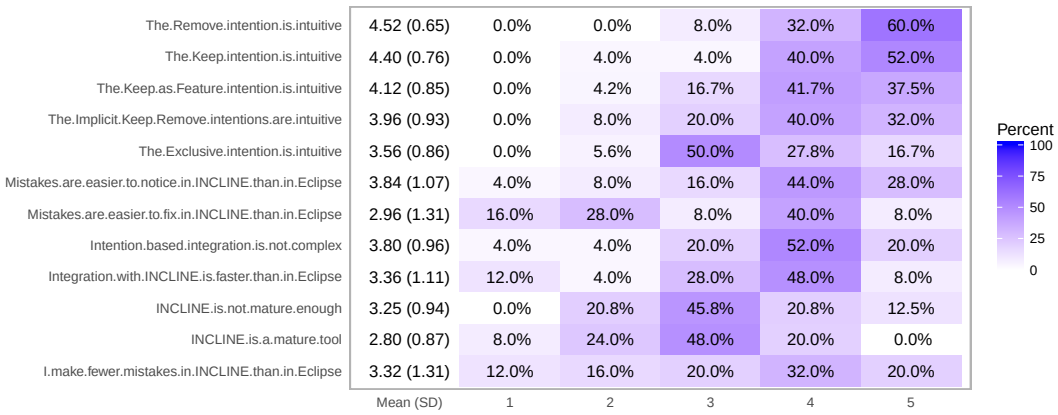


Figure 7.9: Post experiment survey answers on likert-scale in a heatmap representation. The X-axis represents the likert values: 1 being Strongly Disagree and 5 being Strongly Agree. The Y axis shows all the questions in the questionnaire. The stronger color indicates agreement among the participants.

When asked what are the advantages of using intention-based over manual integration, some participants mentioned that *"It doesn't require manual code rewrite so I believe it could be easier avoid unintentional bugs in code and subtle differences"* or *"You get the preview of the result and the projections side-by-side, which seems hugely helpful when you don't have a clear integration goal. Harder to make syntactic mistakes."*. One participant mentions *"It's much more intuitive and is less error-prone."* In general, participants seemed to be satisfied with their first INCLINE experience.

Discussion

This is a first step towards providing concrete support for integration tasks, moving from old fashioned tools like *diff*, that involve manual and error-prone processes. In retrospective, INCLINE suffers from a high learning curve and at times a confusing and immature user interface. This is a challenge as I am not a professional UI designer, nor is any of my colleagues. What I learned from this process, is that when designing a tool it is really important to perform usability studies before releasing the tool to the users. I strongly believe that we could have avoided some critique if we have asked for the services of an expert in user interfaces, to design this more user friendly. Nevertheless, we have seen usability improvements with every iteration of the tool, which is a good indication that this can be further improved.

Although the UI is not perfect, some limitations come from underlying platform that was used, MPS. For example, it is not possible to use the mouse to select a number of nodes, as most users would expect and are used to do in a text editor. Instead, pressing and holding the *shift* key and using the up and down arrows is the way to

perform this selection. It was also impossible to add intentions in the right click contextual menu, which would have made it easier than using keyboard shortcuts or going through a menu and choosing the right intention, which takes time. Looking through the videos, I have found that undoing changes with the default MPS operation causes strange side effects, particularly to intentions and code marked by those intentions.

> Intention-based integration has the potential of being faster and less error-prone than traditional merging of software variants. *H2* is confirmed.

7.5 Summary

In this chapter I presented a prototype for supporting variant integration through high level intentions. These intentions are intended to offer a higher abstraction level from code. We have designed these intentions with a specific goal: to offer general guidance through the integration task. In general, intents are modeled following the natural way of human thinking. Most actions that we perform are impulses of specific intentions.

The INCLINE tool presented in this chapter is a first attempt at offering specialized support for integrating software variants into a common integrated platform for software product line adoption. Through a series of simulations, we have gained valuable insights for the variant integration process. Working with users requires a high degree of quality at the user interface level. Specialized tools like INCLINE are even more challenging to build due to their very specific focus and particular target audience. Nevertheless, INCLINE participants made fewer mistakes and were almost as fast as Eclipse user, while using fewer operations.

Most participants consider that using an intention-based integration tool, they can be faster and make fewer mistakes. This is encouraging as one of the motivation for this work was the high efforts required to execute variant integrations. In particular, copy pasting and editing manually the code can lead to errors and makes it difficult to navigate through own changes due to the general undo operation. Using INCLINE users do not have to deal with preprocessor annotations, and have the freedom of exploring the integration result.

As future work, it would be valuable to experiment and improve the user interface. Engaging with a user interface expert would be extremely useful to help us with the design of our UI. It is likely that after several iterations to improve the UI, the whole experience will be smoother and give the users more trust, leading to faster integration times. In addition, a clear goal for INCLINE is to be used by developers and software product line engineers to perform integration tasks. To achieve this, more studies, preferable with experts from software product line engineering community or companies doing product line engineering, are needed to further improve the tool and understand its usage and benefits.

8. Related Work

8.1 Forking

Several existing works investigate the reasons for forking [Dubinsky et al., 2013, Duc et al., 2014, Robles and González-Barahona, 2012, Mikkonen and Nyman, 2011]. Developers who fork in industrial projects are reported to be motivated by the immediate availability of the method, freedom of control and independence from the old code base [Dubinsky et al., 2013]. Easy access to validated code is also a frequently cited reason [Dubinsky et al., 2013, Duc et al., 2014, Cordy, 2003].

Both in open source software [Mikkonen and Nyman, 2011, Robles and González-Barahona, 2012] and in industry [Duc et al., 2014] forking is also motivated by project organizational matters (who has control over what code). For instance, in commercial projects, the different teams have different road maps for their products, different risk management strategies and different release schedules—all issues that can be circumvented by forking [Duc et al., 2014].

Open source developers may decide to fork in response to needs that are not recognized by the project's maintainers [Robles and González-Barahona, 2012]. A group of developers can fork a project to start a new organization with different priorities and direction than the original one [Robles and González-Barahona, 2012]. As we can see, both in closed and open projects forking centers around adding new functionality, which seems difficult in other ways [Mikkonen and Nyman, 2011, Robles and González-Barahona, 2012, Dubinsky et al., 2013, Duc et al., 2014].

Forking comes at a cost, though. Maintenance becomes more difficult [Dubinsky et al., 2013, Duc et al., 2014], largely because of the lack of traceability between the forks [Dubinsky et al., 2013]. Features and bug-fixes for different code bases are developed independently, therefore duplicated code is created unintentionally [Duc et al., 2014]. A bug that occurs in one code base may exist in others [Dubinsky et al., 2013, Jang et al., 2012], and can remain unfixed [Jang et al., 2012]. Typically the independent teams are unaware of changes that exist in other forks [Duc et al., 2014]. Few forks are merged back into origin and even fewer integrate code from similar or original projects [Robles and González-Barahona, 2012].

One objective of this dissertation was to understand how forking functions as a reuse and variant management mechanism more holistically; not only the reasons for

doing it, but also other aspects such as the extent of synchronization, or criteria for choosing forking vs preprocessor annotations and the life cycle of variants that are eventually integrated. The main finding is that the main reason for using forking is to add new functionality, confirming previous studies. This data was confirmed both in the automatic analysis of changes in the forks, as well as in the survey, as presented in Ch. 5.

In Marlin I have found several important bug-fixes that are not propagated in the project's forks. One interesting observation is that in Marlin, many forks share their changes with the origin, unlike the prior study of Robles et al. [Robles and González-Barahona, 2012] who state that very few forks are merged back and even fewer integrate code from the origin or similar forks. Github's forking and pull request likely play a big role in offering support for sharing changes, thus the different results between this work and the one of Robles et al.

Few studies also mention redundant development [Duc et al., 2014]. In the Marlin study I also confirmed occurrences of redundant development. Moreover, I looked into this problem in more depth and I did a pre-study to verify if this is indeed a real problem and not an artificial one. This study was done using data from GHTorrent [Gousios, 2013], by analyzing the top 500 most forked projects and their pull requests. I randomly sampled 100 *closed* pull requests from the set of tens of thousands of pull requests. I then manually analyzed each pull request to understand the reason for being closed. Approximately 6% of the pull requests were indeed closed due to redundant development. This brings a good evidence that this is indeed a problem, and it will only grow as long as the forking mechanism is heavily used.

Decentralization of information is also a problem that I have encountered in the main study done in this dissertation. Forking enables a simple and controlled collaboration mechanism through the usage of forks and pull requests, but makes it difficult to find new functionality or forks of interest. A variation control system with forking support aims at centralizing the forks' knowledge into one place. In a separate paper that it is not part of this dissertation, together with few other colleagues, we have developed a tool that identifies features developed in forks. To overcome the decentralization of information problem, the tool also extracts an overview of the functionality existing in the forks of a project and summarizes it to the user [Zhou et al., 2018]. This can be used to guide developers to interesting artifacts or forks, while all the information is centralized into one place with easy access to the actual code changes for an in-depth analysis, if needed.

One particular difference in this dissertation compared to many previous studies, is that I focused on only one main study subject, the Marlin open source project. This project follows a typical collaborative community driven development process, with no central organization. This is very different than the industrial processes studied before [Duc et al., 2014, Dubinsky et al., 2013]. Perhaps more importantly, this means that all the data, tools and other artifacts used in this dissertation are available online. Moreover, the study in Ch. 5 differs methodologically combining study of artifacts

(commits, branches, forks, pull-requests), interviewing developers, and surveying both active and passive users of the community. This way researchers working on tools and methods can use this data to evaluate their methods and tools, or replicate findings presented.

Cloning in-the-small

Cloning in-the-small, which is different from forking, meaning that a copy of an artifact is done within a project, can also be beneficial [Kapser and Godfrey, 2006a]. For instance, developers of the Apache web-servers use subsystem cloning to support a large number of different platforms: 51% of code is cloned across the relevant subsystems [Kapser and Godfrey, 2006b]. A study of the projects hosted in the Squeaksource hosting service reveals that 14% of the methods are copied between projects [Schwarz et al., 2012]. Even maintenance is reported to benefit from cloning. Cloning can decrease maintenance risk for program logic, as it allows avoiding any impact on unrelated applications or modules [Cordy, 2003]. Cloning allows to quickly implement a new functionality similar to an existing one. Cordy [Cordy, 2003] reports that in some domains (financial software in his study) cloning is encouraged as it reduces the risk of introducing errors. Experienced programmers clone consciously with intention to reuse knowledge [Kim et al., 2004, Kim et al., 2005]. Still, it is generally agreed that cloning in-the-small introduces software evolution challenges. Roy and Cordy perform an in-depth analysis of clone detection, covering aspects from techniques and tooling, to advantages and disadvantages of cloning and taxonomies of clones [Roy and Cordy, 2007]. Several questions remain open to this day, e.g. if code clones should be removed, encouraged or refactored.

Unlike all the above works in this dissertation the main focus was not on cloning in-the-small, but on forking entire projects. While some drawbacks of cloning in-the-small apply to forking (e.g. fixing bugs in clones), there are some that only apply to forking (e.g. redundant development across projects). This dissertation provides detailed analysis on the advantages and disadvantages of forking in an open source community. The analysis provides a high level perspective, asking process and architectural questions about variant management using preprocessor annotations and forking.

Clone-and-own Support

In more recent years, several works have presented alternative ways to support the ad-hoc clone-and-own for software variant development.

Rubin et al. [Rubin and Chechik, 2013a] present a set of operators used to support clone-and-own. In a subsequent effort [Rubin et al., 2013], the operators are refined through three industrial case studies. The main idea is to verify if proposed operators can be mapped to actual activities performed during a manual re-engineering effort.

The work confirms the applicability of the operators to support the development of variants, and proposes some techniques that can be used to realize the operators. The end goal is to facilitate the development of variants via clone-and-own, and to support re-engineering efforts. However, a main difference is that in this dissertation I take a step further and provide tool support for realizing these goals, while in the previous work there is no implementation that can be used or tested.

Fischer et al. [Fischer et al., 2014, Fischer et al., 2015] take a different approach with the ECCO tool. The idea is to use an extraction and composition mechanism, to reuse existing code from variants developed via clone-and-own. The result is a new product variant that is specified by the features selected. Most of the extraction and composition is done automatically. The user needs to manually verify different hints given by the tool to finalize the product variant.

Rabiser et al. [Rabiser et al., 2016] propose to support clone-and-own prototypes at three granularity levels: products, components, and features. Clones can be created from products, on top of which existing components can be cloned to different prototypes, while compliance levels ensure consistency among the cloned assets.

Schmorleiz et al. propose to annotate source code of variants for a system to automatically keep consistency between variants according to the specified annotations [Schmorleiz, 2015, Schmorleiz and Lämmel, 2016]. Similarly to the idea of Schmorleiz et al., Pfofe et al. use feature expressions to map code to features. Using feature expressions, variants that need to be synchronized can be automatically identified [Pfofe et al., 2016].JSync is a plug-in for Eclipse to detect clones, synchronize the known clones, merge them when needed, and to verify their consistency [Nguyen et al., 2012].

All these works present alternatives to software product line development by enhancing clone-and-own with first class support. Although little is known if these can be used in industrial or even open source projects, they provide another step forward in bringing support for forking to develop software variants.

8.2 Software Product Line Engineering

Adoption

In a recent mapping study on re-engineering variants into product lines, the large majority of the 119 papers analyzed were focused on detecting and analyzing commonalities and variabilities of the variant systems, together with feature identification and location [Assunção et al., 2017]. Actually, only few support the actual variant integration.

Many works focus on re-engineering a single system into a software product line [Schulze et al., 2012,Kolb et al., 2006,Kästner et al., 2007,Kiczales et al., 1997,Fenske et al., 2014], typically proposing refactoring techniques for creating configurable plat-

forms. For instance, Schulze et al. propose to adapt OOP refactorings to feature oriented SPLs refactorings, and define four refactorings suited for FOP product lines [Schulze et al., 2012]. Kolb et al. refactor an existing component to be reused in a software product line and apply semantic-preserving refactorings to the component [Kolb et al., 2006]. Kästner et al. [Kästner et al., 2007] refactor Berkeley DB into an SPL using aspects [Kiczales et al., 1997]. While these works are single-system oriented, the focus in this dissertation is on systematically integrating a set of variants, guiding the process with intentions and views. The main difference is that I focus on integrating software variants originating from clones into a product line, systematically guiding the process with intentions and views.

Rubin et al. present a conceptual framework with seven operators usable to re-engineer cloned variants into a product line [Rubin et al., 2015]. The operators are abstract and some are related to our intentions. Yet, none is implemented and executed on real world systems. Fischer et al. [Fischer et al., 2014, Fischer et al., 2015] propose an extraction and composition method to detect reusable features among variants, allowing to compose them to derive a new system. Martinez et al. present a framework for re-engineering a set of assets into a product line [Martinez et al., 2015]. The framework can be extended and customized to support different kind of artifacts. Klatt et al. [Klatt et al., 2013] present a tool for consolidating cloned product variants. It enhances the initially created integrated platform by providing a variation point analysis that provides recommendations for a developer to aggregate variation points. Fenske et al.'s tool uses clone detection to recover clones among variants in order to lift cloned code to reusable product-line assets by applying refactorings [Fenske et al., 2017]. Around 25% reduction in redundant code could be achieved. Ziadi et al. [Ziadi et al., 2014] automatically create a feature model and a software product line from a set of variants.

Finally, case studies of manual re-engineering exist. For instance, Hetrick et al. re-engineer cloned variants into a product line, extracting core assets from existing codebases, creating variation points, and switching to product line engineering [Hetrick et al., 2006]. Jepsen et al. [Jepsen et al., 2007] compute pairwise differences of two products, and wrap differences using `#ifdef` to create the initial integrated platform. The platform was iteratively refined, deciding to keep, remove or introduce a new feature, which took several years to complete [Jepsen and Beuche, 2009].

Compared to existing approaches, in this dissertation I propose: (1) a set of intentions for various integration goals; (2) editable views to improve variant comprehension and offer a (pre)view to analyze the integration results; (3) doing edits on the integrated platform to explore the effect of intentions and manual edits, with support for undo; (4) a tool chain aimed at C/C++ systems using the preprocessor for variation points, where their correct handling is ensured.

Evolution

Other works provide support for evolving a product line. Montalvillo et al. propose to extend version control system functionality to synchronize core assets and product assets through specific propagation events. These assets are versioned and maintained using Git and Github [Montalvillo and Díaz, 2015]. Different branching models are proposed for core assets and product development, with possibility of synchronizing them when needed. Liebig et al. [Liebig et al., 2015] provide three refactorings (rename identifier, extract function, inline function) that are proven to preserve the variants in a configurable platform. The resolution of our intentions can also be seen as a refactoring, but it is explicitly not variant-preserving. Yet, automatically detecting and applying refactorings to improve the quality and structure of the integrated platform based on their approach would be valuable future work.

Several studies explored changes done in highly configurable software. These studies consider the variability model and the build system [Dintzner et al., 2016, Passos et al., 2015, Lotufo et al., 2010], whereas in this dissertation I only focus on the code level.

Dintzner et al. aggregate feature-evolution information by mining commits [Dintzner et al., 2016], including extensive information of what artifacts are affected. The work mainly considers commits that touch `#ifdef` blocks. The variability model, build system, and source code is used to identify which artifacts are affected and in which layers. The focus of that work is not to detect the exact type of changes, but to offer an overview of the evolution of features. Passos et al. present a catalog of patterns on the co-evolution of features in the variability model, build system, and code, obtained from the Linux kernel [Passos et al., 2015]. Several patterns use only the variability model and/or the build files to add or remove features. Some of the patterns presented in Ch. 6 overlap with the ones from Passos' patterns: P3 AddIfdefElse corresponds to AVONMF (Add Visible Optional Non Modular Feature), P5 AddIfdefWrapThen to FCUTVOF (Featurize Compilation Unit to Visible Optional Feature), and P9 RemIfdef to RVONMF (Remove Visible Optional Non Modular Feature). The main difference between the two is that I zoomed into understanding how source code changes can be realized with a variation control system, disregarding other layers like the variability model or the build system. Nevertheless, considering other layers would be interesting as feature work. For example, the variation control system could create, maintain and ensure consistency of a variability model.

8.3 Projectional Editing

Kruskal describes an editor to handle sequential versioning (deltas) over time and concurrent versions (conditional compilation) [Kruskal, 1984, Kruskal, 2000]. The conditional compilation uses Boolean expressions to wrap code, and the editor allows to

select which code to select by giving a Boolean expression. The code fragments that correspond to that Boolean expression are available to be edited. Several commands are available to the user for maximum flexibility. The main difference is that I use only a projection and an ambition to specify what does the change affect. Their study explains how the editor works, but offers no information on how easy it is to use nor the feasibility of using such a system. I can only infer that the system has been used successfully [Kruskal, 2000].

Lie et al. present an alternative versioning model that has its foundations on logical changes [Lie et al., 1989]. The change-oriented versioning (CoV) focuses on doing changes that are related from the functionality perspective, users manipulating options (similar to what we call today features) using *choice* and *ambition* for reading and respectively for writing to the repository. Munch [Munch, 1993] presents an experiment on how to use CoV on *gcc*. A tool CC2CoV (Conditional Compilation to CoV) translates the existing CC++ source code files into CoV representation in the EPOSDB-II [Gulla et al., 1991]. Conditional compilation macros are mapped to options and are interpreted as compile flags. They then automatically checkin version 2.4.0 of gcc using CoV into the system, and compare how their system fares to the repository stored in RCS [Tichy, 1985]. However, their experiment does not draw any conclusions on how feasible it is to use the system to handle the evolution of large systems.

Atkins et al. develop a *version editor* that hides preprocessor directives, allowing to edit a particular variant of a source file [Atkins et al., 2002]. Edits to the view are propagated back into the source file. Their study on a large telecommunication project shows a productivity increase of up to 40%. In comparison, we focused mainly in understanding what kind of operations should such a tool support, and if indeed, these operations can be used to maintain and evolve highly configurable software systems.

Hofer et al. argue that existing approaches to assist with handling the preprocessor are tied to IDEs, thus, their adoption rate is low [Hofer et al., 2010]. They introduce the filesystem LEVIATHAN that mounts a view representing a variant. Heuristics are used to synchronize changes in the view with the source code in the physical storage. However, it does not allow modifying the structure of the conditional blocks when working on a view.

C-CLR is an Eclipse plugin that allows creating a view by selecting the respective preprocessor macros (features) [Singh et al., 2007]. The tool offers support for generating views, but not for executing changes and updating the view. Similarly, folding is used as a visualization technique by Kullbach et al. [Kullbach and Riediger, 2001] to hide and unhide code in the GUPRO tool [Ebert et al., 1998]. The idea is to fold parts of code (including preprocessor directives) and possibly labeling the fold to easily identify its purpose. Compared to these two works, we wanted to allow modifying the view and updating the repository with the new changes.

Kästner et al. propose colors to show annotated code corresponding to a feature [Kästner et al., 2008], and implement the Colored IDE (CIDE). The tool requires

disciplined preprocessor annotations, such that arbitrary code fragments cannot be annotated. A variant view shows annotated code fragments using a background color according to a feature selection. Markers are used to show code that belongs to features that are not selected.

A similar tool that uses colors (but lacks the ability to hide code) is developed by Le et al. [Le et al., 2011]. Internally it uses the choice calculus. A controlled experiment with students shows that the prototype increases code comprehension compared to the C preprocessor tool. Users were more successful and efficient in completing their tasks and gave more correct answers, which motivates the use of dedicated variation control systems.

Janzen et al. propose to use a concept called *crosscutting effective views* to modularize concerns [Janzen and De Volder, 2004]. The *modules view* provides a decomposed structure in terms of module units of the program. A *classes view* shows the decomposed structure of classes. Changes applied to one view are reflected in the other view, which is automatically modified and updated. The tool stores the structure of the program internally, while the developer edits a so-called *virtual source file* [Chu-Carroll et al., 2002].

Westfechtel et al. propose a uniform version model [Westfechtel et al., 2001] that allows for the version model to be orthogonal to the data model. This effort has resulted in SuperMod [Schwägerl et al., 2015]. This tool is probably the closest to the variation control system. It uses the same idea of projections (called choices in SuperMod) and ambitions to support variant editing. SuperMod is implemented as an Eclipse plug-in, and supports different kind of artefacts. The only current drawback is that it requires an importer to transform current projects that use preprocessor annotations into specific projects that SuperMod can use. One key difference is that SuperMod enforces complete configurations of the system, while the variation control system presented in this dissertation allows for partial configurations.

9. Conclusion

In this dissertation I had two objectives. First, to investigate the challenges in the development of software variants (O1.), and second, to design, develop and evaluate tools that would support the development and integration of software variants (O2.).

One of the most common techniques to implement variability is using annotations. Developers annotate code to include or exclude it at compilation time. Although preprocessor annotations have been heavily criticized, they are still heavily used in practice both in industry and in open source systems.

On the other hand, forking is a cheap and readily available mechanism tailored for quick development and exploration. Despite known problems, particularly costly maintenance, many organizations still prefer to use it to develop software variants. Flexibility and no need for specialized tools or processes are the main strengths for forking.

To retain benefits of both techniques, I explored possible improvements by combining the traditional forking and preprocessor annotations. I analyzed in depth how the two techniques are combined and used by developers in an open source project, and learned that developers prefer to use one or the other as needed. To facilitate this, I developed a variation control system that unifies the two techniques, allowing for higher flexibility. Early results show that the variation control system is feasible to use on large systems, with many features.

Often variants developed through forking need to be integrated into one integrated platform. To this end, I explored how an intention-based integration tool can support the integration. The core idea is the usage of *intentions* and of *exploratory views*, allowing developers to make changes and previewing the end result of the integration, while still being able to see the original code from the two variants. The benefit of the intention-based integration is its intuitiveness and being less error prone. In our experiments, we have shown that INCLINE produces correct results, the set of intentions suffices for most integration tasks and that it scales up to files of 4k LOC. Moreover, in the controlled experiment 8 out of 12 INCLINE users did not do any mistakes in the integration tasks, compared to only 1 participant who did not do any mistakes when using Eclipse.

Future work The variation control system and the intention-based integration are alternative ways for developing and migrating variants to integrated platforms. So far, the experience with both tools is limited but early results are promising. The variation control system was developed as a command line tool. This has some limitations and makes it more difficult to use in some cases. It would be interesting to have an alternative to the command line tool, for example integrating it into a text editor. This would allow to add first class support for the checkin and checkout operations, i.e., allowing users to explore the code by adding or removing features in the projection. Currently, the variation control system does not enforce a feature model to be used and maintained. I believe that adding support for feature models would be a useful addition to the tool, and it would bring it closer to be a complete toolbox for software product line engineering.

INCLINE was developed as a standalone tool to support variant integration. There are few interesting directions to improve INCLINE. First, collaborating with a user interface designer/expert would be of great help to improve the UI and overall usability of the tool, to offer a smooth experience which will potentially lead to faster integration times. Second, many variant systems exist in C/C++ codebases. Currently, INCLINE has a very simplistic language representing the preprocessor language. It would be valuable to fully support C/C++ codebases (e.g., perhaps using mbeddr), to be able to provide compilation support for type checking the integrated results. Finally, a potential direction is to embed the variation control system within INCLINE, as an alternative to coupling it to a standalone text editor. This would offer the needed UI support for the variation control system, while allowing to use the intention-based integration of forked variants as needed.

A. Appendix

A.1 Paper A

Flexible Product Line Engineering with a Virtual Platform

Michał Antkiewicz, Wenbin Ji,
Thorsten Berger, Krzysztof Czarnecki
University of Waterloo, Canada

Thomas Schmorleiz, Ralf Lämmel
Universität Koblenz-Landau, Germany

Ştefan Stănciulescu, Andrzej Wąsowski
IT University of Copenhagen, Denmark

Ina Schaefer
Technische Universität Braunschweig, Germany

ABSTRACT

Cloning is widely used for creating new product variants. While it has low adoption costs, it often leads to maintenance problems. Long term reliance on cloning is discouraged in favor of systematic reuse offered by product line engineering (PLE) with a central platform integrating all reusable assets. Unfortunately, adopting an integrated platform requires a risky and costly migration. However, industrial experience shows that some benefits of an integrated platform can be achieved by properly managing a set of cloned variants.

In this paper, we propose an incremental and minimally invasive PLE adoption strategy called *virtual platform*. Virtual platform covers a spectrum of strategies between *ad-hoc clone and own* and PLE with a *fully-integrated platform* divided into six governance levels. Transitioning to a governance level requires some effort and it provides some incremental benefits. We discuss tradeoffs among the levels and illustrate the strategy on an example implementation.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Management

Keywords

product line engineering, clone management, virtual platform

1. INTRODUCTION

Development of multiple variants of products is often needed in order to satisfy conflicting requirements, legal frameworks, or to adapt the products to different geographical regions and usage conditions. In many cases, such product families are created using *clone-and-own*—a new variant is created by copying and customizing assets from an existing variant.

*Supported by ARTEMIS JU grant n° 295397 VARIES

ICSE Companion'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2768-8/14/05...$15.00
http://dx.doi.org/10.1145/2591062.2591126

Despite having low adoption costs and allowing independence from other developers, cloning easily leads to inconsistencies, redundancies, and lack of control. In the literature, using cloning in the longer term has been considered a harmful practice [8]. It has been traditionally recommended that organizations adopt a more systematic, strategic reuse offered by *product line engineering* (PLE) [10] based on a central platform. Such a platform should integrate the reusable assets and it should be used for deriving new variants of products. Existing incremental PLE adoption strategies [4, 6] discourage relying on cloning due to maintainability issues. However, as shown by industrial practice, eliminating cloning and adopting the integrated platform is not always desirable nor beneficial as it requires high-risk migration processes [2].

In this paper, we present an *incremental* and *minimally invasive* strategy for adoption of product-line engineering called *virtual platform*. Virtual platform allows organizations to achieve many benefits traditionally associated with having a fully-integrated platform but without requiring the high-risk transition processes, while retaining the flexibility and benefits of cloning. Most importantly, it allows organizations to obtain *incremental benefits proportional to incremental efforts* suitable to the frequency of reuse and the required degree of consistency among the variants.

2. VIRTUAL PLATFORM

To describe the spectrum of strategies employed within the virtual platform, we use the following conceptual framework. An *organization* runs many *projects* concurrently. Each project has a *team* and *assets*. The team uses the assets to derive one or more *variants* of products. An *integrated platform* is a special kind of project intended to keep reusable assets that can be used without modification by teams in other projects. The variants can be further characterized by *features*. The customer requests a variant based on the desirable features. The features can be mapped to *fragments* of assets used to specify and implement them.

The main idea of the virtual platform strategy is to apply a clone management approach to make distributed assets reusable instead of physically containing all reusable assets in an integrated platform as typical for PLE. Transitioning from clone-and-own to a fully-integrated platform as advocated in literature is difficult, as it requires transforming assets not intended for reuse into a set of fully reusable assets that features map to. Furthermore, the transition requires introducing new processes, training, and switching development focus from a single variant to the entire family of variants. Such a transition disrupts the organization's

ability to operate and continue development [14].

In reality, there are many practical intermediate points between the clone-and-own and the fully integrated platform. Whether the effort spent by an organization on preparation for reuse (either via clone management or PLE) is *justifiable* depends on the required *frequency of reuse* and the required *degree of consistency* among the reused assets. In the following, we present the governance levels of the virtual platform, discuss their tradeoffs, and illustrate them, where applicable, on the 101companies effort [3].

2.1 L0: Ad-hoc Clone-and-Own

Teams freely copy assets across projects and modify them as needed, without any reuse strategy or process. No preparation for reuse is needed. Entire projects, assets, or fragments of assets are copied. No notion of features is used and therefore no mapping of features to assets exists. A single project containing all assets is used to derive one variant.

Advantages. Cloning is associated with many benefits [2, 8]. It is easy and fast for teams knowledgeable about the project, since no special development tools or processes are needed. Developers of a new variant are also independent from the developers of the original, and free to modify it as needed. Finally, as the original variant may have been tested and used, the new variant may be usable from the beginning.

Disadvantages. If not carefully managed, cloning has serious drawbacks [2, 8]. It does not scale: with an increasing number of variants, the overhead for synchronizing assets may exceed the benefits of the initial reuse. Cloning also requires governance and discipline among developers. Without specified cloning practices and recorded provenance information, the assets used to create the original and the clone easily become disconnected and inconsistent. This can result in redundant work and can hinder long-term evolution.

Tactics. Traditional small-scale reuse tactics such as component libraries and frameworks can be used to make the assets more reusable. Also, cloning can be better managed by using branching and merging capabilities of a (distributed) version control system, which automatically records some information needed for locating features.

Example. The goal of 101companies [3] is to aggregate a set of contributions from different authors who implement the same set of features of a fictitious human resources management system, while illustrating different implementation languages and techniques. The practice within 101companies can be characterized as ad-hoc clone-and-own, except that the system is described by a feature model. Each contribution is also characterized by a set of features, but without any mapping to the assets. No libraries or frameworks are used to make the project assets reusable.

Recommendation. Ad-hoc clone-and-own is appropriate when the frequency of reuse is very low and maintaining the consistency among the projects is not important.

2.2 L1: Clone-and-Own with Provenance

Teams record provenance information about the original projects and per cloned asset. Teams use the provenance information for impact analysis and change propagation.

Advantages. Provenance information enables propagation of extensions and bug fixes among the cloned assets. During development of an original asset, teams can send notifications about changes to teams working on the clones. Conversely, teams working on a cloned asset can decide whether change

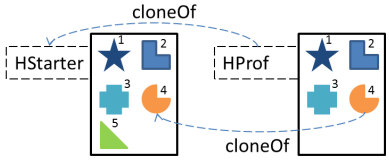


Figure 1: Recording provenance information

progagation is needed upon receiving a notification.

Disadvantages. Since the provenance information is coarse-grained (entire asset), teams need to manually locate the relevant fragments of assets and propagate the change.

Tactics. Develop small, cohesive assets. Use facilities of a version control system, such as branching, to isolate modifications into coherent groups across assets, and exchange changesets. Incorporate metadata using explicit feature annotations in code or in commit messages. Use clone detection tools to recover provenance information.

Example. In 101companies, we extended the metadata of a contribution to contain information about the original contribution it was cloned from. Fig. 1 illustrates two contributions: `HStarter` and `HProf`. Solid line boxes represent the projects. The shapes numbered 1–5 represent fragments of assets contained within the projects. Dashed lines represent the provenance links (`cloneOf`) for project `HProf = cloneOf(HStarter)` and for asset 4 `HProf::4 = cloneOf(HStarter::4)`. We can also see that asset 5 was not cloned. We analyzed the version control history to detect instances of cloning and recover provenance information. We detected that in a commit to `HProf`, a new asset 4 was added which was a clone of an existing asset 4 in project `HStarter`, that is, asset 4 was cloned from `HStarter` to `HProf`.

Recommendation. Clone-and-own with provenance is appropriate when the frequency of reuse is low and maintaining the consistency among the assets is moderately important.

2.3 L2: Clone-and-Own with Features

Features succinctly characterize the functionality of a variant from the customer's point of view. Teams declare features and map them to asset fragments that implement them. Features can be *modular* (implemented in a single asset) or *cross-cutting* (distributed across assets), or *tangled* (a single asset can contain overlapping fragments corresponding to many features). Teams propagate features among projects by cloning the corresponding asset fragments and recording provenance information. Teams leverage notifications about feature-related changes and perform change propagation.

Advantages. Features provide a functional decomposition, and allow reasoning about the co-evolution of projects and their assets in terms of features instead of physical assets. Teams benefit from a better overview of the projects in terms of user-relevant functions. Teams can make better reuse decisions and more easily propagate features across projects, as the relevant fragments of assets can be located easily.

Disadvantages. Features can have complex dependencies and interactions, challenging their reuse. Thus, teams need to rely on intricate domain and implementation knowledge.

Tactics. Use a framework for managing cloned product variants, such as Rubin et al. [13, 11, 12], which treats features as the prime reuse units. It relies on metadata about the features of variants, their location in code, their dependencies, and their origins if cloned from other projects. Thus, relevant fragments can be located. Operators and metadata as speci-

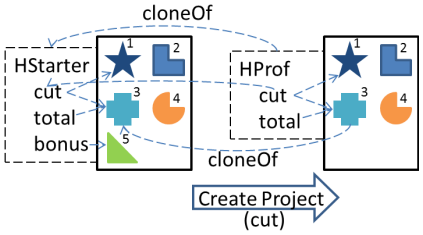


Figure 2: The scenario Create Project

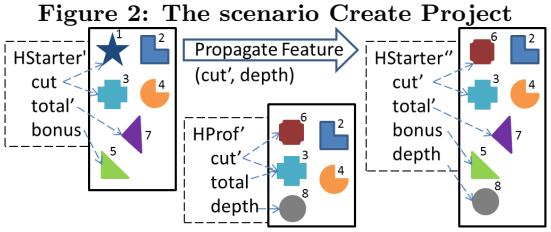


Figure 3: The scenario Propagate Feature

fied by Rubin et al. support the following scenarios in this level: refactoring to introduce features, variability and commonality analysis, propagating and sharing features among variants, retiring features, and establishing new variants [12].

Further, use white-box reuse approaches, such as those by Holmes and Walker [5]. Such approaches identify other parts of the implementation that features depend on using various techniques, including static code analysis.

Example. We applied a variation of Rubin et al.'s clone management framework to 101companies, and implemented tool support for feature location and two scenarios as follows.

The scenario *Create Project* is shown in Fig. 2 (block arrow). We represent the metadata using dashed lines: a box for the features of a project and arrows for the mapping of the features to asset fragments. For example, the project `HStarter` implements three features, `cut`, `total`, and `bonus`, which are mapped to fragments 1, 3, and 5. The other fragments, 2 and 4, are integral to the project. The mapping between features and asset fragments is computed automatically by a simple feature location algorithm. We use the links `cloneOf` to record the provenance of features.

Developers create a new project from an existing project by deselecting undesired features. In Fig. 2, a developer selected the project `HStarter` and deselected the features `total` and `bonus`. Since `cut` requires the feature `total`, both must be cloned. The developer deselected the feature `bonus` and therefore the corresponding asset was not cloned. In our approach, instead of physically removing fragments of the features that are not cloned, we comment these fragments out to compensate for the imprecision of the feature location. Also, features which cannot be located are always cloned since they cannot be commented out. Asset fragments 2 and 4 were also cloned as they are integral parts of the project. Provenance information (`cloneOf`) was also recorded.

Thereafter, developers manually inspect the assets, build the project, and uncomment the code that is still needed. For instance, in one case, parts of the implementation of one unselected feature were used in implementation of a cloned feature—these needed to be uncommented for the project to build. Finally, the developer confirmed the successful creation of a new project, while our tool recorded the set of features and their provenance information.

The scenario *Propagate Feature* is shown in Fig. 3 (block arrow). Developers first identify dependencies of a feature they want to propagate. Next, they retrieve fragments of assets related to the given feature and its dependencies. Finally, they clone the needed fragments to their project, using and recording provenance information. In Fig. 3, the team of `HStarter'` propagated a new feature `depth` and the extended feature `cut'` from `HProf'`. In the resulting project `HStarter"`, fragment 1 was replaced by the new fragment `HProf'::6`, and the fragment `HProf'::8` was added.

Recommendation. Clone-and-own with features is appropriate when the entire features are reused and the frequency of reuse is medium as well as reasoning about the features and maintaining the consistency among them is important.

2.4 L3: Clone-and-Own with Configuration

For individual projects, teams add the capability to disable features and to derive variants by selecting subsets of features. They add feature constraints to exclude invalid combinations.

Advantages. The ability to derive multiple variants from a single project reduces cloning and increases reuse potential.

Disadvantages. Focus of a developer is shifted from a single variant to a set of variants, which complicates development.

Tactics. Use a feature model [7] per project, to define features and constraints; use a configurator. Use traditional variability mechanisms, such as configuration parameters, preprocessors, generators, or component frameworks.

Recommendation. Clone-and-own with configuration is appropriate when frequent derivations of similar variants containing subsets of features are needed and when maintaining the consistency among the projects is important.

2.5 L4: Clone-and-Own with a Feature Model

An organization creates a central feature model that covers all projects and all implemented features. Teams create new projects by taking an existing project as a basis and then propagating the needed features from all other projects as allowed by feature model constraints. Teams extend the central feature model.

Advantages. The assets distributed across the projects are reusable as if they were integrated into a platform. The central feature model constrains the valid combinations of features that can be reused together.

Disadvantages. The assets are still distributed and their consistency still needs to be managed. Multiple versions of the same feature exist. Manual integration of the cloned assets is needed as they are not prepared to work with each other as there is no product line architecture.

Tactics. Merge feature models of projects into the central feature model.

Example. We envision the scenario *Create Project* as follows. Fig. 4 shows a feature model containing features from a number of contributions. The team creates a new project `HSimple` by selecting three features. The tool creates the project by cloning the fragments implementing these features.

Recommendation. Clone-and-own with a feature model is appropriate when the frequency of reuse is high, a global overview of all features and constraints among them is needed, features need to be reused from many projects, and maintaining the consistency among the cloned features is important.

2.6 L5: PLE with an Integrated Platform and Clone-and-Own

An organization creates a platform project and a platform

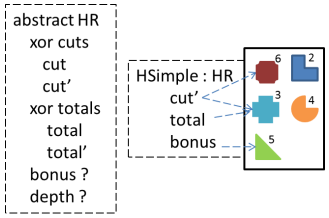


Figure 4: The scenario Create Project, cf. Fig. 3

team. This team adds configurability to the platform so that project teams can derive variants by configuring the platform. However, this team also merges existing projects into the platform and harvests features from existing projects.
Advantages. Increased scale, improved change propagation, reduced redundancy, and configuration over implementation. Although an integrated platform is created, other projects can still be kept and developers can still work on projects as cloning is still allowed. Project teams are not restricted by the platform, which supports innovation.
Disadvantages. Same as level 4, although reduced in severity as the amount of cloning is reduced and consistency is managed through the platform. Projects receive updates when adopting a new version of the platform.
Tactics. Use a clone management framework, such as Rubin et al. [13, 11, 12], to perform development of common architecture and common assets, merging initial set of cloned variants, bringing additional variants into the platform.

Use traditional annotative and compositional PLE techniques [9], which rely on a configuration mechanism (e.g., build system and preprocessor) or a suitable software architecture leveraging programming-language-level mechanisms.
Recommendation. PLE with an integrated platform and cloning is appropriate for frequent reuse, a global overview of all features and constraints among them is needed, and maintaining consistency among projects is important.

2.7 L6: PLE with a Fully-Integrated Platform

Teams only use shared assets contained in the *integrated platform* to derive variants. The platform is completely specified by a feature model: given a set of desired features, a new variant can be completely—often automatically—derived from the platform. No development happens within projects.
Advantages. Makes the sharing of assets explicit, which allows developers to reduce redundant implementation and to propagate new features, extensions, and bug fixes. The full platform integration minimizes custom code for new variants.
Disadvantages. Poses high risks. PLE adoption requires disruptive organizational changes, including a new platform team and new processes for product teams [4]. Relying on the platform for new products also hinders innovation, because the platform restricts developers' freedom, while cloning may still not be entirely prevented [2]. Beginning with a fully integrated platform approach is often not practical, as organizations cannot anticipate all future variants and features. Cost can be also very large [10]. In fact, our survey shows that only a minority of industrial product lines was adopted pro-actively [1]. Most were evolved from one variant or re-engineered from a set of cloned variants.
Tactics. Use annotative and compositional PLE approaches.
Recommendation. A fully integrated platform is hard to achieve, since new features, extensions, and bug fixes are continuously and concurrently developed within projects. Development within a platform, without a project context, is possible but difficult, since projects provide motivation, requirements, and a testing environment. Only changes worth of propagation and sharing are harvested into the platform. Thus, a platform rarely covers 100% of variants.

3. CONCLUSION

We presented an incremental and minimally invasive strategy for adoption of PLE called *virtual platform*. It combines the flexibility of *clone-and-own* with the scalability and consistency of traditional *platform-based PLE* using common variability mechanisms. We presented six governance levels as a roadmap for seamless and gradual adoption of PLE, thus eliminating costly, disruptive, and high-risk transition processes. Adopting each level provides incremental benefit.
Acknowledgements. We thank Julia Rubin for discussions about the relationship between the virtual platform and her clone management framework.

4. REFERENCES

[1] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A survey of variability modeling in industrial practice. In *VaMoS*, 2013.
[2] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An exploratory study of cloning in industrial software product lines. In *CSMR*, 2013.
[3] J.-M. Favre, R. Lämmel, T. Schmorleiz, and A. Varanovich. 101companies: a community project on software technologies and software languages. In *TOOLS*, 2012.
[4] W. A. Hetrick, C. W. Krueger, and J. G. Moore. Incremental return on incremental investment: Engenio's transition to software product line practice. In *OOPSLA*, 2006.
[5] R. Holmes and R. J. Walker. Systematizing pragmatic software reuse. *ACM Trans. Softw. Eng. Methodol.*, 21(4):20:1–20:44, Feb. 2013.
[6] H. P. Jepsen, J. G. Dall, and D. Beuche. Minimally invasive migration to software product lines. In *SPLC*, 2007.
[7] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep., SEI, CMU, 1990.
[8] C. Kapser and M. Godfrey. "cloning considered harmful" considered harmful. In *WCRE*, 2006.
[9] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE*, 2008.
[10] L. N. P. Clements. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
[11] J. Rubin and M. Chechik. A framework for managing cloned product variants. In *ICSE*, 2013.
[12] J. Rubin, K. Czarnecki, and M. Chechik. Managing cloned variants: A framework and experience. In *SPLC*, 2013.
[13] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik. Managing forked product variants. In *SPLC*, 2012.
[14] F. Stallinger, R. Neumann, R. Schossleitner, and S. Kriener. Migrating towards evolving software product lines: Challenges of an SME in a core customer-driven industrial systems engineering context. In *PLEASE*, 2011.

A.2 Paper B

Forked and Integrated Variants in an Open-Source Firmware Project

Ștefan Stănciulescu
IT University of Copenhagen, Denmark
scas@itu.dk

Sandro Schulze
Technical University Braunschweig, Germany
sanschul@tu-braunschweig.de

Andrzej Wąsowski
IT University of Copenhagen, Denmark
wasowski@itu.dk

***Abstract*—Code cloning has been reported both on small (code fragments) and large (entire projects) scale. Cloning-in-the-large, or forking, is gaining ground as a reuse mechanism thanks to availability of better tools for maintaining forked project variants, hereunder distributed version control systems and interactive source management platforms such as Github.**

We study advantages and disadvantages of forking using the case of Marlin, an open source firmware for 3D printers. We find that many problems and advantages of cloning do translate to forking. Interestingly, the Marlin community uses both forking and integrated variability management (conditional compilation) to create variants and features. Thus, studying it increases our understanding of the choice between integrated and clone-based variant management. It also allows us to observe mechanisms governing source code maturation, in particular when, why and how feature implementations are migrated from forks to the main integrated platform. We believe that this understanding will ultimately help development of tools mixing clone-based and integrated variant management, combining the advantages of both.

I. Introduction

Code *cloning* [1], [2], [3] is the practice of creating new code by copying the existing code and modifying it to match a new context. Cloning is used *in-the-small* to reuse implementations of non-trivial algorithms and to reuse local program patterns such as 'boilerplate' code seen in many framework-based systems. Moreover, cloning is frequently used for experimental changes without putting the system's stability at risk [4], [5]. Some authors suggest that, for highly-specialized complex code, cloning might even be the preferred reuse method [6].

Cloning is also used *in-the-large* to create new system variants by *forking*. In such a scenario, an entire project is copied and the copy, a *fork*, is customized to meet new requirements. The practice of forking has been observed both in industry [7] and in open source projects [8], [9], [5]. Reuse of existing tested code decreases the time and cost of delivery [4] and raises confidence in product reliability [7]. Forking, so cloning in-the-large, is a remarkably easy and fast reuse technique, and it is the subject of this study.

Both cloning in-the-small and cloning in-the-large are part of the *clone-and-own* paradigm [2], [10], [11] that is recognized as a harmful practice, credited for decreasing code quality [4] and multiplying maintenance problems [12], [13], [4], [7]. A bug found in one clone can exist in other clones, thus, it needs to be fixed multiple times [14]. Even just locating all cloned code may be nontrivial. Unintentional parallel development of the same functionality in different forks increases implementation and test costs [15]. Finally, merging diverged code forks is very laborious.

Our first objective is to understand how far the known benefits and drawbacks of cloning in-the-small apply to forking. To this end, we investigate forking practices in Marlin, an open source 3D printer firmware project. Marlin is an appropriate study subject due to an unprecedented amount of forks created in a very short period. The Marlin project has been forked 1588 times in the period of 3 years and 3 months. We investigate the following research questions using Marlin:

- RQ1 What are the main reasons for creating forks?
- RQ2 How do ongoing project development and maintenance benefit from existence of many forks? To what extent do forks retrieve changes from their origins? To what extent do forks contribute changes to their origins?
- RQ3 To what extent known drawbacks of cloning in-the-small (e.g., difficulties in propagating changes) apply to forking? Are there any new challenges?

Historically, *forking* had a negative antisocial connotation. It denoted a community schism, when a project is split and an independent development starts in a diverging direction [16]. The term has acquired a less negative meaning since the arrival of distributed version control systems and, in particular, of Github, which introduced traceability and easy propagation of commits between forks. Github makes forking relatively easy. Consequently, forking has become a potentially viable way to maintain concurrent program variants. Forking *can* now be seen as a software reuse mechanism next to established examples such as object-oriented reuse patterns, aspect-oriented programming, or software product line architectures.

The relation of forking and product line architectures has been noted in several recent works [17], [10]. The central part of a product line architecture is an *integrated platform* gathering together the core assets. The integrated platform uses programming language mechanisms, for instance conditional compilation [18], to maintain multiple variants simultaneously. Recently, we have proposed a lightweight methodology that combines the integrated platform approach with forking, referred to as a *virtual platform* [10]. The virtual platform attempts to combine the flexibility and low initial investment

of forking with acknowledged benefits of software product lines (the integrated platform). It proposes to use advanced automatic traceability mechanisms to maintain concurrent development of clones, and to allow for an easy migration of mature software fragments to an integrated platform. This way, small changes as well as experimental code could created and integrated easily.

The Marlin project is of interest in this context as it uses both conditional compilation (in the role of the integrated platform) and intensive forking (for more ad hoc volatile changes). As such, Marlin and Github can be seen as a spartan prototype of the virtual platform idea. Our second objective is to understand how the mixed variant development works in Marlin and to derive detailed requirements for developing the actual virtual platform. We formulate the following research questions:

RQ4 Under what circumstances is forking preferred over integrated variability for creating and maintaining variants?

RQ5 What are the criteria to introduce variants using conditional compilation instead of forking?

RQ6 What are the criteria that lead to integrating a forked variant into the platform using conditional compilation?

Previous studies focused mostly on analyzing cloning in-the-small, and there is little empirical evidence on what benefits and drawbacks cloning in-the-large exhibits in practice. We provide insights into forking as a reuse practice that can be beneficial for researchers and tool developers. We analyze cloning in-the-large from the variant management perspective. We provide evidence as to when and why cloning in-the-large is used for creating variants, and when it is better to have explicit variability. Our study uses an open source ecosystem, therefore the study subject can be further reused by other researchers to develop tools and methods, and run empirical studies for their tools. We provide traceability links to source material for interesting cases of cloning, merging, concurrent development, and others.

In the remainder, we provide basic definitions in Sect. II. Then, Sect. III presents the subject system, our methodology and the experiment design. Section IV surveys the results and synthesizes key observations. We then discuss threats to validity in Sect. V, related work in Sect. VI, and conclude with Sect. VII.

II. Background and Definitions

A. Git and Github

Git is a distributed version control system that allows for local repositories, which can be set to point to a remote repository. In the local repository, the user can commit his changes. When needed, changes can be pushed to the remote repository via the *git push* command. Git employs a lightweight and simple branching system, with each branch being nothing more than a label attached to a commit.

Github is a hosting service for Git repositories, offering a platform for collaborative development. Github allows for copying repositories in a structured way. This mechanism, known as *forking*, creates a traceability link between the copied repository, *the fork*, and the original project. On Github, a user can create a pull request which resembles a traditional change request. A pull request can be created either in the same repository, e.g. to allow a team to discuss the change, or from a fork to the original project. It consists of a description, possible comments from users, and a set of commits.

B. Basic Definitions

Repository. A repository is a structured storage for a project. The content is organized by a version control system.
Commit. A commit is an atomic change that was applied to a repository. It uses a similar syntax to UNIX patches, with a message attached that describes the change.
Fork. A fork is a copy of a project created by cloning in-the-large. A *formal* fork has been made using Github's forking mechanism. An *informal* fork is a copy of an existing repository created simply by copying files elsewhere, without any automatic traceability links. An *active* fork is a fork that has either synchronized with the origin after its creation, or has had changes applied to it. An *inactive* fork exhibits no activity in the repository after the creation date.
Variant. A variant is a project that was cloned and modified to satisfy certain requirements. Variants can also be created by derivation from an integrated platform, given a configuration.
Pull request. A pull request is a change request that contains commits and information about the change. A pull request can exist in one of the following three states: *open*—when the pull request is created and awaits to be verified, *merged*—the pull request is accepted into the target repository, and *closed*—the pull request has been rejected.

III. Methodology and Research Design

A. Research Questions

In Sect. I we formulated six research questions aiming at two main objectives. First, we want to understand whether forking is useful for the community, and to what extent it bears the benefits but also drawbacks of cloning-in-the-small. Second, we want to investigate the relation between integrated and explicit variant management. In particular, we want to reveal when and why conditional compilation, forking, or a combination of both is used.

B. Subject System

Marlin is a 3D printer firmware that works with ATmega microcontrollers [19]. It has been created by reusing parts of two existing firmware projects, Sprinter and Grbl, to which new code was added. The firmware computes and controls the movements of the printer, by interpreting a sliced model. Marlin must be flexible enough to deal with different hardware and printer types. It has about 140 features, which can be controlled using compile-time parameters. At the time of our data retrieval (November 2014), the main Marlin repository contained more than 1500 commits, and it has been forked 1588 times. The high number of forks and the fact that Marlin has explicit variability, makes it a good choice for our study.

The project was initiated by one Github user, ErikZalm, in August 2011. It gained attention and popularity due to several improvements over Sprinter. Over time, more than 100

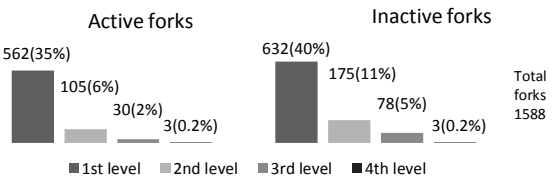


Fig. 1: Marlin's active and inactive forks, and forks' levels (percentages show the relative size in the set of all forks). The 1st level forks were created by forking main Marlin repository, the 2nd level forks were created by forking 1st level forks of Marlin, and similarly for 3rd and 4th level.

developers contributed to the project, out of which only 15 have direct commit access. Several other firmware projects are inspired by, or forked from, Marlin[1]. Besides hobbyists, 3D printer manufacturers use the Marlin firmware in their products.

Marlin is released under the GPL license. The project and its formal forks are hosted on Github[2]. In early 2015, the repository was transferred to a Github organization MarlinFirmware. In the paper we point to both, the new repository MarlinFirmware/Marlin and to the old one ErikZalm/Marlin (currently listed just as a fork of MarlinFirmware/Marlin).

C. Methodology

In our study we use mixed-methods combining qualitative and quantitative analysis. To obtain quantitative data, we built a simple tool that uses Github's public API to retrieve information about a repository, its branches, commits (and commits of each branch), issues, pull requests, forks and the owner of a repository, all as JSON files, which are then used to populate a database. The main purpose of this quantitative data is to get a comprehensive overview of the development process of Marlin and to identify points of interest that are further subject for detailed investigation.

To get insights why users fork and use conditional compilation, we classify the forks into two categories; purpose of the fork, and fork activity and nesting depth (Fig 1). We analyzed the corresponding commit messages (using key word search) of all forks using a simple heuristic in order to classify them by their main purpose (i.e., why a particular fork has been created). To obtain qualitative data, we analyze rejected pull requests to retrieve information about reasons for rejecting proposed changes from forks. In addition, we employ two short surveys directed towards active and inactive fork owners. The surveys contained both closed and open questions. They were available for ten days, after which we closed the possibility of receiving answers. We asked for the reasons to fork Marlin; what are the challenges encountered in getting their patches accepted; why they do not retrieve changes from Marlin and if they do it, how often they synchronize. We also asked about usage of conditional compilation. We distributed the survey invitation

[1]http://reprap.org/wiki/List_of_Firmware
[2]https://github.com/MarlinFirmware/Marlin

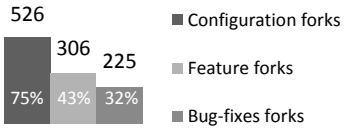


Fig. 2: Number of active forks used for configuration, developing features or bug-fixes (overlapping categories, percentages show the relative size in the set of active forks)

to 336 e-mail addresses (the other Github users did not add an e-mail address to their account), 185 belonging to owners of active forks and 151 to inactive ones. The response rate was 18.3% (34 respondents) for active fork owners, and 15.2% (23 respondents) for inactive forks. Finally, we interviewed two active maintainers of Marlin (in writing, open answers).

In the discussion, we link to exact commits by giving the Github user and the name of the repository, together with the first 8 characters of the commit hash. For example, ErikZalm/-Marlin #750f6c33, where ErikZalm is a Github user name, Marlin is the name of the repository belonging to that user and the commit hash is preceded by #. The commit hash is hyperlinked to the corresponding URL of the commit on Github, which can be conveniently followed if reading online. The above example points to https://github.com/ErikZalm/Marlin/commit/750f6c33e30ca16fab1ebe552a6b3422282bc66a. We also point to specific pull requests in the main Marlin repository, using the letter P followed by the number of the pull request, e.g. P1 points to https://github.com/MarlinFirmware/Marlin/pull/1.

Our database, survey questions, and other artifacts are available at http://bitbucket.org/modelsteam/2015-marlin.

IV. Results and Analysis

RQ1. What are the main reasons for creating forks?

We establish the purpose of forks using a term-based classification and substantiate the results with a quantitative questionnaire to fork owners. We first analyze the 700 active forks (Fig. 1) automatically. We establish the purpose of each fork analyzing commits on each branch heuristically, using the commit message. We divide the forks into the following three categories:

- *Configuration forks* are forks that change the configuration of the firmware. We detect them by checking if the main configuration files (files `configuration.h` and `configuration_adv.h`) are modified.
- *Development forks* are used to add new functionality. We detect them by matching the following search terms: *add, support, feature, new, added, implemented*.
- *Bug-fix forks* are used to fix defects. We identify them with these terms: *corrected, fix, bugfix, bug, fixed, replace*.

Commit messages of a single fork may match search terms of more than one category. In such case the fork is classified as belonging to all matching categories.

Figure 2 summarizes the results of the heuristic categorization. It indicates that 75% of the active forks have configured the

firmware. Detailed inspection shows that users of Marlin adjust the configuration to match their hardware, to enable/disable features or to fine tune parameters to ensure a good quality print. Some need to change the configuration to test a functionality that they are working on. Examples of configuration forks include: 3DPrintFIT #6343044c, jmil #04b8ef41, and Makers-Tool-Works #651b99d1. The configuration category is the most interesting. To the best of our knowledge, using version control is not a well-recognized way of handling variant configurations, while it is the dominating reason for forking in Marlin. The other two categories are expected and cover development of new functionality (43%) and bug-fixes (32%).

In order to verify the automatic, keyword-based classification, we manually analyzed a random sample of 40 active forks. For this analysis, we manually checked if changes have been done to the fork, and categorized the type of change into one of the three categories explained above. For each fork, we checked whether it has been classified correctly by comparing the manual and automatic analysis. We found that the precision of the heuristic on this sample is 97% for configuration forks, and 94% for both development and bug-fix forks, thus the data of Fig. 2 appears reliable.

Finally, we have approached the owners of the forks with a quantitative survey, asking them what were the reasons for creating their forks. Among the active fork owners, 62% (21 responses) report that they had originally intended to configure the firmware. This fraction is smaller than 75% of the actual number of configuration forks (Fig. 2). The intention to just configure the firmware was even more dominating among inactive fork owners (74% or 17 responses). The situation is the opposite for the non-configuration forks: 68% of active fork owners report that their intention was to contribute new functionality or modify the existing one. This is more than the actual number established heuristically: 54% of forks are used for new functionality or bug-fixes, which is the sum of the last two columns in Fig. 2, corrected for the intersection. Apparently, when you start with an intention to contribute new code, you cannot be certain to succeed. You may end up just configuring the code, even if you did not intend to do this.

Maintaining variant configurations in forks of entire projects is a very simple and effective mechanism. It does not require specialized configuration management or variability management tools. The fork owner has a reliable backup copy of the configuration, and the configuration can always be easily reconciled with upstream, if that becomes desirable. The Marlin community is extremely successful using this mechanism for the purpose. As mentioned above, some developers end up following this practice, even though this is not what they initially expected. It is actually somewhat surprising that this practice is not to be found in classic product line literature.

Observation 1. *Storing variant configuration data of a product family in forks of the entire project is a lightweight and effective configuration maintenance mechanism.*

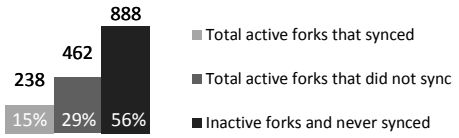


Fig. 3: Retrieval from origin (non-overlapping groups, percentages show the relative size in the set of all forks)

RQ2. How do ongoing project development and maintenance benefit from existence of many forks?

We answer this question by investigating to what extent there is a flow of contributions between the forks and main Marlin. We split the discussion into two subquestions and then synthesize.

To what extent do forks retrieve changes from their origins? To answer this question, we checked for each fork whether its branches contain any commits added to the main Marlin repository after the fork's creation date. We found that only 238 forks (15% of all forks) have synchronized at least once with the main Marlin repository (Fig. 3), which amounts to only 34% of active forks.

Observation 2. *Most forks do* not *retrieve new updates from the main Marlin repository.*

Marlin developers fork significantly more often than merge. This is a striking observation, given that merging of concurrent development strands is the key purpose of git. The forks in the Marlin ecosystem are characterized by a short maintenance lifetime (101 days on average). Once a fork achieves the desired functionality (the printer operates as expected) the incentives to maintain the fork decrease, and it becomes inactive (32% of active forks did not receive any commits between January 2014 and November 2014). Thus the period in which upstream changes are relevant for many developers is relatively short.

We confronted this hypothesis with the developers in our questionnaire. In the responses, only 18% of active fork owners synchronize monthly, and 6% synchronize weekly. Others do not synchronize at all, or synchronize irregularly. When asked, they state that the upstream changes are uninteresting for them, or that they do not wish to take in new changes as integrating them costs additional work. Merging new changes from upstream can be difficult and time consuming. One inactive fork owner explained that *I fear that my settings/calibration could change, sometimes I stay 1–3 months without changing the firmware of my printer*. This reinforces our understanding that most Marlin developers use Github to manage their variant, and not to collaborate with others.

At the same time, the *altruist* developers that want to contribute to the community synchronize more frequently. From 306 development forks, 142 have retrieved changes from the main Marlin repository. Moreover, 87% of pushed patches from development forks, come from those that synchronized. Being up-to-date with the main repository is key for producing clean up-to-date patches. This is consistent with our hypothesis that the maintenance span determines the need for synchronization.

To what extent do forks contribute changes to their origins? We found that only 202 forks (253 with forks that were deleted before our retrieval, and they are represented as unknown in pull requests by Github) contributed with patches to the main Marlin repository, so not even all feature development forks (306 in Fig. 2) have contributed pull requests upstream. Nevertheless, 714 commits have been integrated in main Marlin by merging pull requests. These 714 commits constitute 58% of all commits in the main Marlin repository, excluding the empty commits acknowledging merges. Most pull requests come from the first level forks; only seven come from the second level forks (Tbl. I). We conclude that Marlin is strongly supported and developed by the community.

An example of a contributed functionality is the *Auto Bed Leveling* (ABL) feature. A prototype of ABL was first implemented in a fork, which does not exist anymore at the time of writing. The commits have been accepted into another fork, akadamson/Marlin #728c355f (traced in our database which was populated before the original fork has been deleted). Then it was ported in alexborro/Marlin-BedAutoLev #0344dbfc to the latest version of Marlin and finally included in the main Marlin repository #253dfc4b. The feature was later improved in fsantini/solidoodle2-marlin #cc2925b7, and the improvements were accepted in the main Marlin repository #89a304fd. This example demonstrates how innovation and improvements happen thanks to collaboration of several developers that are distributed in space and time. Each of them has control over its own fork, which is the key for innovation.

Observation 3. *The ability to fork gives developers control over the code base, which encourages innovation. More than half of the commits in the main Marlin repository come from forks of Marlin.*

Let us now return to the question RQ2. The Marlin project was forked 1588 times in three years and three months (an average of 40 forks per month). As shown above, more than half of the commits in Marlin come from these forks.

Another benefit of easy forking for developers is showing up in testing and debugging. Testing 3D printer firmware is difficult, because maintainers do not have access to all the supported hardware. Hence, changes that are related to new hardware are usually tested by users having the corresponding hardware (e.g. P335, P572). During the life of the project, many users debugged problems, reported bugs, and contributed fixes developed in their own forks (e.g. P335, P594).

TABLE I: Contributions to the main Marlin repository. The last row refers to deleted forks whose level is unknown

Fork level	# forks	# pull requests (total)	# open pull requests	# merged pull requests	# closed pull requests
1	197	389	56	245	88
2	5	7	3	3	1
3 and 4	0	0	0	0	0
unknown	51	92	2	51	39

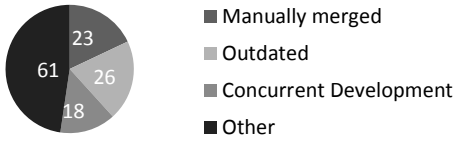


Fig. 4: Reasons for not merging pull requests. The numbers represent how many pull requests were rejected in each category. Other includes: closed by the pull request author (no reason), bad patch, pull request created on wrong repository, not fixing anything

Forking facilitates a gradual involvement of contributors. Fork owners gain experience working on their own forks. Once they gain reputation, they become committers in the main project. We have identified such cases both in our survey and in the interviews with the maintainers of the main project.

In summary, the development and maintenance of Marlin benefited from the multitude of forks in the following ways:

- Forks contribute new features and new hardware support.
- Fork owners test and improve the firmware on different hardware and configurations.
- Working on forks grooms new maintainers for the project.

RQ3. To what extent known drawbacks of cloning in-the-small (e.g., difficulties in propagating changes) apply to forking? Are there any new challenges?

We approach this question using the following methods: (i) studying the reasons for rejecting contributions, (ii) tracking how bug-fixes for important problems are propagated in the repositories and (iii) asking the fork owners about importance and challenges of receiving and contributing bug-fixes from upstream. All of these aspects can potentially reveal information about frictions in project management on the boundary of forks. We organize the discussion along the identified challenges starting with decentralization of information in forks, and moving to difficulties in propagating changes, redundant development, and other maintenance issues.

Decentralization of information. Decentralization of information is an issue that is specific to forking, where much more information is cloned than with cloning in-the-small. The modifications and extensions to Marlin are not kept in one neatly organized repository but in several hundreds of forks. For example, malx122/Marlin #69052359 added support for a second serial communication and malx122/Marlin #326c59f6 support for fast SD card transfer. These two features are not in the main Marlin repository, but they were actually taken from a fork (pipakin/Sprinter) of another firmware (kliment/Sprinter). Finding such features in the multitude of forks is extremely hard for the community members. This is consistent with an observation of Berger et al. [20] that an excessive use of clone-and-own to create variants in industrial projects leads to loss of overview of the available functionality. They name centralized information as a key advantage of integrated variability management and feature modeling over fork-based management. Also Duc et al. acknowledge that diverged code bases make it hard for

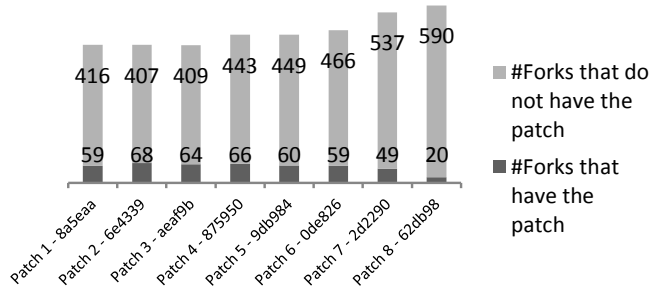


Fig. 5: Synchronization of active forks for patches. The sum of the two represents existing active forks at the time creation of that patch

individual teams to know *who is doing what*, and what features exist elsewhere [15]. This is a problem in Marlin as well, even though the individual forks do not depart far from the mainline. The sheer amount of forks makes it difficult to understand breadth of the available code.

Formal forking makes it easier to navigate the space of the available code compared to using entirely ad hoc forking. For example, the RichCatell/Marlin repository was not created using the forking mechanism of Github. This informal fork has several improvements including an updated auto-calibration feature, but it is far less known than the main Marlin repository.

Observation 4. *Decentralization of information in many forks is a challenge in fork-intensive development.*

Redundant development. The analysis of rejected pull requests showed that 18 pull requests (14% of all rejected pull requests) are rejected because of concurrent development (Fig. 4). The requested change either contained a feature or a bug-fix that already existed, or it was developed in parallel by two developers. This is a challenge not only for the contributors (e.g. P1087 or P223) but also for the maintainer who needs to have a good overview of all open pull requests to resolve the conflicts in the best possible way (e.g. P594). Berger et al. [20] confirm that concurrent development is a similar issue in industrial projects using clone-and-own.

Challenges in change propagation. The fact that forks do not retrieve changes from the main repository is problematic as fixes and new features are not propagated. In order to understand this phenomenon, we selected eight patches fixing important bugs in the main repository between January 2014 to November 2014, and verified if they were propagated to the forks. Figure 5 shows to what extent these patches have been adopted in forks. The light color part of the bar represents the number of active forks that have not pulled the patch, while the darker grey part represents active forks that have the patch (this only considers the active forks that existed before the patch was committed). For example, patch 1 in ErikZalm/Marlin #8a5eaa3c fixes a bug in a feature that may damage the printer. All the considered patch adoptions exhibit the same pattern.

Observation 5. *Propagation of bug-fixes is a problem for forking, just like for cloning in-the-small, even though git offers facilities for selective download of patches from upstream.*

At the same time, forks do not push changes back, so important fixes from the forks may never make it to the main repository. As many as 447 forks (63% of active forks) did not submit any patches to the origin. The survey data shows that one of the challenges is to prepare a robust pull request that does not break other features. A developer who works on his own fork, may find it difficult to take into account how his fix will affect configurations of all the other users. See for example a case of pull request P594 mentioned above, where a developer proposes to fix the problem for one hardware configuration, by removing the code that is necessary to make other configurations work. It is easier to maintain this general view on the variants, when integrated variability is used.

Other maintenance issues. When a project has many forks the maintenance becomes costly due to the large number of incoming change requests. An interviewed project maintainer explains that Marlin needs more developers and maintainers, as there are not enough people to support all the desired changes from the community.

RQ4. Under what circumstances is forking preferred over integrated variability for creating and maintaining variants?

In order to understand why developers prefer forking over integrated variability management, we investigated scenarios in Marlin's history that are typical of fork-based development. We selected the cases that appeared to be beneficial based on developer's statements or on our experience in variability management. This qualitative analysis included forks that are used to manage configurations and forks that develop features, but do not push changes upstream. We disregarded forks that push changes upstream as these have to integrate variation using conditional compilation. Since Marlin itself was created as a fork, we investigated its origin and the initial rationale.

Observation 6. *We have observed that Marlin developers preferred forking over integrated variability under the following reasons:*

- *S1. The fork extension has little relevance to other users.*
- *S2. The maintenance time span for the developed code is expected to be short.*
- *S3. The external developer has no control over the upstream project.*
- *S4. A developer wants to create experimental code.*
- *S5. An active project provides a good skeleton for adding new functionality.*
- *S6. A defunct project contains code that can be reused.*
- *S7. The developer wants to change the programming language.*

Next, we detail the above scenarios using concrete examples. Our data shows that there are 526 forks that did modifications to the configuration files. There are 316 forks (45% of active forks) that modified only the configuration files and made no other changes (S1). For instance, 33d/marlin-i3 #abaec3b3 configures the firmware to comply with a specific hardware. Some other forks mix configuration changes with other development changes 0xPIT/Marlin #70c7dde7, which makes it difficult to create pull requests containing just the new code and no

configuration noise. Both previous examples have not been updated afterwards (S2). Recall that 66% of active forks never synchronized with upstream after making their changes. Once the firmware is configured and running on the printer, new changes are not desired and no further maintenance is associated with these forks (S2). The average lifetime of a Marlin fork is 101 days, according to our data.

Forks commonly develop features for their own use, which may be highly experimental. For instance, martinxyz/Marlin #a8d59b1a modifies the IRQ functionality of the software and even adds an alternative IRQ code (#2a1c0766) for the stepper motor control in the firmware (non-standard IRQs are unlikely to be used). In this case, experimenting with code and adding new features that the original project lacks are the main reasons for forking, but the changes are, at least initially, not interesting to other users (S1, S4).

The fork jcrocholl/Marlin is a first level fork of main Marlin. It adds support for a new type of printer, a so called *delta*, that works differently than the normal Cartesian printers. Delta printers use spherical geometry and compute the location of the movements using trigonometric functions, such that the nozzle is not moved along the Cartesian planes. Before this extension, Marlin provided already support for some existing hardware and most of the needed software (S5), which made it easy to introduce the extension. From user's perspective this was a large qualitative leap, almost a new project though, as it supported hardware with completely different design. The main Marlin project was not affected in any way by these changes. Moreover, the developer had complete control (S3) over his fork allowing him to progress fast.

Originally, Marlin itself was created by cloning and extending parts of kliment/Sprinter and grbl/grbl. The Sprinter firmware was itself a fork of tonokip/Tonokip-Firmware, which was based on Hydra-MMM firmware. So heavy forking in this community predates Marlin's time. These earlier projects provided a good skeleton, from which Marlin could evolve into a solid standalone variant (S5, S6). There were few reasons to incorporate Marlin improvements into the upstream projects, as soon as the main developer realized that he has different goals and a different roadmap. The original projects would have never agreed to accept them anyways (S3, S4).

Another interesting example is the Traumflug/Teacup-Firmware firmware that started as a complete rewrite of the triffid/-FiveD_on_Arduino firmware. This fork decided to switch from C++ to C (S7). In such case integrating the differences between the two projects as variation points made little sense (S7). For Teacup, forking was the only possibility of development.

RQ5. What are the criteria to introduce variants using conditional compilation instead of forking?

Use of integrated platforms has been a subject to extensive studies in the field of product line engineering. The main text books of the field give criteria to select variation points and features [21], [22], [23]. Still, we ask this question in the context of Marlin, as the community uses both integrated and forked variants. The accumulated experience sheds light on the choice between the two mechanisms. We approach the question by qualitatively analyzing forks that developed changes involving preprocessor directives, excluding those that modify only the configuration files (configuration changes involve changing preprocessor directives). We supplement this data with answers to our survey and interviews.

Observation 7. *Marlin developers prefer integrated variability (conditional compilation) over forking in following situations:*

T1. The flexibility to use several variants is needed.
T2. Coding conventions expect usage of #ifdefs.
T3. Project maintainers require conditional compilation for new features submitted to main Marlin repository.

We identified 261 forks (37% of active forks) that introduced preprocessor directives in their commits. Conditional compilation is used in these forks for the same reasons as in other system level software [24]: to reduce use of memory, to disable functionality that is incompatible with current hardware, or to control inclusion of experimental code. Marlin supports printers based on 8-bit ATmega micro-controllers that have limited flash memory, usually 4–256kB. Seven out of 11 developers report use of conditional compilation for managing memory limitations (e.g., *Not all boards have enough space to run all the features so my feature was only compiled into larger chips*). In general, developers use conditional compilation to guard functionality that is optional, allowing it to be switched off either for themselves or for other users. Hence, flexibility of integrated variants (T1) is needed to meet memory requirements of different use cases and different hardware in the same fork.

Interestingly, many forks that do not contribute any changes to main Marlin use conditional compilation. Only 83 out of the 261 forks using this mechanism actually created pull requests (32%). They might have needed the flexibility for their own printers or experiments (T1). However, several survey respondents clearly state another reason—the coding conventions of the surrounding code: *That was how it was done by the guys before me, I didn't want to break the feel of the code.* (T2).

Finally, if a developer plans to contribute a feature to the main repository, he has to follow Marlin's coding guidelines. One of the main project maintainers states in the interview: *Every new feature contribution requires conditionals in some amount. New features that don't do this will be deferred until they do* (T3). Indeed, the pull request P594 discussed above, was initially rejected because it did not properly consider conditional compilation. Obviously, the Marlin maintainers enforce this rule, because the broad Marlin community needs the flexibility prescribed by criterion T1.

RQ6. What are the criteria that lead to integrating a forked variant into the platform using conditional compilation?

We define integrating a forked variant as either merging the entire fork or one of its features into the main platform. Both phenomena are of interest for researchers working on development modes mixing integrated and forked (virtual) variants [10], [25], [26], [17]. While several research groups are

attempting to build technical solutions for mixed development, we know little about how they should be used. The two previous research questions tell us under what conditions a particular mode can be selected. RQ6 asks about changing the integration mode for a variant. We answer it by analyzing history of variants integrated into the main Marlin. We learn that:

Observation 8. *Marlin fork owners consider integrating their forks into the main platform for the following reasons:*

U1. Integrating widely used variants that need to be kept in sync with upstream reduces effort and evolution cost.

U2. Integrated features are more visible and attract more users.

Observation 9. *A fork-based variant is integrated into the main Marlin platform under the following conditions:*

U3. The quality of the feature is within standards. It has been tested and is known to work as expected.

U4. Project maintainers accept to take over the maintenance, and the feature is aligned with project goals.

The *Auto Bed Leveling* feature (#253dfc4b), described above, has been created in a fork, later updated in another fork and finally integrated into the main repository. The integration took place after the feature has been tested and widely recognized[3] as well functioning (U3). On the other hand, if a feature is not working as expected and may introduce bugs or affect functionality, then it is not integrated. One such example is thinkyhead/Marlin #de725bd4, that adds support for SD card sorting functionality. This feature was not accepted in the main Marlin, because it causes problems in some specific cases (U4), but it was kept in the fork because it is demanded by users.

The fork for delta printers jcrocholl/Marlin has been integrated into the main Marlin repository (#c430906d, #6f4a6e53). Integrating the forked variant into the origin gave both visibility (U2) and lower maintenance efforts (U1). One of the maintainers stated that deltabot was merged *because it was clear and we knew it had been well-tested* (U3). Additionally, developers started to contribute changes to deltabot (P511,P568). In the deltabot fork, there were only nine pull requests created and only one got accepted. On the other hand, after deltabot was integrated into Marlin, there were created 20 pull requests related to Deltabot in ErikZalm/Marlin (55% more), out of which 14 were accepted (U2, U4). Interestingly, jcrocholl/Marlin remains a separate fork where the experimental development (S4) ahead of the main repository is happening. This allows the owner to continue development outside the control of Marlin project maintainers (S3).

V. Threats to Validity

Internal validity. We have classified commits and forks using keyword-based heuristics, and then employed this classification in a qualitative investigation of selected cases. The use of an imprecise heuristics could introduce noise. We did cross-check the precision of the heuristic on a small sample, obtaining a good precision. Thus, we believe that the reported numbers are representative. Also since the detailed analysis is qualitative, misclassification would have been detected in later phases.

Other quantitative data (contributions from forks, pull request classification, propagation of bug-fixes, etc.) have been obtained using exact algorithms and manual analysis. Care has been taken to eliminate human mistakes.

During our study the Marlin firmware continued to change. We have used an offline database obtained during a short time interval to minimize the risk of divergences in the data source. Then we used this database for quantitative queries.

It is possible that inactive fork owners synchronize directly with their local git repositories, without leaving traces in the remotes. Thus, it is hard to quantify how many actually synchronize or push changes outside of the pull request mechanism. Our survey verifies that this practice does not appear at a meaningful scale though.

External validity. Practices in other software domains might not follow those of Marlin developers. It is reassuring though that several observations are consistent with findings of independent studies of related problems. These were cited during our analysis and in the following section. Since little is known about mixed development of integrated and forked variants, any insight, even just into the firmware domain, is useful.

VI. Related Work

Cloning in-the-large. Several existing works investigate the reasons for forking [7], [15], [5], [27]. Developers who fork in industrial projects are reported to be motivated by the immediate availability of the method, freedom of control and independence from the old code base [7]. Easy access to validated code is also a frequently cited reason [7], [15], [6]. Both in open source software [27], [5] and in industry [15] forking is also motivated by project organizational matters (who has control over what code). For instance, in commercial projects, different teams have different roadmaps for their products, different risk management strategies, and different release schedules—all issues that can be circumvented by forking [15]. Open source developers may decide to fork in response to needs that are not recognized by the project's maintainers [5]. A group of developers can fork a project to start a new organization with different priorities and direction than the original one [5]. As we can see, both in closed and open projects forking centers around adding new functionality, which seems impossible in other ways [27], [5], [7], [15].

Forking comes at a cost, though. Maintenance becomes more difficult [7], [15], largely because of the lack of traceability between the forks [7]. Features and bug-fixes for different code bases are developed independently, thus duplicated code is created unintentionally [15]. A bug that occurs in one code base may exist in others [7], [14], and can remain unfixed [14]. Typically, the independent teams are unaware of changes that exist in other forks [15]. Few forks are merged back into origin and even fewer integrate code from similar or original projects [5]. Rubin et al. [28], [17] have extracted a set of fork

[3]Forum discussion: "Bed Auto Leveling.. check this out": http://forums.reprap.org/read.php?151,246132

management operators, based on three industrial cases. The operators support both managing forks and integrating them into a platform. They are meant to reduce the cost of forking.

Our study confirms the above findings in several ways. We have found several bug-fixes that are not propagated in other projects. Adding new functionality is also one of the main reasons for forking in our study, but often the intention is to share the changes. We also confirmed occurrences of redundant development. The present study differs in the following ways. First, our objective is to understand how forking functions as a reuse and variant management mechanism more holistically; not only the reasons for doing it, but also other aspects such as the extent of synchronization, or criteria for choosing forking vs integrated variability and the life cycle of variants that are eventually integrated. We observe that in Marlin many forks share their changes with the origin, unlike the prior study of Robles et al. [5] who state that very few forks are merged back and even fewer integrate code from the origin or similar forks. Second, our study subject is an open source system. It follows a typical collaborative community driven development process, without a central organization. This is very different than the industrial processes studied before [15], [7]. Perhaps more importantly, this means that all the data used in this study is available online. We have provided traceability links wherever possible. This way, researchers working on tools and methods can use our study as an index of entries to evaluation data for their results. Third, this study differs methodologically combining study of artifacts (commits, branches, forks, pull requests), interviewing developers, and surveying both active and passive users of the community.

Cloning in-the-small. Cloning in-the-small can also be beneficial [4]. For instance, developers of the Apache web server use subsystem cloning to support a large number of different platforms: 51% of code is cloned across the relevant subsystems [29]. A study of the projects hosted in the Squeaksource hosting service reveals that 14% of the methods are copied between projects [30]. Even maintenance is reported to benefit from cloning. Cloning can decrease maintenance risk for program logic, as it allows avoiding any impact on unrelated applications or modules [6]. Cloning allows to quickly implement a new functionality similar to an existing one. Cordy [6] reports that in some domains (financial software in his study) cloning is encouraged as it reduces the risk of introducing errors. Experienced programmers clone consciously with intention to reuse knowledge [31], [32]. Still, it is generally agreed that cloning in-the-small also introduces software evolution challenges. Roy and Cordy perform an in-depth analysis of clone detection, covering aspects from techniques and tooling, to advantages and disadvantages of cloning and taxonomies of clones [33]. Several questions remain open to this day, e.g., if code clones should be removed, encouraged or refactored.

Unlike all the above works, we are interested in cloning in-the-large (forking). While some drawbacks of cloning in-the-small apply to cloning in-the-large (e.g. fixing bugs in clones), there are some that only apply to forking (e.g. redundant development across projects). We provide a detailed analysis on the benefits and drawbacks of forking in an open source community. We took a high level perspective, asking process and architectural questions about variant management using forks and conditional compilation. These could not been asked in the context of cloning in-the-small, used to scaffold code rather than to create variants.

Github and its pull request development model. Only 14% of the active projects in Github use the pull request development model [34], but 79% of commercial users of Github report use of its fork and pull request workflow [35]. The most important factors that lead to acceptance of pull requests are code quality, code style alignment, and the technical suitability of the change according to a survey on 770 Github users [36]. Pull requests are often rejected due to poor quality of the code, failing tests, not adhering to project conventions styles, and newer pull requests that solve the same issue but have better quality.

We corroborate a lot of these results. For example, only 15% of Marlin's forks create pull requests. Marlin is not different than other open source projects in this respect. However, instead of analyzing the usability of Github, we seek understanding of fork-based and mixed variant management. The new knowledge is not in understanding how the pull request mechanism works. We study pull request to understand the dynamics and life-cycle of forked variants. For example, the existence of pull requests is just a proxy for generality of the variant's value proposition.

VII. Conclusion

We have investigated how forking and integrated variant management function in a lively open source community. Forks are used to manage variant configurations. Somewhat surprisingly, Github forks of Marlin are used more to create disconnected software variants, than to collaborate on software development. Most forks diverge from the mainline, possibly due to a short activity time. Forking contributes greatly to the creativity in the project, and more than half of the commits originate in forks. Still, forking poses several challenges to project participants: it leads to distribution of large amounts of information in an unorganized space, important bug-fixes are not propagated, and occasionally the same functionality is developed more than once. Finally, we have extracted criteria used by Marlin developers to decide whether a variant should be created as a fork or integrated into the main platform, and when the former should be migrated into the latter. To the best of our knowledge this is the first such list created empirically.

This study will be used to derive requirements for a tool that mixes development of individual forks and many integrated variants, and to propose a development method around it. We have also initiated a project to integrate several Marlin forks into a product line architecture, in order to create a documented and open product-line re-engineering case study.

Acknowledgments. We thank Danilo Beuche for pointing us to the Marlin project, and maintainers Bo Herrmannsen and Scott Lahteine for their valuable input on the Marlin project. This work was supported by ARTEMIS JU grant n°295397 and the Danish Agency for Science, Technology and Innovation.

References

[1] D. Faust and C. Verhoef, "Software Product Line Migration and Deployment," *Software Practice and Experience, John Wiley & Sons, Ltd*, vol. 33, pp. 933–955, 2003.

[2] T. Mende, R. Koschke, and F. Beckwermert, "An Evaluation of Code Similarity Identification for the Grow-and-Prune Model," *J. Softw. Maint. Evol.*, vol. 21, no. 2, pp. 143–169, Mar. 2009.

[3] T. Mende, F. Beckwermert, R. Koschke, and G. Meier, "Supporting the Grow-and-Prune Model in Software Product Lines Evolution Using Clone Detection," in *CSMR*, 2008.

[4] C. Kapser and M. W. Godfrey, "'Cloning Considered Harmful' Considered Harmful," in *13th Working Conference on Reverse Engineering*, 2006.

[5] G. Robles and J. M. González-Barahona, "A Comprehensive Study of Software Forks: Dates, Reasons and Outcomes," in *International Conference on Open Source Systems: Long-Term Sustainability*, 2012.

[6] J. R. Cordy, "Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation," in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, 2003.

[7] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An Exploratory Study of Cloning in Industrial Software Product Lines," in *CSMR*, 2013.

[8] N. A. Ernst, S. M. Easterbrook, and J. Mylopoulos, "Code forking in open-source software: a requirements perspective," *CoRR*, vol. abs/1004.2889, 2010.

[9] L. Nyman, T. Mikkonen, J. Lindman, and M. Fougère, "Perspectives on Code Forking and Sustainability in Open Source Software," in *International Conference on Open Source Systems: Long-Term Sustainability*, 2012.

[10] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Laemmel, S. Stănciulescu, A. Wąsowski, and I. Schaefer, "Flexible Product Line Engineering with a Virtual Platform," in *ICSE*, 2014.

[11] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants," in *ICSME*, 2014.

[12] B. S. Baker, "On Finding Duplication and Near-duplication in Large Software Systems," in *Proceedings of the Second Working Conference on Reverse Engineering*, 1995.

[13] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," in *ICSM*, 1999.

[14] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions," in *Symposium on Security and Privacy, SP*, 2012.

[15] A. N. Duc, A. Mockus, R. Hackbarth, and J. Palframan, "Forking and Coordination in Multi-platform Development: A Case Study," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2014.

[16] E. S. Raymond, "Homesteading the Noosphere," in *The Cathedral and the Bazaar*, T. O'Reilly, Ed. O'Reilly & Associates, Inc., 1999.

[17] J. Rubin, K. Czarnecki, and M. Chechik, "Managing Cloned Variants: A Framework and Experience," ser. SPLC, 2013.

[18] B. W. Kernighan, *The C Programming Language*, 2nd ed., D. M. Ritchie, Ed. Prentice Hall Professional Technical Reference, 1988.

[19] "megaAVR — 8 bit family of microcontrollers," http://www.atmel.com/products/microcontrollers/avr/megaAVR.aspx, accessed: 2015-07-01.

[20] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wąsowski, "Three Cases of Feature-Based Variability Modeling in Industry," in *MODELS*, 2014.

[21] S. Apel, D. S. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013.

[22] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[23] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[24] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, "A Study of Variability Models and Languages in the Systems Software Domain," *Trans. Software Eng.*, vol. 39, no. 12, pp. 1611–1640, 2013.

[25] T. Schmorleiz and R. Lämmel, "Annotations as maintenance tasks in similarity management," 2015, submitted for publication. 11 pages.

[26] E. Walkingshaw and K. Ostermann, "Projectional Editing of Variational Software," in *GPCE*, 2014.

[27] T. Mikkonen and L. Nyman, "To Fork or Not to Fork: Fork Motivations in SourceForge Projects," *Int. J. Open Source Softw. Process.*, vol. 3, no. 3, pp. 1–9, Jul. 2011.

[28] J. Rubin and M. Chechik, "A Framework for Managing Cloned Product Variants," ser. ICSE, 2013.

[29] C. J. Kapser and M. W. Godfrey, "Supporting the Analysis of Clones in Software Systems: Research Articles," *J. Softw. Maint. Evol.*, vol. 18, no. 2, pp. 61–82, Mar. 2006.

[30] N. Schwarz, M. Lungu, and R. Robbes, "On How Often Code Is Cloned across Repositories," in *ICSE*, 2012.

[31] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL," in *Proceedings of the International Symposium on Empirical Software Engineering*, 2004.

[32] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An Empirical Study of Code Clone Genealogies," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 187–196, Sep. 2005.

[33] C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research," *Technical Report No. 2007-54, School of Computing, Queen's University, Kingston Canada*, vol. 115, 2007.

[34] G. Gousios, M. Pinzger, and A. van Deursen, "An Exploratory Study of the Pull-based Software Development Model," in *ICSE*, 2014.

[35] E. Kalliamvakou, D. Damian, K. Blincoe, L. Singer, and D. M. German, "Open Source-Style Collaborative Development Practices in Commercial Projects Using GitHub," in *ICSE*, 2015.

[36] G. Gousios, A. Zaidman, M.-A. Storeyy, and A. van Deursen, "Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective," in *ICSE*, 2015.

A.3 Paper C

Concepts, Operations, and Feasibility of a Projection-Based Variation Control System

Ștefan Stănciulescu
IT University of Copenhagen
Denmark
scas@itu.dk

Thorsten Berger
Chalmers | University of Gothenburg
Sweden
thorsten.berger@chalmers.se

Eric Walkingshaw
Oregon State University
USA
walkiner@oregonstate.edu

Andrzej Wąsowski
IT University of Copenhagen
Denmark
wasowski@itu.dk

***Abstract*—Highly configurable software often uses preprocessor annotations to handle variability. However, understanding, maintaining, and evolving code with such annotations is difficult, mainly because a developer has to work with all variants at a time. Dedicated methods and tools that allow working on a subset of all variants could ease the engineering of highly configurable software. We investigate the potential of one kind of such tools: projection-based variation control systems. For such systems we aim to understand: (i) what end-user operations they need to support, and (ii) whether they can realize the actual evolution of real-world, highly configurable software. We conduct an experiment that investigates variability-related evolution patterns and that evaluates the feasibility of a projection-based variation control system by replaying parts of the history of a highly configurable real-world 3D printer firmware project. Among others, we show that the prototype variation control system does indeed support the evolution of a highly configurable system and that in general, it does not degrade the code.**

I. Introduction

Tailoring systems to the specific needs of users, such as hardware environments, runtime platforms or various combinations of features, is becoming increasingly important. Such systems are typically highly configurable by containing massive amounts of variability, which is often realized using static variability annotations, such as conditional compilation directives (e.g., `#ifdef`) in C code [19]. While such annotations are among the most frequently used and most simple variability mechanisms, their use is known to complicate writing, maintaining (e.g., bug-fixing), and evolving (e.g., adding a cross-cutting feature) source code [12]. Variability annotations obscure the structure and flow of the underlying code [32], since much of the conditionally included code is often irrelevant for a particular code-editing task. In fact, working on all possible variants of the system at once is known to negatively impact the comprehension of source code [26]. Beyond syntax highlighting and code folding, no major IDE supports editing variant subsets while ensuring the consistency of the whole system.

Consider the code excerpt in Listing 1 taken from the Marlin 3D printer firmware, the subject of our study. The code represents several variants related to the printer display. Understanding the code and what impact a change might have is difficult due to the many variability annotations and the variant-specific code. Ideally, when editing one or more features, developers would only see the relevant code without being distracted by code that belongs to irrelevant features. For instance, for editing Marlin's feature `MAKRPANEL`, a developer could choose to select variants that include this feature, in order to obtain a simplified view similar to the one shown in Listing 2. Now, the developer could edit this view, and should then be able to consistently update the underlying code from Listing 1, which contains all the variants.

```
// LCD selection
#ifdef U8GLIB_ST7920
  //U8GLIB_ST7920_128X64_RRD u8g(0,0,0);
  U8GLIB_ST7920_128X64_RRD u8g(0);
#elif defined(MAKRPANEL)
  // The MaKrPanel display,
  // ST7565 controller as well
  U8GLIB_NHD_C12864 u8g(DOGLCD_CS, DOGLCD_A0);
#elif defined(VIKI2) || defined(miniVIKI)
  // Mini Viki and Viki 2.0 LCD,
  // ST7565 controller as well
  U8GLIB_NHD_C12864 u8g(DOGLCD_CS, DOGLCD_A0);
#elif defined(ELB_FULL_GRAPHIC_CONTROLLER)
  // Based on the Adafruit ST7565
  // (http://www.adafruit.com/products/250)
  U8GLIB_LM6059 u8g(DOGLCD_CS, DOGLCD_A0);
#else
  // for regular DOGM128 display with HW-SPI
  // HW-SPI Com: CS, A0
  U8GLIB_DOGM128 u8g(DOGLCD_CS, DOGLCD_A0);
#endif
```

Listing 1. Marlin excerpt (dogm_lcd_implementation.h at commit a83bf18)

Various methods and tools that allow working on dedicated subsets of all variants have been proposed in the literature [38], [23], [3], [20], [18], [29]. We refer to them as variation control systems. To some degree, they can ease the engineering of highly configurable software by providing views that only show the code related to specific variants, while hiding irrelevant code. However, none of them has found widespread adoption. In fact, no empirical data is available that shows *how* exactly they can be used to engineer real-world systems, and what their specific benefits and challenges are. Another kind of tools that provide views on programs, but that are actually adopted in practice [35], are projectional editors, such as Jetbrains' Meta Programming System [1] or Intentional's Domain Workbench [30], [5]. Unfortunately, they lack dedicated operations

```
// The MaKrPanel display,
// ST7565 controller as well
  U8GLIB_NHD_C12864 u8g(DOGLCD_CS, DOGLCD_A0);
```

Listing 2. Code relevant for editing the feature MAKRPANEL

978-1-5090-3806-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSME.2016.88

related to editing source code with variability.

Towards building efficient, projection-based methods and tools for engineering highly configurable software, we need to understand what operations users need and how they could deal with such operations. In this paper, we conduct an empirical study to assess the feasibility and actual use of a variation control system. Specifically, we realize a variation control system prototype and use it to evolve parts of the history of a highly configurable software. Our prototype supports the following workflow: (1) *checkout* a view (a version of the full source code with less variability) according to a given projection condition, (2) *edit* the view, and (3) *checkin* the edited code (updating the underlying, fully variational code).

Our system combines and extends concepts from prior work [38], [24]. We describe it using the choice calculus [10], [36], [37], a concise and formal notation that avoids dealing with intricate C preprocessor semantics and allows reasoning about highly configurable systems.

To identify end-user edit operations that the system should support, we analyze the history of a highly configurable open-source software system: the 3D printer firmware Marlin. We aim at understanding the kinds of changes (patterns) related to variability done by developers. We cross-validate the patterns using Busybox, an open-source project implementing shell tools for embedded systems. Based on the patterns we show how the edit operations for the variation control system can be implemented. We then conduct an experiment to study how the edit operations can be realized using the variation control system. In the experiment, we use our prototype to replay parts of Marlin's history, by applying randomly selected patches.

In summary, we contribute:

- A projection-based variation control system prototype relying on a checkout/checkin workflow and automatically handling variability annotations.
- End-user edit operations that show the use of the variation control system, based on identified variability-related code evolution patterns.
- Empirical data (metrics) that shows feasibility, benefits, and challenges of using the system, specifically showing that the resulting code is not significantly degraded by using the variation control system.
- A replication package in an online appendix,[1] showing the use of the system (i.e., the code before projection, the projection and its resulting view, the code changes, and the code after checking the edited view back in).

II. Motivation

We briefly introduce three previous variation control systems [20], [27], [38] that provide editable views based on projections specified by developers. All rely on variability realized using annotations embedded in code, similar to C's conditional-compilation directives (e.g., `#if` or `#ifdef`). These annotations carry a Boolean expression over *features*, called *presence condition (PC)* in the remainder. The first two systems do not exist anymore and there is hardly any evidence on their usefulness. The third one has not been empirically evaluated and no publicly available tool exists.

Kruskal [20], [21] presents an editor that realizes both concurrent versioning (variability) and sequential versioning (evolution in time) relying on variability annotations. Similar to conditional compilation, code lines are mapped to Boolean PCs, representing both the variant and the version to which the lines belong to. The editor creates views based on a partial configuration (a conjunction of features) called *mask*, supporting workflows where developers start with a relatively broad mask (e.g., projecting on just one feature), potentially restricting the mask by conjoining other features (e.g., "push" more masks on a stack), editing code in the views belonging to the more restricted masks, and then returning to more broad masks (e.g., "pop" masks from a stack). Code lines with PCs that do not contradict the mask are visible for editing. Several convenience commands are available to the user for iterating through variants and for manipulating PCs. The editor is not available anymore, and no empirical data on its use exists.

Lie et al. [24], Munch [27], and Westfechtel et al. [39] present and evaluate an alternative versioning model based on logical changes: change oriented versioning (CoV). It also unifies concurrent and sequential versioning, by attaching Boolean PCs to file fragments. It follows a classical checkout/checkin cycle, where a configuration (conjunction of features) determines both the version and the variant available in the view (e.g., the workspace), which can be edited. It decouples the projection (called "choice") from the expression used to checkin the edited view (called "ambition") to denote to which variants a change applies to. In an empirical study the authors translate existing C/C++ source code files of *gcc* into CoV representation in a database (EPOSDB-II [14]), and compare the performance against RCS [33] when doing a full checkin of version 2.4.0 of *gcc*. The experiment investigates only performance. It does not show how feasible it is to actually engineer a real-world system, and how exactly the checkout/checkin cycles using choices and ambitions can be used by developers.

Walkingshaw et al. [38] present a model for a variation control system called projectional editing.[2] They present a formal specification of the model with the get (create a view using a projection) and put (update the underlying program with changes done within the view) functions at its core. Examples are provided that show how to create the view and how an update executes the changes done to the view. However, in contrast to CoV, the definition of put is founded on an *edit isolation principle* that ensures that the only variants that change in the underlying program are those that can be reached from the view. In other words, when we use put to perform the update, the edits made on the view are guaranteed not to affect code that was hidden by the get function. Given this limitation, and since it was not evaluated on a real-world system, it is not clear whether this model can handle the

[1] http://bitbucket.org/modelsteam/2016-vcs-marlin

[2] Not to be confused with projectional editing [35], [4] used in the Meta Programming System [1] or the Intentional Domain Workbench [5]

$$
\begin{array}{rcll}
B & ::= & true \mid false & \\
F & ::= & B \mid \neg F \mid F \vee F \mid F \wedge F & \\
e & ::= & F\langle e, e\rangle & \textit{Choice} \\
 & \mid & e \cdot e & \textit{Append} \\
 & \mid & a & \textit{Token} \\
 & \mid & \iota & \textit{Identity}
\end{array}
$$

Fig. 1. Choice calculus syntax

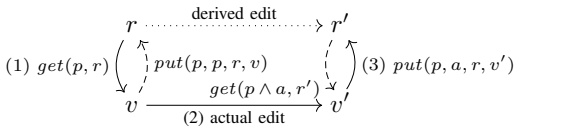


Fig. 2. Projection-based variational editing workflow and relationships. Symbol r represents the repository that contains source code, p is the projection that specifies how to obtain the view v from r using the get function. The ambition a specifies how should the changes from the edited view v' be applied to the repository using the put function. Both p and a are Boolean expressions over features.

engineering of real-world, highly configurable systems.

To study the concepts and the feasibility of using a variation control system based on projections, we create one that takes concepts from previous work.

III. Variation Control System Prototype

Our variation control system prototype can be seen as a generalization of the one described before [38]. We avoid some limitations by allowing partial configurations and using the concept of *ambition* from CoV [27], which specifies what variants are affected by the change when updating the code.

A. Choice Calculus

We use the choice calculus [10], [36], [37], a formal and concise notation for variational software, to describe our projection-based variation control system and for expressing examples in a language independent manner. Fig. 1 describes its syntax.

Unlike previous applications of the choice calculus [11], we do not embed choices within abstract syntax trees. Instead, we use a generic monoid structure. This better models the use of `#ifdef` directives in existing code repositories, since `#ifdef` directives are line-based and do not need to respect the syntactic structure of the underlying object language.

The metavariable e denotes a choice calculus expression (i.e., code). An expression can consist of a choice, the concatenation of one expression to another using the monoid append operation ($\cdot$), an arbitrary token, or the monoid identity element (ι). A choice represents a variation point in-place as a *choice between alternatives*, written $F\langle e_1, e_2\rangle$. The associated condition F is the choice's PC. When configuring a choice calculus expression, each feature is set to $true$ or $false$, then each choice is replaced by either alternative e_1 if F evaluates to $true$, or alternative e_2 otherwise. For example, given the choice $(A \wedge B)\langle 2, 3\rangle$, if both features A and B are set to $true$, the choice will resolve to `2` during configuration, but will resolve to `3` if A or B is $false$. We consider tokens to be arbitrary strings, the append operation to be string and line concatenation, and the identity element to be the empty string. This allows for a finer representation of variability than afforded by `#ifdef`, as it is not constrained to varying lines.

B. Workflow

Fig. 2 shows the intended workflow of our system. Symbols r and r' refer to the highly configurable software source code stored in a repository, and v and v' refer to working copies of the source code that are viewable and editable by the developer. The source code (r, r', v, v') is represented by a choice calculus expression (e in Fig. 1). The operation get is used to checkout a particular working copy from the repository, and put is used to checkin any changes back to the repository. Our workflow is independent of the type of storage used to contain the source code (e.g., version control systems or just folders).

In step (1) the developer obtains a simplified view v from the initial repository r. The parameter p, the *projection*, defines how v is obtained from r. More specifically, p describes a partial configuration of r, eliminating all of the variability that is irrelevant to the current code-editing task. In step (2) the developer edits v into v' using whatever standard tools they prefer. In step (3) an additional parameter a, the *ambition*, is introduced, which specifies how the developer's changes should be integrated into the repository. Note that the put operation takes into account the initial repository, the updated working copy, the projection, and the ambition when producing the updated repository r'.

The two dashed edges in the diagram describe some basic consistency principles that get and put should satisfy. These are derived from the *lens laws* developed in research on bidirectional transformations [13] and constrain the potential definitions of get and put.

The left dashed line requires that a get followed by a put is idempotent. Specifically, if we retrieve a simplified version v with some projection p and then immediately check it back in with the same ambition $a = p$, then the repository should remain unchanged. This enforces that the get operation is not effectful from the perspective of the repository.

The right dashed edge requires that a put followed by a get (with an appropriately structured projection) is idempotent. Specifically, immediately after applying the put function to checkin changes from the edited view v', we can obtain that same working copy by doing a checkout using the conjunction of p and a as the projection. This enforces that the put operation is always reversible.

C. Get and Put Function

Specifying and implementing get is straightforward (e.g., using full or partial preprocessing). We chose partial preprocessing to allow projections that are partial configurations. The function get obtains a view using the following process. It iterates through all top-level choices in the AST. It takes the right alternative if the choice's PC contradicts the projection. It takes the left alternative if the negated PC contradicts the projection. Contradictions are checked using a SAT solver [9]. If neither the PC nor its negation contradict the projection, get keeps the choice as it is. For each not eliminated alternative, get repeats this process descending into each alternative's sub-tree.

FACTORING
$$F\langle e_1, e_2\rangle \cdot F\langle e_3, e_4\rangle \rightsquigarrow F\langle e_1 \cdot e_3, e_2 \cdot e_4\rangle$$
$$F\langle e_1, e_2\rangle \cdot \neg F\langle e_3, e_4\rangle \rightsquigarrow F\langle e_1 \cdot e_4, e_2 \cdot e_3\rangle$$

CHOICE-IDEMPOTENCY
$$F\langle e, e\rangle \rightsquigarrow e$$

CHOICE-DOMINATION
$$\frac{\lfloor e_l \rfloor_F = e_l' \qquad \lfloor e_r \rfloor_{\neg F} = e_r'}{F\langle e_l, e_r\rangle \rightsquigarrow F\langle e_l', e_r'\rangle}$$

JOIN-OR
$$F\langle e_l, F'\langle e_l, e_r\rangle\rangle \rightsquigarrow (F \vee F')\langle e_l, e_r\rangle$$

JOIN-AND
$$F\langle F'\langle e_l, e_r\rangle, e_r\rangle \rightsquigarrow (F \wedge F')\langle e_l, e_r\rangle$$

JOIN-OR-NOT
$$F\langle e_l, F'\langle e_r, e_l\rangle\rangle \rightsquigarrow (F \vee \neg F')\langle e_l, e_r\rangle$$

JOIN-AND-NOT
$$F\langle F'\langle e_r, e_l\rangle, e_r\rangle \rightsquigarrow (F \wedge \neg F')\langle e_l, e_r\rangle$$

Fig. 3. Choice calculus minimization rules

For the put operation it is difficult to identify a simple and rational definition that is consistent with the above requirements by analyzing examples alone. Recall that our previous work [38] relied on an *edit isolation principle*: when doing a put in (3), the edits made in (2) cannot affect code hidden by get in (1). Although this principle is somewhat restrictive, it leads naturally to a definition that satisfies the requirements derived from the lens laws. Generalizing the edit isolation principle, we can obtain a put operation that is less restrictive than in the previous work, while still retaining the properties.

This generalized edit isolation principle can be defined as follows. Let $\mathcal{C}$ be the set of all configurations of r and r', and $r' = put(p, a, r, v')$ as defined in Fig. 2. The get function obtains all choices whose PC does not contradict the projection:

$$\forall c \in \mathcal{C}.\, get(c, r') = \begin{cases} get(c, v') & \text{if } \mathit{SAT}(c \wedge p) \\ get(c, r) & \text{otherwise} \end{cases}$$

The put update function consists of constructing a new choice with the updated view v' in the left branch, and the original source r in the right branch:

$$put(p, a, r, v') = minimize(F\langle v', r\rangle)$$
$$F = (p \wedge a)$$

We minimize the choice expressions to a more compact representation, which reduces redundancy, using the rules shown in Fig. 3. Note that they can change the syntax of a choice, but preserve its semantics. For an in-depth description of choice-idempotency, choice-domination, and the join rules, please refer to previous work [38]. In addition, we introduce the two `FACTORING` rules, which join two consecutive choices that have compatible PCs into a single choice.

D. Implementation

We implement the variation control system prototype, comprising parser, get and put function, minimization rules, and pretty printer, in Scala. The prototype is programming-language independent (line-based), but the parser and pretty printer recognize and write variability annotations in C preprocessor syntax (e.g., `#if`, `#ifdef`, `#endif`). Internally, the choice calculus [36] is used.

IV. STUDY DESIGN

In our study we investigate two research questions:

RQ1: What edit operations should a variation control system support? We analyze the history of an open-source project repository at the source code level to identify variability-related editing patterns. We show how edit operations for the variation control system can realize the patterns found.

RQ2: Can a variation control system be used to maintain and evolve a highly configurable system? In an experiment we replay parts of the history of our subject system using the variation control system prototype, and check how many cases we can support and which are not trivial to support. Using metrics we study characteristics of the code before checkout, of the view itself, and of the code after checkin. Specifically, we check that there is no negative effect (e.g., deterioration) on the source code by our variation control system prototype.

Our investigation of both research questions is followed by a discussion of the challenges we encountered (e.g., choosing a projection and ambition) during the experiment, and of the edit operations we identified.

Subject System

We study *Marlin*, a highly configurable firmware for 3D printers written in C++, which uses conditional compilation to implement variability. Marlin emerged in 2011 as a mixture of the existing projects Grbl (firmware for CNC machines) and Sprinter (firmware for 3D printers), and original code. We choose Marlin as it is large and complex enough for our purpose (40 KLOC and over 140 features) and we have prior experience with understanding the system [34], which reduces the chance of misinterpretations or mistakes. We use the development branch of Marlin's Git repository[3] on Github. We clone the "MarlinDev" repository with the HEAD pointing to commit 3cfe1dce1. This version of Marlin consists of 187 source files (excluding additional library files supporting Arduino boards).

RQ1: Identification of Edit Operations

To identify edit operations we analyze patches and extract variability-related edit patterns. We retrieve all 3747 commits (without merge commits) from Marlin's history and split each commit into a patch per changed file, excluding those files that were added, removed or renamed, resulting in 5640 patches.

We classify the patches into patterns in three steps. First, we randomly extract 50 commits that add or remove `#ifdef` directives in code using *grep*, and manually inspect the patch to understand the change and recognize patterns. Second, to automatically classify to which edit pattern a patch belongs to, we create several regular expressions to represent each pattern defined previously, and apply them on the pool of patches. We analyze the results of this step by verifying whether all the patches have been classified. Third, for patches that remain unclassified, we add the respective regular expressions and re-run the classification. We repeat these steps until each patch

[3]https://github.com/MarlinFirmware/MarlinDev

is classified by at least one pattern (note that a patch can belong to multiple patterns).

To cross-validate the patterns we run our classifier on the Busybox project, a larger project with 175 KLOC. We use the project's Git repository[4] at commit a83e3ae, containing 13,700 commits excluding merge commits, and again split them into a patch per file, which yields 34,018 patches.

RQ2: Replaying a Sample of Marlin's History

We replay randomly selected patches from Marlin (RQ2). We filter the 5640 Marlin patches down to 2322 by considering only those patches that modify files containing only Boolean PCs. This is justified as we are not interested in analyzing the complexity of Marlin's PCs. Other kinds of PCs could be handled using an SMT or CSP solver, without affecting the variation control system's main design features. From the 2322 patches we randomly select three for each identified edit pattern. Some patterns did not have any purely Boolean representative in the selection. For these we randomly pick missing patches from the whole pool of 5640 (Boolean and non-Boolean) patches, and transform non-Boolean expressions into Boolean ones by introducing new variables for non-Boolean sub-expressions (a simple form of predicate abstraction).

1) Experiment Setup: For each randomly selected patch, we manually conceive the projection and ambition required to replay it. Recall that the projection represents a set of configurations for which the code is changed. To conceive it we localize the change in the file using line numbers from the patch's meta-data and then check if those lines exist under a PC. The projection is then a conjunction of all these PCs. The ambition is conceived using only the patch information. We replay each change in three steps, where S1 and S3 are done by our prototype, and S2 is done manually in a text editor:

S1 Checkout the original source code file using the projection,
S2 Edit the view to apply changes from the patch,
S3 Checkin changes using the ambition.

2) Metrics: We compute metrics for the stages in our workflow (cf. Fig. 2): original source code (r), view on source code (v), and updated source code by our system (r'). To compare the latter to the updated original source code, we also compute the metrics for the original update (r') from Marlin's Git repository. We use the following metrics:

- LOC: lines of code in a file, including comments but excluding blank lines.
- NVAR: number of variation points; more precisely: choices (represented by `#if`, `#ifdef`, `#ifndef`, etc.) in a file. A high NVAR challenges code comprehension.

We compute the *reduction factor* for LOC and NVAR by dividing their values after checkout (view) to the values before checkout, which would show any positive or negative impact of creating views. Finally, we consider the number of checkout/checkin cycles per executed change. In some cases, we need to apply different projections and ambitions to realize a change. We record this number of steps.

[4]https://git.busybox.net/busybox

TABLE I
CODE-ADDING PATTERNS

Name	#Multi	#Only	Example
P1 AddIfdef	969	129	$\iota \to F\langle e, \iota\rangle$
P2 AddIfdef*	424	32	$(\iota \to F\langle e, \iota\rangle)^*$
P3 AddIfdefElse	271	4	$\iota \to F\langle e_1, e_2\rangle$
P4 AddIfdefWrapElse	43	17	$e_2 \to F\langle e_1, e_2\rangle$
P5 AddIfdefWrapThen	13	3	$e_1 \to F\langle e_1, e_2\rangle$
P6 AddNormalCode	4683	871	$\iota \to e$
P7 AddAnnotation	293	12	*not applicable*

V. IDENTIFIED EDIT OPERATIONS

We now explain each identified pattern and how it can be turned into an edit operation (RQ1) on top of the variation control system. We use a stripped notation of the *unified diff* program to represent the changes. A plus (+) in front of a line indicates that the line is to be added, a minus (-) that the line is to be removed. A line without plus or minus remains unmodified. We remove meta information (e.g., header, hunks, range information), as it is not particularly useful for representing the pattern. Running the pattern classifier, we identify 14 types of edit patterns. We split these into three categories: Code-Adding Patterns, Code-Removing Patterns, and Other Patterns.

These patterns represent edit operations that projection-based variation control systems need to support. We show for each pattern how the workflow described in Sec. III-B can be applied to realize the particular edit.

A. Code-Adding Patterns

Table 1 shows patterns where new code is added, together with the number of patches belonging to each pattern. The #Multi column indicates the number of patches that match the given pattern and also one or more other patterns, while the #Only column indicates the number of patches that match only that pattern. The last column provides a brief illustration of the pattern using the choice calculus.

P1 AddIfdef. In this pattern a simple `#ifdef` with no `#else` branch is added in the code, as shown below.

```
+ #ifdef ULTRA_LCD
+  lcd_setalertstatuspgm(lcd_msg);
+ #endif
```

This pattern can be achieved in the variation control system by doing a checkout with the trivial projection $true$, adding the second line, then checkin changes with the ambition `ULTRA_LCD`.

In Fig. 4 we illustrate this workflow using the choice calculus. For simplicity of presentation, we assume starting with an empty file or with just a few lines of code, in this and all of the following examples. We use U as shorthand for the `ULTRA_LCD` feature and `lcd` for the code on the second line. We will use these abbreviations throughout the section.

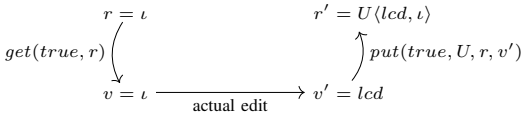


Fig. 4. P1 AddIfdef editing workflow.

*P2 AddIfdef**. In this pattern two or more simple `#ifdef` with no `#else` branches are added to the code. We distinguish this pattern from the previous one, since adding multiple `#ifdef` blocks at once may require multiple checkout/checkin sequences if the PCs are different. If multiple `#ifdef` blocks are added that have the same PC, then the edit can be executed in the same way as the P1 AddIfdef pattern.

P3 AddIfdefElse. In this pattern, presented below, an `#ifdef` with an `#else` branch is added in the code.

```
+ #ifdef ULTRA_LCD
+  lcd_setalertstatuspgm(lcd_msg);
+ #else
+  alertstatuspgm(msg);
+ #endif
```

This pattern is supported by the variation control system in two ways. The first is to do a checkout with the trivial projection $true$, add the full `#ifdef`–`#else`–`#endif` block directly, and then checkin with the trivial ambition $true$. However, it is also supported by a sequence of two edits, one that edits the configurations where the `ULTRA_LCD` feature is enabled, and one that edits the configurations where it is disabled.

Fig. 5 illustrates this edit using our workflow. We use `lcd` and `alert` as shorthand for the code on lines 2 and 4, in the pattern above.

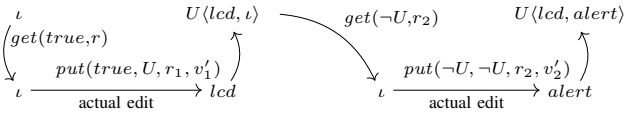


Fig. 5. P3 AddIfdefElse editing workflow.

P4 AddIfdefWrapElse. This pattern represents cases where some existing code becomes the `#else` branch of a new `#ifdef` block. The pattern is presented below.

```
+ #ifdef ULTRA_LCD
+  lcd_setalertstatuspgm(lcd_msg);
+ #else
   alertstatuspgm(msg);
+ #endif
```

This pattern is well supported by our workflow, as illustrated in Fig. 6, where we checkout with a trivial projection, edit the original code, and then checkin with the ambition `ULTRA_LCD`.

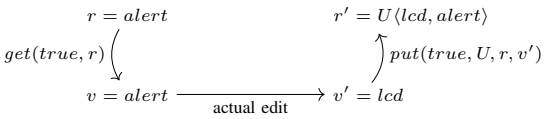


Fig. 6. P4 AddIfdefWrapElse editing workflow.

Note that a checkout with projection U would yield the same result, but the advantage of this workflow is that we can decide after making the edits how they are be applied to the repository.

P5 AddIfdefWrapThen. This pattern is dual to the previous pattern. In this pattern, the original code becomes the then-branch of a new `#ifdef` block, as illustrated in the code below:

TABLE II
CODE-REMOVING PATTERNS

Name	#Multi	#Only	Example
P8 RemNormalCode	3932	209	$e \to \iota$
P9 RemIfdef	534	24	$F\langle e_1, e_2\rangle \to \iota$
P10 RemAnnotation	228	2	

```
+ #ifdef ULTRA_LCD
   lcd_setalertstatuspgm(lcd_msg);
+ #else
+  alertstatuspgm(msg);
+ #endif
```

The workflow to support this pattern is as the one from P4 AddIfdefWrapElse, except that we checkin with ambition $\neg U$.

P6 AddNormalCode. This pattern represents changes that do not affect the variability of the code base. That is, the modified code is either (1) non-variational or (2) exists under a specific PC. This is the most common of the operations performed during system evolution [25]. In case (1), we just checkout and checkin with $p = a = true$. In case (2), we checkout with a p equal to the PC of the modified code, then checkin with $a = true$. In case (2), if the preprocessing eliminates a significant amount of surrounding code, then we expect our editing workflow to convey significant usability benefits since this code is irrelevant to the edit being performed. This is confirmed in a case in our experiment, where more than half of the code is eliminated in the view.

P7 AddAnnotation. This pattern captures cases where *preprocessor annotations* are added to the code. This usually corresponds to adding a new `#ifdef` line or a new `#endif` line to the code to fix a previous mistake. We do not exclude whitespace changes, thus, this pattern happens also when there are changes in the line that contains the annotations (e.g., adding a comment to the line). Our editing model does not support such cases since we permit only well-formed variability annotations.

B. Code-Removing Patterns

Table II lists patterns that relate to removing code.

P8 RemNormalCode. This pattern captures cases where code is removed, regardless of whether it is under a PC or not. The pattern is presented below:

```
#ifdef ULTRA_LCD
- lcd_setalertstatuspgm(lcd_msg);
 alertstatuspgm(msg);
#endif
```

The update in this case is simply to checkout the source code with projection `ULTRA_LCD`, delete line 2, and checkin with ambition `ULTRA_LCD` as shown in Fig. 7.

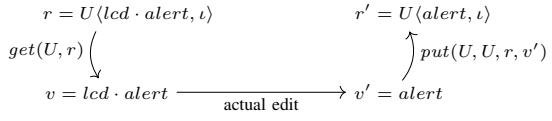


Fig. 7. P8 RemNormalCode editing workflow.

For cases where an `#else` branch exists in the `#ifdef` block, the workflow is the same, but the projection is the negation

of the PC. All the numbers corresponding to P9 in Table II include any removed code from an `#ifdef` block.

P9 RemIfdef. This pattern captures cases where code blocks guarded by PCs are removed. This pattern covers the removal of both simple `#ifdef` blocks and those containing an `#else` branch. The pattern is presented below:

```
- #ifdef ULTRA_LCD
-  lcd_setalertstatuspgm(lcd_msg);
- #else
-  alertstatuspgm(msg);
- #endif
```

This edit is dual to the P3 AddIfdefElse and can be similarly supported either by a trivial projection or by a sequence of two edits, as illustrated in Fig. 8.

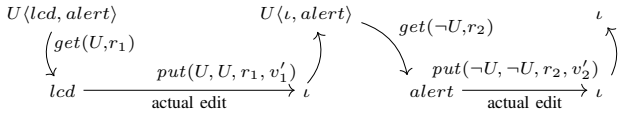


Fig. 8. P9 RemIfdef editing workflow.

P10 RemAnnotation. This pattern represents cases where annotations are removed from code. This can happens when an `#ifdef` or `#endif` line was inconsistently removed, resulting in ill-formed code. As with P7 AddAnnotation, this pattern cannot be reproduced (and would not occur) using our editing model, since we do not support ill-formed variability annotations.

C. Other Patterns

The remaining editing patterns are listed in Table III.

P11 WrapCode. This pattern describes cases where an existing piece of code is made variational, as shown below:

```
+ #ifdef ULTRA_LCD
    lcd_setalertstatuspgm(lcd_msg);
+ #endif
```

We show how to support this pattern in Fig. 9. We checkout with a trivial projection, delete the code that should be conditionally wrapped, in this case line two, and then checkin with an ambition that describes the configurations in which the code should no longer appear (e.g., $\neg$`ULTRA_LCD`).

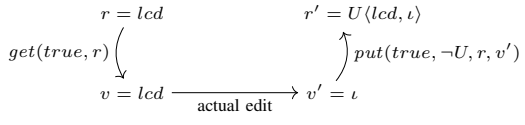


Fig. 9. P11 WrapCode editing workflow.

P12 UnwrapCode. This pattern describes the opposite case of the previous pattern. In this pattern, an existing piece of variational code is made non-variational, that is, the surrounding `#ifdef` and `#endif` annotations are removed, as shown below:

```
- #ifdef ULTRA_LCD
    lcd_setalertstatuspgm(lcd_msg);
- #endif
```

This pattern is not very amenable to the projectional editing model. The workflow for this pattern is shown in Fig. 10. Essentially, it requires to obtain the variants that do not include the code, re-adding the same code, and checkin with the same ambition as the projection.

Note that before minimization, the put will produce a choice $U\langle$`lcd`$,$`lcd`$\rangle$, where both alternatives are the same. This can be simplified to simply lcd using the choice idempotency minimization rule, resulting in r'.

TABLE III
OTHER PATTERNS

Name	#Multi	#Only	Example
P11 WrapCode	77	29	$e \to F\langle e, \iota\rangle$
P12 UnwrapCode	12	2	$F\langle e, \iota\rangle \to e$
P13 ChangePC	225	74	$F_1\langle e_1, e_2\rangle \to F_2\langle e_1, e_2\rangle$
P14 MoveElse	5	2	$F\langle e_1, e_2 \cdot e_3\rangle \to F\langle e_1 \cdot e_2, e_3\rangle$

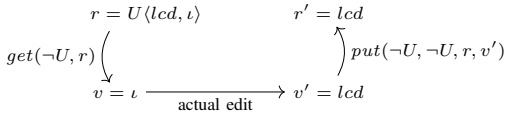


Fig. 10. P12 UnwrapCode editing workflow.

Observe that the manual edit to transform v into v' requires reproducing the code that was projected away during checkout. Although this can be accomplished for a single `#ifdef` with copy–paste, clearly this is not ideal. Therefore, this scenario is better supported by a non-projectional edit, doing a checkout with the trivial projection $true$, removing the `#ifdef` and `#endif` annotations, and using the ambition $true$ for checkin.

P13 ChangePC. This pattern describes cases where the PC associated with an `#ifdef` is changed, as shown below:

```
- #ifdef ULTRA_LCD
+ #if ULTRALCD && ULTIPANEL
    lcd_setalertstatuspgm(lcd_msg);
  #endif
```

In this example two changes have occurred: the `ULTRA_LCD` option has been renamed to `ULTRALCD`, and an additional constraint `ULTIPANEL` has also been added.

This pattern is perhaps better supported without a projectional edit because it would require to remove all code under the old PC and then add the same code that was removed under the new PC. In future work, we plan to explore operations for explicitly supporting such edits, including feature renaming and systematic modifications to PCs.

P14 MoveElse. This pattern captures cases where an `#else` annotation is moved in order to move some code from one set of configurations to another. This pattern is presented below:

```
  #ifdef ULTRA_LCD
    lcd_setalertstatuspgm(lcd_msg);
- #else
    alertstatuspgm(msg);
+ #else
   cleanup(msg);
  #endif
```

This (infrequent) pattern is another that is better supported without projectional editing, but it can be achieved by two edits as illustrated in Fig. 11. We use the first letter of lines 2, 4, and 6 from the pattern above, to indicate the respective line of code.

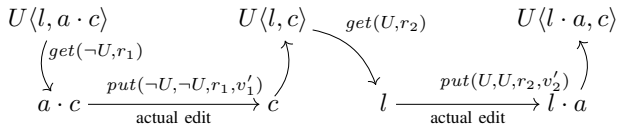


Fig. 11. P14 MoveElse editing workflow.

As with the P12 UnwrapCode pattern, this requires reproducing some part of the code between projectional edits (e.g., `alert` in this example).

VI. Results

We now present the results of applying the variation control system. All evaluation data is available in the online appendix.[5]

Our objective in this experiment is to verify if the edit patterns described in Sec. V are indeed supported, and what kind of projections and ambitions are used. Moreover, we are interested to see if there is any negative effect on the source code when using the prototype variation control system.

A. Applying the Changes

Following the methodology from Sec. IV, we randomly selected patches that belong to only one edit pattern, covering 12 edit patterns out of 14. Patches from the remaining two edit patterns (P7 AddAnnotation, P10 RemAnnotation) cannot be executed with the prototype. We therefore ignore these two update patterns and obtain 34 patches. One patch belonging to P12 UnwrapCode contains merge conflicts leading to an ill-formed variation and is excluded. In total we use the variation control system on 33 patches.

All the selected patches were successfully applied using the variation control system. The actual changes on the view were performed with a simple text editor. Note that for all patches that add or remove `#ifdef` blocks, we only touched the code between the annotations to realize the edit; the annotations are handled by the variation control system.

> *A projection-based variation control system can support all the presented edit patterns when no malformed variability annotations exist.*

B. Complexity of Projections and Ambitions

Since some patches required multiple steps to execute the change, we performed a total of 37 *projections*. Of these, 14 use one feature and 11 the trivial condition *true*. The remaining 12 projections use two, three or four features in their expressions. In three cases the projection is the conjunction of four features, making these projections more difficult to understand and use.

Yet, it is not uncommon that a developer needs to consider two or more features (i.e., ≥ 4 system variants) when fixing bugs. In fact, Abal et al. [2] identify 30 bugs that occur when there is a combination of at least two configuration options. In such cases, using a projection-based editing tool could simplify the task, focusing only on the variants in which the bug appears.

[5]http://bitbucket.org/modelsteam/2016-vcs-marlin

TABLE IV
LOC and NVAR metrics with the min, max, and median values for the 33 changes for our put function. Repository update represents the change done by the developer in the original Git repository of the project.

	LOC		NVAR	
	Our *put* function	Repository update	Our *put* function	Repository update
MIN	65	72	1	1
MAX	2448	2368	193	147
MEDIAN	447	449	20	21

In *ambitions*, the highest number of features is the same as in projections, four. But we see a decrease of trivial ambitions, which is expected, as for example P11 WrapCode edits may be performed on trivial projections, but require an ambition different than $true$. In one case the expression used for both projection and ambition is a conjunction of a feature and a disjunction, $p = A \wedge (B \vee C)$. Finally, for 18 changes the ambition equals the projection.

C. Metrics and Reduction Factors

Table IV shows the aggregate values (min, max, and median) of our metrics on the source code resulted from the update done by the prototype, and the original update from the Git repository. While our goal is not to improve code with regards to LOC or NVAR, Table IV shows that the prototype does not perform worse than the original update in almost all cases.

The boxplot in Fig. 12 shows the reduction factor for LOC and NVAR after the projection. For the LOC reduction factor when doing projections, we would expect it to be zero, in the case of using a trivial projection, or larger than zero when a non-trivial projection is used. Table IV and the boxplot confirm this hypothesis. The average number of LOC after projection is smaller than before projection. In one outlier case the LOC in the view was reduced by half, compared to the original file. The reason is that the source code has a sequence of `#if-#elif-#else-#endif` directives with many `#elif` branches, which naturally contradicted the projection. In such cases, the benefit of projecting views can be high, especially for code comprehension (e.g., to understand the control flow).

As we would also expect, NVAR is reduced when projecting the code, although this reduction is minimal in most cases. In our experiment, many changes are done on features that wrap an entire file's source code or use the trivial projection $true$. Nevertheless, in three outlier cases there is a high decrease in NVAR when many `#ifdef` blocks are projected away.

Finally, we investigate whether there is any degradation of code when using the variation control system. Comparing the

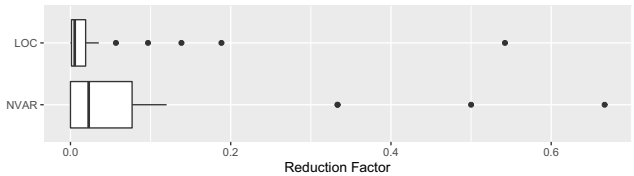


Fig. 12. Reduction factors for LOC and NVAR for the view

LOC between the update done by the variation control system and the original update, shows that the former has the same or less LOC in most cases. The differences mostly occur when two consecutive choices (`#ifdef` blocks) with the same PC are merged, or when one has the other one's negated PC, as formulated by the factoring rule shown in Fig. 3. This merging also affects the number of variation points, generating a lower number in the case of the variation control system's update.

A projection-based variation control system can be used to engineer a highly configurable system. Our prototype did not negatively impact the code in terms of LOC, NVAR. In some cases it even improved the code.

VII. Discussion

Let us now discuss challenges we encountered in the experiment (RQ2) and then get back to the editing operations (RQ1).

A. Challenges of Using the Variation Control System

Most importantly, the editing workflow is different. This requires some mental effort in understanding what projection and ambition to use. However, in our experience, choosing a projection and ambition was straightforward in most cases, but more difficult for changes that required two or three checkout/checkin cycles.

An interesting case to consider is choosing the ambition when making code optional, that is, wrapping existing code with a PC. Both P11 WrapCode and P5 AddIfdefWrapThen required the ambition to be the negated desired PC. The intuition is that the we have to choose an ambition that describes in which configurations should the code not appear. This may seem unintuitive at first, but it is easy to see why this is necessary. In a text editor or IDE, the user could select the code that should be under a PC and just enter that PC.

B. Edit Operations for a Variation Control System

Some of the identified edit patterns were difficult to replay using the projection-based editing workflow and our realization of the get/put functions. However, the edit patterns should not be seen as the edit operations a developer would use when using a variation control system. We use the edit patterns to derive, where needed, the *edit operations* for a variation control system. Most patterns can be used in a straightforward manner and do not require specialized operations. However, a variation control system would require a specialized edit operation for renaming and changing a PC. We also need better support for P12 UnwrapCode and P14 MoveElse patterns, as an extra copy-paste editing step is required. These would require more specialized primitive operations, ideally in a text editor or IDE. In future work we plan to define and implement these edit operations, and experiment how well they can be used.

Finally, we identified a limitation of the variation control system, that is not solved by any of the existing ones either. The generalized edit isolation principle (cf. Sec. III-B) raises the following problem: How to handle the cases when an ambition is *weaker* than the projection? An example scenario could be fixing a bug in a particular variant, where the fix might affect other variants as well. So instead of fixing the bug in all variants, we would like to have a specific projection, but then perform the change with a weaker ambition. Our definition of the put function (which conjoins projection and ambition) cannot handle this case. Solving this problem in a sound way is subject to future work.

VIII. Threats to Validity

A. Internal Validity

We consider our updates to be correct as the put function is correct by construction. To check for bugs in the prototype, we used KDiff3 to compare the update result of the prototype with the original update from the repository. We examined and compared the two updates visually. We double-checked the two cases where our update performs worse, which showed that the update result is correct and preserves semantics.

We developed a general parser for the C preprocessor language, as it is simpler and less error-prone than language-specific parsers. This allows to use the system with any source code that implements variability using preprocessor annotations.

Our definition and implementation of the variation control system might be incorrect, and we might have introduced bias when identifying the edit operations or conducting the experiment.

To identify edit operations we followed a systematic approach: first studying samples, then iteratively creating regular expressions to validate them on all 5640 Marlin patches. We also cross-checked the identified operations on 34,018 patches from Busybox, another highly configurable software from a different domain. 99.27% of the patches in Busybox were classified in one or more patterns that we previously identified.

In the experiment, we reduced bias of replaying changes with the variation control system by randomly selecting patches spanning all twelve edit operations considered.

B. External Validity

The identified edit patterns overlap with some extracted in previous work [17], [28], [7], which increases our confidence in the method and completeness. While more patterns might exist, our collection was sufficient for executing complex changes.

Composing edit operations can allow a user to execute the same change as in a normal editing model, modulo the number of steps. The edit operations are generic and also specific enough to allow to execute any change. Using the trivial projection and ambition $true$, the proposed workflow behaves similarly as the normal editing model.

We did not consider systems that use a variability model and a dedicated variability-aware build system for implementing more complex variability, such as the Linux Kernel or Busybox. However, the variation control system can still be used to directly manipulate the source code, as well as the variability model and the build files.

IX. Related Work

In addition to the three variation control systems discussed in Sec. II that our prototype is most similar to, other techniques to realize views on configurable code exist.

A. Views on Source Code

Atkins et al. develop a *version editor* that hides preprocessor directives, allowing to edit a particular variant of a source file [3]. Edits to the view are propagated back into the source file. Their study on a large telecommunication project shows a productivity increase of up to 40%. In comparison, we focused mainly in understanding what kind of operations should such a tool support, and if indeed, these operations can be used to maintain and evolve highly configurable software systems.

Hofer et al. argue that existing approaches to assist with handling the preprocessor are tied to IDEs, thus, their adoption rate is low [15]. They introduce the filesystem LEVIATHAN that mounts a view representing a variant. Heuristics are used to synchronize changes in the view with the source code in the physical storage. However, it does not allow modifying the structure of the conditional blocks when working on a view.

C-CLR is an Eclipse plugin that allows creating a view by selecting the respective preprocessor macros (features) [31]. The tool offers support for generating views, but not for executing changes and updating the view. Similarly, folding is used as a visualization technique by Kullbach et al. [22] to hide and unhide code in the GUPRO tool [8]. The idea is to fold parts of code (including preprocessor directives) and possibly labeling the fold to easily identify its purpose. Compared to these two works, we wanted to allow modifying the view and updating the repository with the new changes.

Kästner et al. propose colors to show annotated code corresponding to a feature [18], and implement the Colored IDE (CIDE). The tool requires disciplined preprocessor annotations, such that arbitrary code fragments cannot be annotated. A variant view shows annotated code fragments using a background color according to a feature selection. Markers are used to show code that belongs to features that are not selected.

A similar tool that uses colors (but lacks the ability to hide code) is developed by Le et al. [23]. Internally it uses the choice calculus. A controlled experiment with students shows that the prototype increases code comprehension compared to the C preprocessor tool. Users were more successful and efficient in completing their tasks and gave more correct answers, which motivates the use of dedicated variation control systems.

Janzen et al. propose to use a concept called *crosscutting effective views* to modularize concerns [16]. The *modules view* provides a decomposed structure in terms of module units of the program. A *classes view* shows the decomposed structure of classes. Changes applied to one view are reflected in the other view, which is automatically modified and updated. The tool stores the structure of the program internally, while the developer edits a so-called *virtual source file* [6].

B. Evolution of Highly Configurable Software

Several studies of changes performed to highly configurable software consider the variability model and the build system [7], [28], [25], whereas we focus only on the code level.

Dintzner et al. aggregate feature-evolution information by mining commits [7], including extensive information of what artifacts are affected. They mainly consider commits that touch `#ifdef` blocks. They create this data for the variability model, build system, and source code. Their focus is not to detect the exact type of changes, but to offer an overview of the evolution of features. In contrast, our work identifies what kind of code edits occur in real systems and whether these can be applied using a variation control system.

Passos et al. present a catalog of patterns on the co-evolution of features in the variability model, build system, and code, obtained from the Linux kernel [28]. Several patterns use only the variability model and/or the build files to add or remove features. Some of our patterns overlap with theirs: P3 AddIfdefElse corresponds to AVONMF (Add Visible Optional Non Modular Feature), P5 AddIfdefWrapThen to FCUTVOF (Featurize Compilation Unit to Visible Optional Feature), and P9 RemIfdef to RVONMF (Remove Visible Optional Non Modular Feature). The main difference between our work and theirs is that we did not want to understand how a system evolves in all three spaces, but whether source code changes can be realized using a variation control system.

X. Conclusion

Maintaining and evolving highly configurable software is challenging for many projects. Using a projection-based variation control system may overcome some of these challenges. But so far, the experience with *variation control systems* is limited.

In this paper, we have designed and formally described a variation control system that combines and extends concepts of prior proposals in the literature. In a study, we identified 14 variability-related edit patterns from the highly configurable 3D printer firmware Marlin. We used the patterns to derive what edit operations a projection-based variation control system needs to support. We then conducted an experiment using our variation control system prototype to replay real changes from the subject system, to show that it can in fact be used to maintain and evolve a highly configurable software system.

We found that while the projection-based editing model can support most edit operations, some are difficult to realize with this model and require extra effort. However, when executing changes with the prototype, we found that in most cases the code does not degrade with respect to code size and number of variation points, and that it is fairly easy to use. In a few cases, the view had considerably less code.

In future work, we strive to allow developers to work in parallel on a project, which means that we need to handle code merges, merge conflicts, and other possible code-integration aspects. We are also currently developing a user interface that is connected to the prototype, to allow us to implement and test the identified edit operations on highly configurable systems.

A.4 Paper D

Intention-Based Integration of Variants

Max Lillack University of Leipzig, Germany
Ștefan Stănciulescu IT University of Copenhagen, Denmark
Wilhelm Hedman Chalmers University of Technology, Sweden
Thorsten Berger Chalmers | University of Gothenburg, Sweden
Andrzej Wąsowski IT University of Copenhagen, Denmark

ABSTRACT

Cloning is an efficient and simple strategy to develop new variants of a system, well-supported by version-control systems. However, as the effort of long-term maintenance of clones outgrows the initial benefits, the variants often need to be re-engineered into an integrated and configurable platform. Such an integration is challenging, mostly because variation points need to be consistently introduced and properly assigned to features, to achieve the integration goal. In this sense, variant integration differs from traditional merging, which does not produce or organize variation points, but creates a single, non-variational system. Unfortunately, little support exists for the transformation phase of variant integration.

We introduce integration *intentions* as domain-specific actions (e.g., 'keep functionality' or ' keep as a configurable feature') allowing the integration editing to happen at a much more abstract level than in traditional merge tools. Developers interactively apply intentions on the source code supported by different views. Quickly applying or undoing intentions allows exploring the integration. We implement our approach in a full IDE tool supporting the integration of file variants that use preprocessor annotations for variation points. We evaluate our approach's correctness, completeness, and benefits in a set of user studies with altogether 31 participants who replay real integration steps mined from the history of three highly forked open-source projects, with file variants up to 4K lines of code. Our evaluation shows that our approach is correct, applicable, and reduces mistakes done by developers during integration.

ACM Reference format:
Max Lillack University of Leipzig, Germany, Ștefan Stănciulescu IT University of Copenhagen, Denmark, Wilhelm Hedman Chalmers University of Technology, Sweden, Thorsten Berger Chalmers | University of Gothenburg, Sweden, and Andrzej Wąsowski IT University of Copenhagen, Denmark . 2017. Intention-Based Integration of Variants.

1 INTRODUCTION

Many software systems need to exist in multiple variants. Innovators create variants to experiment with new ideas and products. Organizations create variants to customize systems towards market segments, hardware platforms, runtime environments, or non-functional properties (e.g., performance or energy consumption).

A widespread practice for creating variants is *cloning*—copying a variant's code and adapting it to the new requirements [8, 11, 12, 36]. Despite critique for introducing maintenance problems, cloning is appreciated as easy and readily available [20], commonly used in industrial [7, 11] and open-source [41] projects. Nowadays, cloning is often realized using facilities of version-control systems, which allow to quickly create *forks* (project-wide clones) and support some degree of traceability and back-porting facilities. However, in the long term, the effort for maintaining and evolving forked variants outweighs the initial benefits when more than a few variants exist.

Organizations often need to re-integrate variants and establish an *integrated platform*—a.k.a., software product line—sharing the previously separated features [8]. Such a platform is usually configurable through variation points (e.g., using compilation directives), which allow deriving individual variants that realize specific combinations of features. Some large integrated platforms were created from scratch, such as the Linux kernel with around 15,000 configuration options. However, most configurable platforms are the result of integrating formerly cloned variants [8, 19, 37].

Integrating variants is costly and risky. Developers need to understand the variants and their differences, to consistently introduce variation points reflecting the integration goal. For instance, if two functionalities are conflicting but should co-exist, variation points need to be incorporated that prevent deriving variants with both functionalities. If the variants already contain variation points, these need to be made consistent. If the C preprocessor is used, developers need to obtain a single `#if` hierarchy guarded by correct `#if` expressions—especially difficult when multiple variations in forks are overlapping. In general, developers are lacking tool support and often do not even attempt to integrate larger and conflicting variants [14, 31], which is especially difficult when variants have intricate and undisciplined `#if` structures. Such structures are typical in configurable C projects, including our evaluation subjects [28, 32, 43]. In fact, we can easily find commits in them where developers needed to fix `#if` structures.[1] Furthermore, a think-aloud exercise we conducted on manual variant integration with a plain diff tool exhibited, as we will show, problems with the alignment of changes among the variants, a lack of overview on the variants, and difficulties undoing parts of the integration.

These challenges of variant integration also illustrate the high-level differences to traditional code merging. The latter aims at creating a single implementation out of contributions coming from

ICSE 2018,

DOI: 10.1145/nnnnnnn.nnnnnnn

[1] https://git.io/v7pzb, https://git.io/v7pgL

parallel developments. The main challenge in code merging is conflict resolution. In contrast, variant integration aims at creating an integrated platform. The main challenge is to control features coming from various forks, while incorporating common improvements into the core assets, and dropping uninteresting changes.

Prior work has mainly focused on understanding the commonalities and differences among variants [6, 41], largely sidestepping the actual integration. In this work, we focus on supporting engineers who work on integrating variants into an integrated platform. We do not aim at alleviating developer's domain knowledge about the variants, but at supporting variant comprehension using views, and at helping developers to structure and organize the changes into variation points and features in an iterative process.

We present the approach and tool suite INCLINE (intention-based clone integration) to support the variant-integration process. To support variant comprehension, INCLINE offers developers various views on the variant code that shall be integrated. To support the actual integration, INCLINE offers *intentions*—domain-specific actions that can be declared by developers over code—lines or individual nodes of the abstract syntax tree (AST)—in the views, defining how the functionalities should be integrated (e.g., keep functionality, remove functionality, or keep functionality as a configurable feature). INCLINE resolves the intentions by automatically creating variation points (i.e., a correct `#if` structure) in the target integrated platform, which can be previewed interactively by developers. The developers can quickly change or undo intentions, allowing an iterative process to explore the effect of intentions and reaching the desired integration. In summary, we help to raise the abstraction level at which the integrator works, from writing low-level `#if` directives and expressions to declaring architectural *intentions*, helping to focus on the actual integration problem. In other words, intentions are intentionally simple, but can have complex resolutions (e.g., through interactions with other intentions)—a complexity that is hidden from the developer.

We evaluate INCLINE's completeness, correctness, and the benefits for developers when integrating variants by simulating real variant-integration steps mined from the history of three popular open-source systems with forks: Marlin, a 3D printer firmware; Vim, a UNIX text editor; and BusyBox, a tool suite of shell tools. We perform a set of consecutive simulations with a total of 31 participants, comprising file variants up to 4K lines of code, cumulating in a controlled experiment with 12 experienced PhD students who perform integration tasks in a realistic setting. Our evaluation shows the completeness and correctness of our approach and, most importantly, that declaring intentions is simple and intuitive, compared manual integration using Eclipse. Developers also make less mistakes when using INCLINE.

We contribute: (i) a definition of *intentions* as a vocabulary to express integration goals for individual variant-related functionalities, (ii) a formalization of the intentions as transformations on a variational AST, (iii) a complete IDE tool that provides views together with facilities for declaring intentions, executing them as in-place transformations, and for editing code to adjust the result, and (iv) an online appendix [2] with the INCLINE tool, evaluation material, and a replication package.

```
#ifdef ULTIPANEL
 uint8_t lastEncoderBits;
 uint32_t encoderPosition;
 #if PIN_EXISTS(SD_DETECT)
  uint8_t lcd_sd_status;
  #endif
#endif  // ULTIPANEL

menu_t cM = lcd_status_scrn;
 bool ignore_click = false;
```

```
#ifdef ULTIPANEL
 uint8_t lastEncoderBits;
  int8_t encoderDiff;
 uint32_t encoderPosition;
 #if (SDCARDDETECT > 0)
  bool lcd_oldcardstatus;
 #endif
#endif //ULTIPANEL

menu_t cM = lcd_status_scrn;
```

Figure 1: Code excerpts from Marlin's mainline (left) and the corresponding fork (right). Colors indicate differences.

In our view, the research on generic re-engineering and project manipulation tools is presently in rather early stages. Few general tools exist. Mostly experiences from manual re-engineering processes are reported. This is despite the increasing amount of legacy code and, in this context, the increasing amount of forked software variants. With this tool we hope not only to help concrete forked projects, but also to inspire the research community to work on the related challenges, towards more effective and intelligent re-engineering tools.

2 MOTIVATION AND BACKGROUND

2.1 Variant Integration versus Code Merging

Variant integration differs from traditional code merging [34] that focuses on merging changes performed in isolation into a *single system*. Merging does not directly support realizing variants or building a configurable integrated platform. Traditional merge algorithms combine as much code as possible and delegate conflicts they cannot resolve to the developer. In contrast, our focus is on consistently and efficiently transforming system variants—that were developed in parallel and that can realize conflicting (i.e., mutual-exclusive) functional- or non-functional requirements—into a configurable platform, where they can co-exist. Deciding whether a change should be shared by all variants, or be specific to a particular variant, has to be done regardless whether a merge conflict occurs or not. Even smoothly merging changes might need to become optional in a given project.

Research on variant integration can be found under the broader research area of re-engineering legacy products into a software product line [6]. However, works in this area almost solely focus on discovering commonalities and variabilities between codebases to gain an understanding of how similar or far apart they are, together with research on identifying and locating features.

2.2 Variant Integration Challenges

Consider Fig. 1, which shows code from the Marlin mainline repository (our running example for the remainder). The right-hand side shows the corresponding part in a Marlin fork. Both variants evolved after forking: the highlighted lines were changed or added.

When integrating the variants, the developer needs to decide about a target `#if` structure, which changes should be included, and under what *presence conditions* (Boolean expressions over features, determining when the respective code should be included in a variant derived from the integrated platform). Specifically, the fork's

```
5  ...
6  #ifdef FORK
7    #if (SDCARDDETECT > 0)
8      bool lcd_oldcardstatus;
9  #else
10   #if PIN_EXISTS(SD_DETECT)
11     uint8_t lcd_sd_status;
12 #endif
13 #endif //ULTIPANEL
14 ...
```

Figure 2: An invalid `#if` structure produced by `diff -D`

line 4 (added variable `encoderDiff`) could be either made mandatory or optional, the latter by adding an `#ifdef` with an expression specifying that the line belongs to a feature. Furthermore, this feature could be the same as for the fork's lines 6–8 or a different one, according to the actual integration goal based on the developer's expert knowledge. Furthermore, consider lines 5–7 in the mainline and lines 6–8 in the fork. In our example, the final relative order of these blocks does not matter, since they have no side effects. But, if these were statements, it might be necessary to enforce an order.

Consider viewing these excerpts in a traditional diff tool, which highlights the differences, but does not help with the integration. Working with a diff tool, the developer needs to comprehend such diffs, while editing the text to create an appropriate `#if` structure.

While cheap-and-quick ways of integrating variants exist, they typically do not lead to the desired integration goal, as they disregard the domain knowledge. The most trivial solution is to create a new feature that represents the variant (we will call such a feature `FORK` in the remainder) and wrap the complete files in an `#ifdef-#else` block. This fails to recognize any commonalities and creates a highly redundant integrated platform, not providing much benefit. Another approach could be wrapping all differences in `#ifdef` blocks. GNU's *diff* tool provides the option `-D`, which wraps any differences between two files in `#ifdefs`.[2] In our example, we would obtain an invalid `#if` structure, since the newly created directives interfere with existing ones in the variants, as illustrated in Fig. 2.

The strategy of doing pairwise diffing and then wrapping the changes using `#ifdefs` is relatively common. For instance, Danfoss, used this to integrate forks [18, 19], creating "features" representing individual variants. GNU diff was also used. While this integration was relatively quick (months) for a system with around 1.5M lines of code, it took the organization years to achieve the desired integrated platform. It required deciding, among others, which functionality to keep, remove, or which should become a new feature. In this process, the platform was iteratively verified. The effort was mostly manual, with minimal tool support. It was challenging to refactor the variant-based "features" into around 1000 variant-cross-cutting features, which required regrouping variation points logically.

As the example (Fig. 1) illustrates, many ways exist to integrate variants, and the developer has to first create and then iteratively refactor conditional structures. This task is both laborious and error-prone, but hard to automate entirely due to dependence on architectural decisions and domain knowledge.

Another challenge is the cognitive load introduced by the C preprocessor, whose `#if` directives clutter source code and challenge program comprehension [40, 43], as the developer needs to work

[2] `diff -w -D FORK file1 file2`, where -w ignores whitespace and FORK is the feature

```
1  #ifdef ULTIPANEL
2   uint8_t lastEncoderBits;
3   #ifdef FORK
4    int8_t encoderDiff;
5   #endif
6   uint32_t encoderPosition;
7   #ifdef FORK
8   #if SDCARDDETECT > 0
9     bool lcd_oldcardstatus;
10   #endif
11  #else
12   #if PIN_EXISTS(SD_DETECT)
13    uint8_t lcd_sd_status;
14   #endif
15  #endif
16 #endif
17
18 menu_t cM = lcd_status_scrn;
19 #ifndef FORK
20  bool ignore_click = false;
21 #endif
```

Figure 3: The integrated AST for the mainline/fork of Fig. 1

with all variants simultaneously. This can easily lead to code ending up in the wrong variant (the wrong `#if` block) [33].

In summary, integrating forked variants into a product line is a challenging and time-consuming effort that needs to be handled iteratively. One needs to decide how to integrate separately developed variants. If variants already contain variation points, their variation spaces also need to be merged: a deeper understanding of existing and desired variation points is necessary.

2.3 Think-Aloud Exercise

To better understand these challenges, we conducted a think-aloud exercise where three participants (student developers) executed two integration tasks, one from the Marlin project and one from Busybox, using Eclipse's integrated diff tool. The participants received three files: a file with the code from the main, a file with fork code, and the target solution. The main Marlin file has over 2,000 lines of code and over 100 `#ifdef` blocks. The Busybox file has 25 lines of code and only two `#ifdef` blocks. The fork files are similarly in size and number of `#ifdef` blocks. During the integration, the participants were asked to speak aloud what are they doing to integrate the two files. We recorded this process and reviewed the spoken comments. All users report that the diff tool creates alignment problems which lead to mismatching. The tool tries to align the text and the changes, but often fails to do it correctly. The second problem is that using a single view where variants are explored, and modified as well, hinders the overall integration process. More exactly, the users need better views to explore the variants, and how changes influence the end result, with the ability to easily undo changes. One user mentions the need for creating a new file where all the changes should be stored, such that the two variant views can be used to understand the differences. Another user is overwhelmed by the fact that the diff tool highlights every line.

3 THE VARIANT INTEGRATION PROCESS

We now present the proposed integration process. We integrate two variants at a time, the mainline and a fork, in three steps:

1. Automatically generate an integrated AST. We automatically create an integrated AST from the mainline and the fork

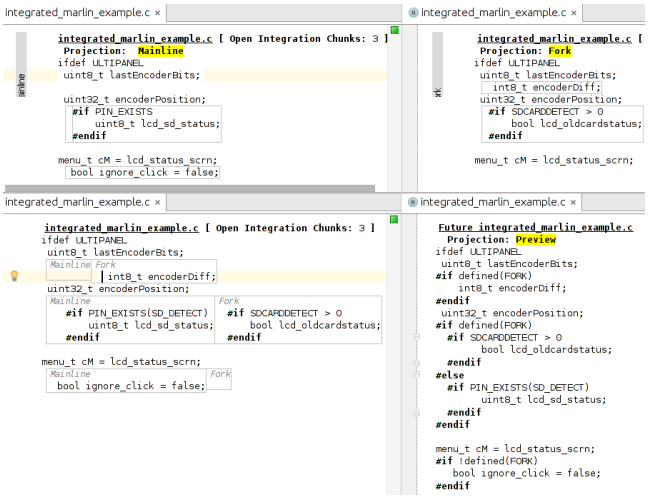


Figure 4: Screenshot of views supporting the integration: *mainline view* (top left), *fork view* (top right), *integrated side-by-side view* (bottom left), and *result view* (bottom right)

variant. In addition to the features already contained in the input variants, we introduce a special feature `FORK`. The AST captures the commonalities and differences between two variants. AST nodes that are specific to the fork variant are annotated with the presence condition `FORK` and those specific to the mainline with `!FORK`.

Figure 3 illustrates the creation of the integrated AST in concrete syntax for our example from Fig. 1. Given that the AST is always syntactically correct, we can derive, compile, and execute individual variants during the integration process, e.g., to run a test suite.

2. Explore the integrated AST using views. The developer explores the AST and navigates through it using different *views* that ease comprehension of the variants and their differences. We provide five different views implemented in our tool, whereas further views can be realized on top of our implementation. Figure 4 illustrates four of them. The *mainline view* and the *fork view* (top left and right of Fig. 4) render a *projection* of the integrated AST where only a subset of the derivable variants is shown. The *mainline view* shows all variants with feature `FORK` disabled, and *fork view* with this feature enabled. The *integrated side-by-side view* (bottom left of Fig. 4) renders the whole integrated AST, but with the differences between mainline and fork arranged next to each other without `#ifdef` directives. Alternatively, this view can be switched to the *integrated view* (Fig. 3), which shows the code as an `#if` structure corresponding to the integrated AST. In these views, the developer can define intentions (explained shortly), and explore and edit code. The *result view* (bottom right of Fig. 4) renders a preview of the final result, after all intentions are resolved and any manual edits applied.

Using all views, the developer can see how a change affects the individual variants, and the resulting integrated platform. All views are projections from the integrated AST, and are updated synchronously whenever a change is made.

3. Edit integrated AST and add integration intentions. The developer edits the integrated AST (using one of the *integrated views*, the *mainline* or the *fork view*) and declares intentions on AST nodes. The user interface supports navigating the variant differences, and declaring intention on selected code. The *result view* is updated when an intention is added or removed. Finally, the developer commits the intentions, which updates the integrated AST. Thereafter, a developer can repeat steps 2 and 3 with the freshly modified AST until all changes are handled.

4 THE INTEGRATION INTENTIONS

Intentions specify the goal of integrating a change, that is, how the integrated AST should be customized with respect to the input variants. For instance, a developer could ask: should the change be made common to all variants? Or only to some of them? Or should it remain variant-specific? Is the change standalone or is it intended to belong to a feature implementation?

We extracted a set of intentions from the history of Marlin repositories. We studied merge commits involving preprocessor directives, pull requests, and conflicting commits—all these being examples of developers performing integration, while dealing with variability. For these, we tried to understand the original intention of the developer, which we carefully assessed and discussed. As a result we propose the following intentions to be used for high-level control of the integration process: *Keep*, *KeepAsFeature*, *Exclusive*, *Remove*, *AssignFeature*, *Order*. We define them in detail below.

4.1 Variational AST and Views

A variational *AST* is a syntax tree with embedded `#ifdefs`. For simplicity of the discussion we often see it just as a set of nodes (*Nodes*). For each node n, we identify a sequence of conditions, $c_1^n, c_2^n, \cdots, c_k^n$, used in the `#ifdefs` on the path from the root (c_1^n) of the AST to n (c_k^n). A node that is not wrapped by any `#ifdef` (non-variable node) has an empty sequence of conditions.

We define the *presence condition* $pc(n)$ to be the conjunction of all conditions used by the `#ifdefs` the node $n \in Nodes$ is contained in. The presence condition of a non-variable node is *true*.

$$pc(n) = \bigwedge_{i \in 1..k} c_i^n$$

We define a *block* to be a set of nodes in the underlying AST: $block \in \mathcal{P}(Nodes)$. We also introduce an *order* of nodes, that describes the syntactic order of the C/C++ program. If a node n_1 exists before node n_2 in the syntactic order, we write $n_1 < n_2$.

A *view* is a projection of the AST showing only a specific set of variants specified by a *view constraint* ρ over features. For the views *mainline* and *fork*, $\rho = \neg FORK$ and $\rho = FORK$, respectively. For the *integrated views* $\rho = true$. Of course, ρ can be a more complex expression, not just a literal, to filter out more variability not relevant for the integration task at hand. In the remainder, we limit ourselves to simple view constraints, though. Finally, note that the view through which an intention is declared forms the context for the intention, and as such influences its resolution.

To determine how conditions are shown in a view, we substitute every occurrence of the view constraint in the conditions with *true*:

$$\forall n \in Nodes \quad c_\rho^n = c^n[\rho \leftarrow true],$$

where c_ρ^n denotes the conditions shown in the view.

In the variant views, we render the variational AST as follows:

- We hide nodes where $pc(n)[\rho \leftarrow true] \equiv false$

```
#ifndef FORK // block_not_fork
 int servo_e1[] = SE
 int servo_e2[] = SEA
#else // block_fork
 int16_t servo_e1 = SE
 int16_t servo_e2[] = SEA
#endif
```

```
 int servo_e1[] = SE
 int servo_e2[] = SEA
#ifdef FORK
 int16_t servo_e1 = SE
 int16_t servo_e2 = SEA
#endif
```

Figure 5: *Keep* intention (left) and result (right)

- We show nodes without the surrounding `#ifdef` if $c_k^n \equiv true$
- For the remaining nodes, we simplify presence conditions with respect to the view constraint using an SMT solver

Special care needs to be taken for blocks containing nodes with *complex* presence conditions. Complex conditions contain both the view constraint ρ and some unrelated terms. When using views, a complex presence condition is shown as a non-complex condition, since the view constraint is simplified. Still, the hidden view constraint is part of the context that the intentions are declared in, just as is if the node was implicitly wrapped in an `#ifdef` with the view constraint as its condition. Formally, we say a condition is *complex* iff $c^n \not\equiv c^n[\rho \leftarrow true]$. For an `#ifdef` node n with a *complex* condition, we rewrite the sequence of conditions so that it ends with the view constraint. We use the notation $pc(n, \rho)$ to denote the presence condition of node n in a view with constraint ρ:

$$pc(n, \rho) = \bigwedge_{i \in 1..k} c_i^n \wedge \rho$$

4.2 Semantics of Intentions

We now define the individual intentions and illustrate them with examples. Intentions are partial functions transforming ASTs. We formalize their semantics as effects they have on the presence conditions and ordering of nodes. Later, in Sec. 4.3, we show how the intentions are resolved (implemented) on the AST. The figures (for instance, Fig. 5) show the integrated AST on the left, selected nodes on which an intention is declared in gray, and the desired result on the right. The examples use the *integrated view*, which shows all variants at once. The verbosity of this view makes it most suitable to explain how intentions work. For each intention we use the notation $pc'(n)$ to illustrate the resulting presence condition of the node n, and $pc(n)$ for the presence condition before the intention resolution.

Keep. The *Keep* intention includes a block as it appears in mainline or fork in an unconditional manner, without guarding it with any additional feature.

Consider the example in Fig. 5 where we define *block_not_fork* to represent the set of nodes in the ¬*FORK* branch highlighted with gray, and the *block_fork* represents the set of nodes in the *FORK* branch. The fork changes the type of the *servo* variables to be 16-bit signed integers, because different hardware and compiler are used for this variant. During the integration process, it is decided that the hardware used in the fork should no longer be supported, and only the code from mainline is kept. We apply the *Keep* intention on *block_not_fork* set of nodes.

The right side shows the result of applying *Keep* on the selected *block_not_fork*. The nodes from the fork should be removed (which can be done with the dual of *Keep*, *Remove*, described below). Note

```
#ifdef FORK // block_fork
 card.pauseSDPrint();
#endif
```

```
#ifdef SDSUPPORT
 card.pauseSDPrint();
#endif
```

Figure 6: *KeepAsFeature* intention (left) and result (right)

that the integration is not completed, as there is still a block from the forked variant, which should be resolved later.

The effect of *Keep(block)* on the presence conditions is:

$$pc'(n) = \begin{cases} c_1^n \wedge \ldots \wedge c_{k-1}^n, & \text{if } n \in block \\ pc(n), & otherwise \end{cases}$$

The nodes for which *Keep* was declared should no longer be under the constraint created by the `#ifdef` that directly wraps those nodes (in the example we drop `!FORK`). Their new presence condition is the conjunction of all but the last constraint that directly wrapped the nodes. All nodes that are not part of the intention are unchanged.

KeepAsFeature. The *KeepAsFeature* intention preserves a block from one of the variants, but makes it conditionally present,only linked to a certain feature or combination of features. It wraps the block with a new presence condition given with the intention.

In the example of Fig. 6, a fork developer added functionality to pause a 3D print from an SD card. Not concerned with other devices than the one for which the fork was developed, she included the new behavior unconditionally. However, in the integration process, it became clear that this functionality only makes sense in variants supporting SD cards, thus, it needs to be included conditionally. The desired result is shown on the right side of the figure.

KeepAsFeature(block, F) is defined as replacing the last constraint from the sequence of constraints with the new presence condition:

$$pc'(n) = \begin{cases} c_1^n \wedge \ldots \wedge c_{k-1}^n \wedge F, & \text{if } n \in block \\ pc(n), & otherwise \end{cases}$$

Exclusive. This intention declares that two code blocks should be mutually exclusive (enforcing the separation of conflicting functionality), controlled by a choice condition.

In Fig. 7 the fork introduces a new optional feature `FIL_DISPLAY` and keeps the line that prints a message on the LCD display under a specific condition. The integration requires keeping the optional feature and ensuring that when this feature is not selected a message is shown on the LCD (to not break the mainline variant). Therefore,

Figure 7: *Exclusive* intention with the three parameters $block_1$, $block_2$, *FIL_DISPLAY* (left) and result (right)

```
#ifndef FORK //block2
 lcd.print(msg);
#else
 #ifdef FIL_DISPLAY //block1
  if(condition){
   lcd.print(msg);
  }else{
   lcd.print(trnsf(data));}
  }
 #endif
#endif
```

```
#ifndef FIL_DISPLAY
 lcd.print(msg);
#else
 if(condition){
  lcd.print(msg);
 }else{
  lcd.print(trnsf(data));
 }
#endif
```

```
#ifdef SD
 card.pauseSDprint();
#endif
```

```
#ifdef SDSUPPORT
 card.pauseSDprint();
#endif
```

Figure 8: *AssignFeature* intention (left) and result (right)

we keep both blocks as a mutually exclusive implementation using the *Exclusive* intention.

We introduce the helper function *common(block)* which returns the longest common subsequence of conditions of nodes in the block (semantically akin to the prime implicate of the set of presence conditions of the block's nodes):

$$common(block) = c_1 \wedge \ldots \wedge c_s \text{ such that}$$
$$\forall_{n \in block} \bigwedge_{i \in 1..k} c_i^n \rightarrow \bigwedge_{i \in 1..s} c_i \text{ and } s \text{ is maximal such.}$$

Then the *Exclusive(block1, block2, F)* has the effect as follows:

$$pc'(n) = \begin{cases} common(block_1 \cup block_2) \wedge F, & \text{if } n \in block_1 \\ common(block_1 \cup block_2) \wedge \neg F, & \text{if } n \in block_2 \\ pc(n), & otherwise \end{cases}$$

We use the common conditions of the nodes in $block_1$ and $block_2$ as the basis and then include the feature condition F (or its negation) to control the selection of the variant.

Remove. This intention deletes the selected nodes from the AST. By definition, it ensures that the selected nodes do not exist in the updated AST':

$$\forall n \in block \quad n \notin AST'$$

AssignFeature. This intention is used when code was already integrated, but its presence condition should be changed (e.g., simplified, weakened, or strengthened). This intention can only be declared for complete #if-#else-#endif blocks. Fig. 8 shows the renaming of feature *SD* (left) to *SDSUPPORT* (right).

The effect of *AssignFeature(n, F)* is that the last constraint of nodes from both branches (`#if` and `#else` of the `#ifdef` block) is replaced with the given feature, and respectively the negated feature:

$$pc'(n) = \begin{cases} c_1^n \wedge \ldots \wedge c_{k-1}^n \wedge F, & \text{if } n \in \textit{if branch of } n \\ c_1^n \wedge \ldots \wedge c_{k-1}^n \wedge \neg F, & \text{if } n \in \textit{else branch of } n \\ pc(n), & otherwise. \end{cases}$$

Order. This intention prescribes an order of blocks from the variants for the integrated AST (with respect to the concrete syntax). As a notation, we resort to the operators > and <, which declare that the first block be put before the second block, and vice versa. This intention re-orders blocks or ensures their correct order during integration, especially when further intentions are applied.

For example, we apply the intention *Keep* on a mainline and a fork block, but we want the mainline code to be executed first, we declare $order(block\_fork, block\_main, <)$, which then yields the correct order.

Composing Intentions. It is often necessary to declare multiple intentions. Fig. 9 shows two function calls, *card.pauseSDprint()*, one that is guarded by an `#ifdef`, the other is not. A developer can classify this change as a bug-fix and resolve it using two intentions: *Keep* fork code, but *Remove* mainline code. For convenience, she can select in the UI, when applying a *Keep* intention on all the nodes in *block_fork*, that the *Remove* intention should be automatically applied on nodes of *block_not_fork*.

4.3 Intention Resolution

We conceive an AST in-place transformation to resolve intentions, which needs to consider *all* declared intentions at once, since intentions can interact. Although intentions are declared on blocks of nodes, their resolution will affect other nodes.

We resolve intentions in a specific *order*, first based on our own defined priority of intentions, and second based on the AST structure (top down, from outermost to innermost node). The priorities (descending, from resolved first to resolved last) are: *Keep*, *KeepAsFeature*, *Exclusive*, *Order*, *AssignFeature*, *Remove*.

During resolution, each intention will use the result from the previous intention as its context, therefore the execution of one intention will likely influence the context for intentions that are declared for nested (child) nodes. The idea is that intentions for nested nodes are more specific than intentions for outer nodes.

Fig. 10 illustrates the importance of the resolution order by showing two declared *Keep* intentions, one for lines 2–4 (`#ifdef`) and one for line 3. We resolve the intention for the `#ifdef` first by unwrapping the node from the surrounding condition `FORK`. Then, we apply *Keep* on the node in now line 2 (previously, line 3), which moves it out of the condition `SDSUPPORT`. The two empty `#ifdef` are removed in a cleanup operation, leaving only the line `card.pauseSDprint()`. Resolving reversed would move line 3 out of the condition `SDSUPPORT` and then move the now empty `#ifdef` out of the outer `#ifdef FORK`.

We now explain each intention's resolution, which typically includes *moving* nodes and (un)wrapping them from `#if`s.

Keep. For $Keep(block)$ we first group the *block* nodes into a list of adjacent nodes, that is, each group of nodes will have the same parent and is in the same branch. For each group, we split any surrounding `#ifdef` into blocks before and after the group. Both blocks get the same presence condition. Fig. 11 illustrates this transformation, showing how the resolution of Keep splits an `#if` block.

KeepAsFeature. We resolve $KeepAsFeature(block, feature)$ similarly to *Keep*, except that the nodes unwrapped during the *Keep* step are now wrapped in a new `#ifdef` with the condition *feature*.

Exclusive. We resolve $Exclusive(block_1, block_2, feature)$ using *Keep* on $block_1$ and $block_2$. The results are wrapped in a new `#ifdef` where the nodes of $block_1$ are moved to the *true* branch and the nodes from $block_2$ are used in the *else* branch. The condition of the new `#ifdef` is given by the parameter *feature*.

Order. To resolve $order(block_1, block_2, <)$ we assume the nodes of $block_1$ and $block_2$ have the same parent. If all nodes in $block_2$ are,

Figure 9: Composition of *Keep* intention (left, gray lines) with *Remove* intention (left, blue lines), and result (right)

```
#ifdef FORK //block_fork
 #ifdef SDSUPPORT
  card.pauseSDprint();
 #endif
#else //block_not_fork
  card.pauseSDPrint();
#endif
```

```
#ifdef SDSUPPORT
 card.pauseSDPrint();
#endif
```

```
1 #ifdef FORK
2  #ifdef SDSUPPORT
3   card.pauseSDprint();
4  #endif
5 #endif
```

Figure 10: Two *Keep* intentions (lines 2–4 and 3) to illustrate the importance of a resolution order

based on their position in the parent's list of child nodes, before the nodes of $block_1$, we switch their position. In a similar way, we perform this action with reversed parameters for operator >.

Remove. We resolve *Remove*(*block*) simply by removing all nodes in *block* from the AST. If a node was already removed (e.g., when *Remove* was declared for an ancestor of the node) nothing is done.

5 TOOL IMPLEMENTATION

We use the language workbench JetBrains MPS [1] on top of which we realize our tool INCLINE [2]. MPS relies on projectional editing, which is well-suited for creating editable views. Projectional editing (a.k.a., syntax-directed editing or structural editing) [9, 42] is conceptually different from traditional parser-based editing, since the user's program-editing gestures directly change the underlying AST, which is still rendered into concrete syntax the user sees. Projectional editing eases language composition and allows flexible notations (e.g., the *integrated side-by-side view*).

We make use of MPS' meta-modeling facilities and implement our own stripped version of the C preprocessor language, including only the `#ifdef`, `#else`, `#if`, `#elif`, `#endif` macros. For example, we define `#ifdef` as a language concept that defines a node containing a condition and having three child nodes for the nodes in the *true*, *else if*, and *else* branches. In addition, we add a `Text` concept that represents a line of actual source code (C/C++ in our case) in the AST. Each of the concepts can have different properties, e.g., the `#ifdef` concept has a *condition* attribute that holds the presence condition associated with this `#ifdef`. These language concepts suffice to represent source code files that use preprocessor annotations to implement variability.

We specify how the user can interact with our language, that is, edit the AST using the *editor* functionality of MPS. For each concept we create a visualization definition, which controls the rendering of it. The intentions are implemented using MPS *actions*, user-invoked commands to change the AST, selectable via the UI.

We implement three additional components. PPParse is a parser for C preprocessor directives that creates the initial AST, based on the clang compiler infrastructure. PPMerge is a tool to create the integrated AST of two files using the C preprocessor. It first parses the input files to create XML-based ASTs, then constructs the integrated AST using JNDiff [10], which diffs the ASTs, followed by transforming the diff into optional changes in the input file. PPConstraintSolver is a tool to perform operations on preprocessor conditions using the SMT solver Z3[35]. Specifically, we use it to calculate projections, i.e., to decide *if* and *how* conditions are shown in the views.

Figure 11: Resolution of *Keep* (`b()`) splits `#if` block (right)

```
#if A
 a()
 b()
 c()
#endif
```

```
#if A
 a()
#endif
 b()
#if A
 c()
#endif
```

6 EVALUATION

To study how well our approach and tool supports developers during variant integration, we conduct a series of simulations of real development. We conceive integration tasks from real evolution steps of the three often forked open-source systems Marlin, Vim, and BusyBox, as opposed to artificially creating tasks—a reasonable trade-off between complexity and real-world cases.

6.1 Subject Systems

All subjects use preprocessor annotations to realize variability. We enhance external validity by sampling over their source files and forks (>4,000 Marlin forks exist). Our selection of forks (three Marlin forks, one fork each from BusyBox and Vim) is based on the fork's activity, viability (i.e., has variability-related changes), and popularity (stars on Github). This way, we avoid bias towards a particular usage of the preprocessor. In addition, a study including Vim and BusyBox confirms that the preprocessor is used similarly among open-source and industrial systems [17].

Marlin (>40 KLOC of C++ code) has many forks that evolved separately or independently added new functionality. Given this richness of changes and new functionality, and existing re-integration efforts of the community, Marlin is an ideal subject for our evaluation. We mined from 1,715 approved pull requests and three forks.

BusyBox (>160 KLOC of C code) is a tool suite of common shell programs (e.g., grep, cut). We use a fork tailored for Android. [3]

Vim (>300 KLOC of C code) is a popular console-based text editor for Unix-like systems. We use a fork that adds support for OS2. [4]

6.2 Evaluation Design

Evaluating an interactive tool with a rich UI such as INCLINE is challenging, since users do not only face the inherent complexity of the integration task, but also have to learn the tool and handle potential usability issues of a research prototype. We conduct a series of simulations, going from an internal evaluation via a preliminary user study with MSc students to a controlled experiment with experienced PhD students in a realistic integration setting.

First, we verify the completeness of our intentions. We replay a set of real-world merge commits from Marlin by applying intentions and checking that the commits are realizable. We also check that the result is semantically identical (i.e., code lines obtain the correct presence condition) to the original merge commit.

Second, we validate the correctness of the implemented intention resolutions and investigate the usability and scalability of INCLINE. Three authors simulate ten Marlin integration tasks and thoroughly cross-check the correctness by reviewing the resulting integrated files. While this does not allow comparing the efficiency, it validates that INCLINE produces correctly integrated files if intentions

[3] https://github.com/jcadduono/android_external_busybox
[4] https://github.com/h-east/vim/tree/clpum

are assigned correctly, and that it scales. It also provides valuable experiences to improve INCLINE and to fix bugs.

Third, we investigate the INCLINE approach with 16 MSc students to learn how they perceive the intentions, the views, and the editing efficiency and usability of our tool. We conduct a user study where we create realistic, but reasonably small tasks from Vim and BusyBox and let participants solve them using INCLINE and Eclipse. Since the participants lack domain knowledge for the integration tasks, we give the final result. As such, this experiment also lets us obtain information about the pure editing efficiency with INCLINE (recall its underlying projectional editor, which can cause editing challenges [9]), potential improvements, and bugs.

Fourth, after improving INCLINE with insights from the steps above, we validate INCLINE in a realistic setting. We conduct a controlled experiment with 12 experienced PhD students. We reuse the Vim and BusyBox tasks, but instead of providing the final result (which, as we learned from the user study, lets developer just apply low-level edits without understanding what they are doing), we provide detailed domain knowledge about the variants to be integrated. Tasks are solved with INCLINE and Eclipse using a 2x2 Latin square design. This controlled experiment lets us obtain information about the efficiency of integrating smaller files.

In summary, this set of consecutive experiments lets us obtain information about the benefits and limitations of our approach. As we will show, the scalability of INCLINE to files of up to 4K LOC, together with the benefits measured for integrating smaller files, evidences the applicability of INCLINE to smaller and larger integration project. Yet, we believe that training and experience can further increase the efficiency of developers working with INCLINE, which needs to be validated in a longitudinal study. Such a study is beyond the scope of this paper, but part of our future work.

6.3 Completeness of Intentions

Methodology. To show that the defined intentions are sufficient to handle real-world integration tasks, we replay non-trivial (conflicting) merges from Marlin history. We retrieve all 2,065 merge commits of the mainline, and extract those that had conflicts, to identify complex merge tasks, yielding 49 merges. We discard two merges that had conflicts only in documentation files, two that conflicted in whitespace, three that conflicted due to configuration changes. Another three merges are discarded because some related artifact had syntax errors and could not be compiled. Additionally, four merges are discarded because they simply accepted the mainline changes as evolution (empty changeset). We use the remaining 35 merge commits as tasks.

Results. We successfully simulate all 35 commits using intentions. Details are shown in the online appendix [2] together with examples illustrating the intentions. As we will show in Sec. 6.5 and 6.6, examples mined from Vim and BusyBox can be handled by our intentions as well.

The set of intentions suffices for real-world variant integration.

6.4 Correctness and Scalability

Methodology. With an internal study we validate that INCLINE produces correct results when intentions are assigned correctly, and that we can use it on large files without scalability problems.

We simulate ten integrations by randomly sampling seven commits tasks from the 35 merge commits in the previous experiment, and conceive three tasks simulating the integration of files from Marlin forks[5]. The selected forks contain significant changes to the mainline, covering both evolution and new features.

Most tasks comprise only a single file, some two or three, but each file can be very large (up to nearly 4,000 LOC and hundreds of `#ifdef` blocks in a single file). The first three authors serve as evaluators, among whom we distribute the ten tasks to execute with INCLINE. We then manually peer-review the integration results to detect any errors that INCLINE might have introduced. For the seven tasks based on merge commits, we compare to the actual merge result. For the three tasks based on the fork integration, the correct results was determined during the peer review.

Results. All of the observed errors could be explained by errors done by the user or errors introduced by the tool. We fixed errors in the implementation and analyzed mistakes done by us to improve usability. This shows that the intention resolution, as defined in Sec. 4.2, works as expected. Furthermore, the broad range of file sizes (from tens up to thousands of lines of code) evidences the scalability of INCLINE.

INCLINE produces the correct output when correct intentions are applied and it scales to files up to 4K lines of code.

6.5 User Perception and Editing Efficiency

Methodology. We continue with a user study to get first-hand experience how users deal with INCLINE and its UI. We recruit 16 MSc students to execute two newly developed integration tasks with INCLINE and Eclipse.

The two new integration tasks are derived from BusyBox (P1) and Vim (P2), by using files from the main project variant as well as a fork of each project (see Sec. 6.1). We select chunks of code based on our understanding and experience with the systems, as well as code blocks that involve integrating variability (`#ifdef` blocks). We merge these chunks into one condensed file for brevity and comprehension. The end result is an integrated file consisting of 74 LOC (P1), and 50 LOC (P2). The P1 file has 8 `#ifdef` blocks, with 37 LOC within these blocks; the P2 file contains 5 `#ifdef` blocks, with 32 LOC within these blocks. These tasks represent a good trade-off between complexity and real-world integration scenarios.

The participants are given the (correct) target solution and brief description of the integration goal. Giving the target solution reduces the influence of (a lack of) domain knowledge because all participants work towards the same goal. On the other hand, it is less realistic because the participants do not need to understand the example code or think on the level of integration goals. We observe their usage of the tools (through screen recordings), and at the end we ask them to take part in an exit questionnaire.

Results. We observe that INCLINE users only need to declare few intentions to reach their desired result, whereas Eclipse users use

[5] github.com/[jcrocholl | esenapaj | Marlin_STM32]/Marlin

the keyboard much more heavily. INCLINE users mostly used the intentions *Keep* and *Remove*, since they are the most obvious intentions for new users and they were sufficient for the selected tasks. The user's behavior and their feedback suggest they miss more advanced functionality, e.g., the side-by-side view which was disabled by default. We also learn that the UI needs better highlighting of the intentions that were applied, that keyboard shortcuts would help to quickly apply an intention (the users needed to use a menu button to reach the intentions), that the multiple views help exploring the code to reach consensus on how to integrate it, and that the arrangement of views shown in Fig. 4 is most intuitive.

Not surprisingly, the Eclipse tasks are solved faster, since we provided the final result, and as such, solving the integration task amounts to straightforward, low-level editing. Analyzing the recordings of the Eclipse tasks in fact shows that the participants seem to thrive with the direct low-level editing of the code. They quickly start copy-pasting code, and introduce preprocessor annotations to match their solution to the actual target solution. INCLINE users are slower as they need to understand the integration solution and map it back to intentions first, which is more demanding. In summary, this shortcoming in plain editing efficiency illustrates an important limitation of our approach. Yet, even in INCLINE, developers could resort to writing preprocessor annotations by hand, making up for this limitation, which we did not instruct them.

6.6 Benefits of INCLINE

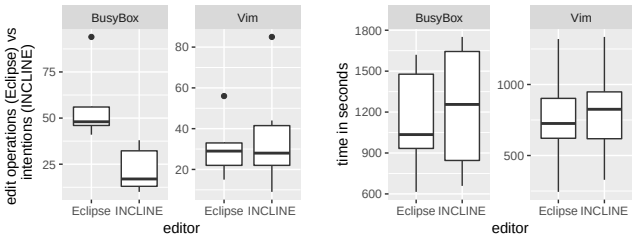
Methodology. We conduct a controlled experiment with 12 experienced PhD students who are familiar with the C preprocessor. Recall that the setting should be realistic, so we adapt the situation where a developer who has domain knowledge about the variants shall integrate them. To this end, we provide a detailed, but abstract explanation of the purpose of the variants' individual parts and how they should be integrated.

We use a 2x2 within-subjects counterbalanced Latin square design and reuse the Vim and BusyBox tasks. That is, each participant performs two tasks, using two treatments: Eclipse, and INCLINE on P1 or P2, in a random order to reduce learning effects. Using a within-subjects design, we can have a lower number of participants, while every subject participates in each task. Furthermore, we mitigate learning effects by randomizing the order of the tasks (counterbalanced part of the design).

Participants are trained through a video tutorial on how to use both tools, as well as being instructed on preprocessor usage (they only needed to use `#ifdef`, `#else`, and `#endif`). Then, we asked the participants to solve a warmup task extracted from Marlin, to get familiar with the tools. We record the screens and log information about keystrokes (in Eclipse) and intentions (in INCLINE).

We compare the performance of participants with both tools by measuring the mistakes done per task, the time to complete each task, and the number of edit operations (and number of intentions) applied per task as a proxy measure of effort.

We count mistakes done by the participants as follows. For Eclipse, we check if the end result is the same as the expected result. A mistake can be a missing preprocessor annotation, missing code or extra code. For INCLINE, we check for wrong intentions or no intentions applied by the participant that leads to errors in the resulting file. For both tools, errors concerning comments are counted as a half mistake, errors in formatting of code is ignored.



(a) Intentions/edit operations **(b) Task completion times**

Figure 12: Participants' effort and completion times

Results. INCLINE participants did visibly less mistakes than participants using Eclipse (7 vs 17.5). Furthermore, only four participants did mistakes in INCLINE compared to 11 participants in Eclipse. This is no surprise, as INCLINE has better support for keeping or removing code without needing to copy&paste or create `#ifdef` structures (a good source of errors). Users made mistakes in INCLINE when they missed relevant nodes in the declared intentions, declared incorrect intentions, or declared different intentions for the same node with an unexpected result for the user. Common mistakes with Eclipse included failures in the `#ifdef` structure, leaving code that should be removed or removing too much code.

INCLINE is also better in terms of intentions needed to execute the task (see Fig. 12a). In BusyBox, INCLINE users need only a handful of intentions to integrate the two variants, whereas BusyBox requires almost 50 edit operations for achieving the same goal. In perspective, Vim is less demanding for Eclipse users, and INCLINE users use a very similar number of intentions to execute the task.

If we compare the times to execute the task, INCLINE integrations are almost as fast as the ones in Eclipse (see Fig. 12b). The reason for being slower is twofold. First, participants spent a lot of time (which we count in the result) reading back and forth through the descriptions to understand their integration goal. Second, some participants were always verifying the preview view after applying an intention. A possible explanation for the latter case is that users are not very familiar with the tool and intentions, and thus either do not trust the tool or are not sure if they applied the right intention. However, this is exactly where INCLINE shines as it facilitates exploration, quick undo, and offers multiple views for manipulating and understanding the different variants and the integration result. Along these lines, one of the participants mentions that *"It was really useful to declare all the intentions while still having the original files in sight and previewing the result."*

INCLINE users are almost as fast as Eclipse users, but perform much fewer mistakes.

6.7 User Survey

After both user studies with student developers, we asked to fill in a survey questionnaire. We received 25 responses (cf. appendix [2]).

Participants agree in majority (over 90%) that the *Keep* and *Remove* intentions are intuitive. However, the *Exclusive* intention seems more confusing, because the user cannot directly select it.

Interestingly, over 50% of the participants consider that integration with INCLINE is faster than with Eclipse. One potential reason is that by not doing many copy pasting operations or editing text, INCLINE feels faster through the usage of intentions, though results show the two tools are close. Similarly, over 50% of the participants agree that intention based integration is not complex, suggesting that there is potential for intention-based integrations.

When asked what are the advantages of using intention-based over manual integration, some participants mentioned that *"It doesn't require manual code rewrite so I believe it could be easier avoid unintentional bugs in code and subtle differences"* or *"You get the preview of the result and the projections side-by-side, which seems hugely helpful when you don't have a clear integration goal. Harder to make syntactic mistakes."* There is in general a consensus that *"It's much more intuitive and less error-prone"*.

6.8 Threats to Validity

External Validity. We mitigate selection bias, as the main threat in our evaluation, by using multiple open source projects that have been actively developed and many variants have emerged. We use both main source files as well as forks to create realistic integration tasks. For the controlled experiment, we recruited experienced PhD students. However, only basic program understanding was required, and we recapped the preprocessor use, which mitigated any potential differences in programming experience among the participants. Furthermore, students are known to perform like industry participants in similar conditions [16, 38].

Internal Validity. Simple bugs in the tool chain can hide or distract from potential correctness issues of the intention concept. To mitigate this, we classify any errors in the result files as bugs in the tool chain, which are not related to specific tasks or the intention concept, and focus on errors where a user declared the wrong intentions or the intention resolution performed not as specified.

The experiment participants using INCLINE have disadvantages compared to plain merge tools, mostly due to the lacking experience and the rough UI of a research prototype. We mitigate this threat by training users through a tutorial and a warmup task on how to use INCLINE. Furthermore, we randomly assigned the tasks to the participants, minimizing the risk of learning effects.

7 RELATED WORK

Product-Line Adoption. Many works focus on re-engineering a single system into a software product line [21, 22, 24, 39], typically proposing refactoring techniques for creating configurable platforms. The main difference is that we focus on integrating multiple system variants (originating from cloning) into a product line, systematically guiding the process with intentions and views.

Product-Line Evolution. Other works provide intelligent support for evolving a product line. For instance, Liebig et al. [29] provide three refactorings (rename identifier, extract function, inline function) that are proven to preserve the variants in a configurable platform. The resolution of our intentions can also be seen as a refactoring, but it is explicitly not variant-preserving. Yet, automatically detecting and applying refactorings to improve the quality and structure of our integrated AST would be valuable future work.

Re-Engineering Multiple Variants. Compared to existing approaches, in summary, the novelties of our work are: (1) a set of intuitive intentions for various integration goals, controlling how code is integrated, which improves over character-based changes done in traditional merge/diff tools; (2) editable views that improve comprehension of the variants, the automated integration, and the end result; (3) doing edits on the variational AST to explore the effect of intentions and manual edits, with support for undo; (4) a tool chain aimed at C/C++ systems using the preprocessor for variation points, where we ensure their correct handling; and (5) instead of proposing automatic merges, we rather strive to support developers with intuitive, partially automated mechanisms.

A recent mapping study on re-engineering variants into product lines identifies 119 papers on this topic. However, the large majority focuses on detecting and analyzing commonalities and variabilities of the variant systems, together with feature identification and location [6]. Only few support the actual variant integration.

Rubin et al. present a conceptual framework with seven operators usable to re-engineer cloned variants into a product line [37]. The operators are abstract and some are related to our intentions. Yet, none is implemented. In contrast, we provide full tool support.

Fischer et al. [13] propose a method to detect reusable features among variants, allowing to compose them to derive a new system. Martinez et al. present a framework for re-engineering a set of assets into a product line [30]. The framework can be extended and customized to support different kinds of artifacts. Klatt et al. [23] present a tool for consolidating cloned product variants. It enhances the initially created integrated platform by providing a variation point analysis that provides recommendations for a developer to aggregate variation points. Ziadi et al. [44] automatically create a feature model and a software product line from a set of variants. All these works lack support for handling variability using preprocessor directives as the most common technique for variation points.

Finally, case studies of manual re-engineering exist. For instance, Hetrick et al. re-engineer cloned variants into a product line, extracting core assets from existing codebases, creating variation points, and switching to product line engineering [15]. Jepsen et al. [19] compute pairwise differences of two products, and wrap differences using `#ifdef` to create the initial integrated platform. The platform was iteratively refined, deciding to keep, remove or introduce a new feature, which took several years to complete [18].

Software Merging. Recall that variant integration is different from traditional merging. Still, we are inspired by techniques known from it. Our technique for creating the initial integrated platform works on ASTs and as such is related to structural merge [4, 5, 27, 34], which also relies on ASTs. Furthermore, detecting structural differences is used for many kinds of models in software engineering research [26]. For instance, JNDiff [10] implements a generic differencing algorithm for any XML-based model. Finally, model merging [25] and model composition [3] are related approaches to combine existing models, ASTs in our case, to one resulting model.

8 CONCLUSION

Forking is a fast way for prototyping and creating new variants. However, the long-term costs can easily outrun the initial benefits, which often requires re-integrating such forks.

We presented an approach to integrate forked variants into a configurable integrated platform. The core idea of our approach is to offer a set of intuitive *integration intentions* resembling domain-specific actions to execute the integrations, at the core of a tool-supported variant-integration process. Instead of focusing on low-level `#if` directives, the developer can express the integration goal using intentions declared on code blocks of the original variants, make edits to the code, and immediately observe the result. At any time, the platform is in a consistent state, and variants can be derived for validation and verification. Our evaluation showed the benefits of our approach, substantially reducing the number of editing operations required. We also showed that our approach can handle complex integration tasks and merges with conflicts. Declaring intentions was easy, and only rarely direct code editing was required. Although understanding the integration goal is sometimes difficult, the different views help to explore and navigate the code. Applying intentions and undoing them is particularly useful to explore the result, before committing the changes. Participants did mistakes with INCLINE when the integration goal was not well understood and applied the wrong intentions or no intentions at all. Often, complex tasks could be solved more efficiently with intentions than editing source code directly through a merge tool.

REFERENCES

[1] Jetbrains MPS. http://www.jetbrains.com/mps.

[2] Online Appendix. https://sites.google.com/site/myicse2018paper/.

[3] S. Apel, C. Kastner, and C. Lengauer. Language-independent and automated software composition: The featurehouse experience. *IEEE Trans. Softw. Eng.*, 39(1):63–79, Jan. 2013.

[4] S. Apel, O. Leßenich, and C. Lengauer. Structured merge with auto-tuning: Balancing precision and performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 120–129, New York, NY, USA, 2012. ACM.

[5] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner. Semistructured merge: Rethinking merge in revision control systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[6] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering*, pages 1–45, 2017.

[7] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wasowski. Three cases of feature-based variability modeling in industry. In *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, pages 302–319, 2014.

[8] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A survey of variability modeling in industrial practice. In *VaMoS*, 2013.

[9] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund. Efficiency of projectional editing: A controlled experiment. In *24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2016.

[10] A. Di Iorio, M. Schirinzi, F. Vitali, and C. Marchetti. A natural and multi-layered approach to detect changes in tree-based textual documents. In *Proceedings of the 11th International Conference on Enterprise Information Systems*, pages 90–101, Berlin, Heidelberg, 2009. Springer.

[11] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An exploratory study of cloning in industrial software product lines. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, pages 25–34, 2013.

[12] A. N. Duc, A. Mockus, R. L. Hackbarth, and J. D. Palframan. Forking and coordination in multi-platform development: a case study. In *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, pages 59:1–59:10, 2014.

[13] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 391–400, 2014.

[14] M. L. Guimarães and A. R. Silva. Improving early detection of software merge conflicts. In *ICSE*, pages 342–352, Piscataway, NJ, USA, 2012. IEEE Press.

[15] W. A. Hetrick, C. W. Krueger, and J. G. Moore. Incremental return on incremental investment: Engenio's transition to software product line practice. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 798–804, 2006.

[16] M. Höst, B. Regnell, and C. Wohlin. Using Students As Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Softw. Engg.*, 5(3):201–214, Nov. 2000.

[17] 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*, 2015.

[18] H. P. Jepsen and D. Beuche. Running a Software Product Line: Standing Still is Going Backwards. In *SPLC*, 2009.

[19] H. P. Jepsen, J. G. Dall, and D. Beuche. Minimally invasive migration to software product lines. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*, pages 203–211, 2007.

[20] C. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[21] C. Kästner, S. Apel, and D. S. Batory. A case study implementing features using aspectj. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*, pages 223–232, 2007.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming*, pages 220–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[23] B. Klatt, M. Küster, and K. Krogmann. A graph-based analysis concept to derive a variation point design from product copies. *Proc of REVE*, 13, 2013.

[24] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. Refactoring a legacy component for reuse in a software product line: a case study. *Journal of Software Maintenance*, 18(2):109–132, 2006.

[25] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Merging models with the epsilon merging language (eml). In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*, pages 215–229, Berlin, Heidelberg, 2006. Springer-Verlag.

[26] D. S. Kolovos, D. D. Ruscio, A. Pierantonio, and R. F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 1–6, May 2009.

[27] O. Leßenich, S. Apel, and C. Lengauer. Balancing precision and performance in structured merge. *Automated Software Engineering*, 22(3):367–397, 2015.

[28] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. 2010.

[29] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer. Morpheus: Variability-aware refactoring in the wild. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, 2015.

[30] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon. Bottom-up adoption of software product lines: a generic and extensible approach. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 101–110, 2015.

[31] S. McKee, N. Nelson, A. Sarma, and D. Dig. Software Practitioner Perspectives on Merge Conflicts and Resolutions. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution*, ICSME'17, 2017.

[32] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. The love/hate relationship with the c preprocessor: An interview study. In *ECOOP*, 2015.

[33] J. Melo, C. Brabrand, and A. Wąsowski. How Does the Degree of Variability Affect Bug-Finding? In *International Conference on Software Engineering (ICSE)*, 2016.

[34] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Software Eng.*, 28(5):449–462, 2002.

[35] L. D. Moura and N. Bjørner. Z3: An efficient smt solver. In *Proc. Int'l Conf. on Tools and algorithms for the construction and analysis of systems (TACAS/ETAPS)*, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[36] J. Rubin, K. Czarnecki, and M. Chechik. Managing cloned variants: a framework and experience. In *17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan - August 26 - 30, 2013*, pages 101–110, 2013.

[37] J. Rubin, K. Czarnecki, and M. Chechik. Cloned product variants: from ad-hoc to managed software product lines. *STTT*, 17(5):627–646, 2015.

[38] P. Runeson. Using Students as Experiment Subjects—An Analysis on Graduate and Freshmen Student Data. In *Proc. EASE*, 2003.

[39] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake. Variant-preserving refactoring in feature-oriented software product lines. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, 2012.

[40] H. Spencer and C. Geoff. #ifdef Considered Harmful, or Portability Experience With C News. In *USENIX Summer Technical Conference*, pages 185–198, 1992.

[41] Ș. Stănciulescu, S. Schulze, and A. Wąsowski. Forked and Integrated Variants in an Open-Source Firmware Project. In *31st International Conference on Software Maintenance and Evolution (ICSME)*, 2015.

[42] M. Völter, J. Siegmund, T. Berger, and B. Kolb. Towards User-Friendly Projectional Editors. In *SLE*, 2014.

A.5 Marlin Survey Active Forks

Marlin firmware - Active fork

Dear participant,

Thank you for taking part in our survey. We are three researchers, Andrzej Wasowski and Stefan Stanciulescu from IT University of Copenhagen in Denmark, and Sandro Schulze from TU Braunschweig Germany, who are interested in the evolution of the Marlin firmware. We want to understand how Marlin and its variants evolve, and how Git and Github contribute to this development.
Our work is part of an european project, VARIES - www.varies.eu, that deals with handling configurable software, and finding ways to better manage this kind of software and its development.
The purpose of this survey is for research only, and it will only be used by the three of us.
All the information collected in this survey will be anonymous (unless you provide us with further details, such as an e-mail address), and we will preserve confidentiality.

Completion Time: Approximately 10 minutes

Thank you,
Andrzej, Sandro and Stefan

*Required

1. **Were you active on your fork (did some changes, synchronized with upstream, etc)?** *
   Note: If you have forked this Marlin repository more than once, please respond to the following questions having in mind just one of the forks (e.g. the one that was the most important, with most contributions).
   Mark only one oval.

   - ◯ Yes
   - ◯ No *Skip to "Not active fork."*

Questions regarding Marlin and your fork

2. **What was your reason for forking Marlin repository? (choose one or more suitable answers)** *
   Tick all that apply.

   - ☐ I wanted to configure the software to work with my own hardware
   - ☐ I am actively using this firmware, thus I wanted to have a copy of the repository
   - ☐ I am a developer or an enthusiast, and I wanted to modify Marlin for my own needs
   - ☐ I am a developer or an enthusiast, and I wanted to contribute to Marlin with patches for features and bug-fixes
   - ☐ I had no specific intention when I forked Marlin
   - ☐ Other: _____

Synchronization

3. **Do you synchronize your fork with Marlin?** *
   do you take (pull) changes that are done in the main Marlin repository, and apply them to your fork
   Mark only one oval.

   - ◯ Yes *Skip to question 4.*
   - ◯ No *Skip to question 5.*

Start this form over.

Synchronization with Marlin

4. **How often do you synchronize with Marlin?**
   Mark only one oval.

   - ◯ Rarely
   - ◯ Every month
   - ◯ Every week

Skip to question 6.

Synchronization with Marlin

5. **What is the reason for not synchronizing with Marlin? (choose one or more suitable answers)**
   Tick all that apply.

   - ☐ I do not know how to pull changes from Marlin repository
   - ☐ I do not want any changes from Marlin
   - ☐ The changes are not interesting for me
   - ☐ I am unaware of the new changes in Marlin
   - ☐ Other: _____

Pull-request and patches

6. **Did you create pull requests for Marlin?** *
   Mark only one oval.

   - ◯ Yes *Skip to question 7.*
   - ◯ No *Skip to question 7.*
   - ◯ I do not know what a pull-request is

Patches

7. **Did you have patches accepted in Marlin or other related project?** *
   Mark only one oval.

   - ◯ Yes *Skip to question 8.*
   - ◯ No *Skip to question 10.*

Patches explanations

8. **What was the main challenge in getting your patches accepted?**

9. **How many patches did you get accepted?**

Skip to question 10.

Pull-request

10. **Are you using other mechanism than Github's pull-request to send patches to the maintainers of Marlin (or related projects?)** *
    Mark only one oval.
    - Yes *Skip to question 11.*
    - No *Skip to question 12.*

Pull request explanation

11. **How are you sending patches to maintainers of Marlin?**
    Please explain

New functionality (or features)

12. **Have you added new functionality in the code using #ifdef annotations?** *
    Mark only one oval.
    - Yes *Skip to question 13.*
    - No *Skip to question 14.*

New functionality using #ifdef annotations

Please explain why did you introduce new functionality using #ifdef preprocessor annotations

13. **Explain why using #ifdef. Did you see any advantage in using the preprocessor annotation?**

Contact information

14. **Would you be interested in receving a copy of our research paper? If yes, please enter a valid e-mail address.** *
    Else write no

15. **Would it be ok to be contacted for further information? If yes, please enter a valid e-mail address.** *
    Else write no

Skip to "Outro."

Not active fork

It looks like you selected that you did not do any changes to your fork, nor have synchronized with upstream (Marlin). Please complete this survey instead.
https://docs.google.com/forms/d/1esY8pMkQ_BQ_QOO2bemqlCeuJGEUEirTKXZ09M6k7fM/viewform?usp=send_form

Start this form over.

Outro

Thank your for your time.
Andrzej, Sandro and Stefan

A.6 Marlin Survey Inactive Forks

Marlin firmware - Inactive fork

Dear participant,

Thank you for taking part in our survey. We are three researchers, Andrzej Wasowski and Stefan Stanciulescu from IT University of Copenhagen in Denmark, and Sandro Schulze from TU Braunschweig Germany, who are interested in the evolution of the Marlin firmware. We want to understand how Marlin and its variants evolve, and how Git and Github contribute to this development.
Our work is part of an european project, VARIES - www.varies.eu, that deals with handling configurable software, and finding ways to better manage this kind of software and its development.
The purpose of this survey is for research only, and it will only be used by the three of us.
All the information collected in this survey will be anonymous (unless you provide us with further details, such as an e-mail address), and we will preserve confidentiality.

Completion Time: Approximately 10 minutes

Thank you,
Andrzej, Sandro and Stefan

*Required

1. **Were you active on your fork (did some changes, synchronized with upstream, etc) and pushed your changes on your repository on Github?** *
   Note: If you have forked this Marlin repository more than once, please respond to the following questions having in mind just one of the forks (e.g. the one that was the most important, with most contributions).
   Mark only one oval.

   - ◯ Yes *Skip to "Active fork."*
   - ◯ No

Questions regarding Marlin and your fork

2. **What was your reason for forking Marlin repository? (choose one or more suitable answers)** *
   Tick all that apply.

   - ☐ I wanted to configure the software to work with my own hardware
   - ☐ I am actively using this firmware, thus I wanted to have a copy of the repository
   - ☐ I am a developer or an enthusiast, and I wanted to modify Marlin for my own needs
   - ☐ I am a developer or an enthusiast, and I wanted to contribute to Marlin with patches for features and bug-fixes
   - ☐ I had no specific intention when I forked Marlin
   - ☐ Other: _____

Local changes

3. **Do you have local changes (on your local repository on your own computer) that you have not pushed on Github?** *

   Mark only one oval.

   - ◯ Yes *Skip to question 4.*
   - ◯ No *Skip to question 5.*

Start this form over.

Reasons for only local changes

4. **What is the reason for not pushing changes up to your repository on Github (to the remote repository)?** *

   If there exist local changes (commits) but these are not pushed on Github, we would like to understand why.
   Tick all that apply.

   - ☐ I do not want to push my changes to the public repository
   - ☐ I forgot about my changes, and I do not need them to exist in the remote repository
   - ☐ I did a change once and then did not use the software anymore
   - ☐ Other: _____

Branches

5. **Do you use local branches (only on your own local repository on your computer) to work with different configurations of the software?** *

   Mark only one oval.

   - ◯ Yes
   - ◯ No

6. **Do you use local branches (only on your own local repository on your computer) to develop additional features or fix bugs?** *

   Mark only one oval.

   - ◯ Yes
   - ◯ No

Pull-request

7. **Are you using other mechanism than Github's pull-request to send patches to the maintainers of Marlin (or related projects?)** *

   Mark only one oval.

   - ◯ Yes *Skip to question 8.*
   - ◯ No *Skip to question 9.*
   - ◯ I do not know what a pull-request is *Skip to question 9.*

Pull requests explanation

8. **How are you sending patches to maintainers of Marlin?**
   Please explain

Synchronization with upstream

9. **Why have you not synchronized with Marlin?** *
   Tick all that apply.

   - [ ] I did not know that Marlin has evolved
   - [ ] I do not know how to pull changes from Marlin repository
   - [ ] I do not want any changes from Marlin
   - [ ] I did not keep up with the project after I forked
   - [ ] Other:

Contact information

10. **Would you be interested in receving a copy of our research paper? If yes, please enter a valid e-mail address.** *
    Else write no

11. **Would it be ok to be contacted for further information? If yes, please enter a valid e-mail address.** *
    Else write no

Skip to "Outro."

Active fork

It looks like you selected that you did changes to your fork, or maybe have synchronized with upstream (Marlin). Please complete this survey instead.
https://docs.google.com/forms/d/16ZdnXmdlLzRjMPEnryS8VMeWAT2bWZPQ_DlMwVQ8R1M/viewform?usp=send_form

Start this form over.

Outro

Thank your for your time.
Andrzej, Sandro and Stefan

A.7 Intentions Examples

Keep

Code taken from `temperature_47c1ea7_integrated.cpp`.

See also `Remove`.

Case I: Explicit keep

Expected outcome: accept change as evolution.

Integrated view:

```
#include "watchdog.h"
#if !defined(FORK)
  #include "language.h"
#endif
#include "Sd2PinMap.h"
```

Mainline view:

```
#include "watchdog.h"
#include "language.h"
#include "Sd2PinMap.h"
```

Clone view:

```
#include "watchdog.h"
#include "Sd2PinMap.h"
```

Resolutions:

In order to achieve the expected outcome of integrating this change as evolution, there are two possible actions, both involving `Keep`, but applied on two different views. Let `K` denote that the `Keep` intention is applied to the node. Only one of a) and b) below are required, but they are equal.

a) Keep on Integrated view

```
#include "watchdog.h"
#if !defined(FORK)
K  #include "language.h"
#endif
#include "Sd2PinMap.h"
```

b) Keep on Mainline view

```
#include "watchdog.h"
K#include "language.h"
#include "Sd2PinMap.h"
```

Outcome:

```
#include "watchdog.h"
#include "language.h"
#include "Sd2PinMap.h"
```

Case II: Implicit keep

Expected outcome: Discard the changes from the fork, accept the changes in the mainline as evolution.

Integrated view:

```
#if !defined(FORK)
  int cycles = 0;
#else
  int cycles=0;
#endif
```

Mainline view:

```
int cycles = 0;
```

Clone view:

```
int cycles=0;
```

Resolutions:

Note that the actual applied intention is `Remove`, denoted by `R`. The `K` is added by the tool after prompting the user. This is triggered because all the children of the if/else-block has a `Remove` intention applied - therefore the user is asked whether they want to automatically apply `Keep` to all children of the sibling if/else-block.

To reach the expected outcome, we can apply the `Remove` intention in either the integrated view or the clone view. For view clarity, we only show the integrated view here, but the same principle as above still stands.

Integrated view:

```
#if !defined(FORK)
K  int cycles = 0; // All nodes in this block get Keep because
    we apply the Remove intention below!
#else
R  int cycles=0; // Apply Remove to all nodes in this block -
    prompt user for Keep on all nodes in if-block.
#endif
```

Outcome:

```
int cycles = 0;
```

Keep as feature

Code taken from `373f3ec/Marlin_main.cpp`.

Expected outcome: Integrate Delta-specific changes from fork as features.

Integrated view

```
static float destination[NUM_AXIS] = {  0.0, 0.0, 0.0, 0.0};
#if defined(FORK)
  static float delta[3] = {0.0, 0.0, 0.0};
#endif
static float offset[3] = {0.0, 0.0, 0.0};
```

Mainline view:

```
static float destination[NUM_AXIS] = {  0.0, 0.0, 0.0, 0.0};
static float offset[3] = {0.0, 0.0, 0.0};
```

Clone view:

```
static float destination[NUM_AXIS] = {  0.0, 0.0, 0.0, 0.0};
static float delta[3] = {0.0, 0.0, 0.0};
static float offset[3] = {0.0, 0.0, 0.0};
```

Resolution:

In order to achieve the expected outcome of integrating this change as a feature, there are two possible actions, both involving `KeepAsFeature`, but applied on two different views. Below we will show the Clone view only, but the exact same principle applies for the Integrated view also. Let `F` denote that the `KeepAsFeature` intention is applied to the node. The presence condition used for the applied intention is `defined(DELTA)`.

KeepAsFeature on Clone view

```
static float destination[NUM_AXIS] = {  0.0, 0.0, 0.0, 0.0};
F static float delta[3] = {0.0, 0.0, 0.0}; // PC: defined(DELTA)
static float offset[3] = {0.0, 0.0, 0.0};
```

Outcome:

```
static float destination[NUM_AXIS] = {  0.0, 0.0, 0.0, 0.0};
#if defined(DELTA)
  static float delta[3] = {0.0, 0.0, 0.0};
#endif
static float offset[3] = {0.0, 0.0, 0.0};
```

Remove

Code taken from `temperature_47c1ea7_integrated.cpp`.

See also `Keep`.

Case I: Explicit remove

Expected outcome: Discard the variable `ms`, as it is no longer required.

Integrated view:

```
#if !defined(FORK)
  unsigned long ms = millis();
#endif
if (temp_meas_ready == true) {
  // stuff
}
```

Mainline view:

```
unsigned long ms = millis();
if (temp_meas_ready == true) {
  // stuff
}
```

Clone view:

```
if (temp_meas_ready == true) {
  // stuff
}
```

Resolutions:

In order to achieve the expected outcome of integrating this change as evolution, there are two possible actions, both involving `Remove`, but applied on two different views. Let `R` denote that the `Remove` intention is applied to the node. Only one of a) and b) below are required, but they are equal.

a) Remove on Integrated view

```
#if !defined(FORK)
R  unsigned long ms = millis();
#endif
if (temp_meas_ready == true) {
  // stuff
}
```

b) Remove on Mainline view

```
R unsigned long ms = millis();
if (temp_meas_ready == true) {
  // stuff
}
```

Outcome:

```
if (temp_meas_ready == true) {
  // stuff
}
```

Case II: Implicit remove

Expected outcome: Discard the changes from the fork, accept the changes in the mainline as evolution.

Integrated view:

```
#if !defined(FORK)
  SERIAL_ECHOLN(MSG_PID_AUTOTUNE_START);
#else
  SERIAL_ECHOLN("PID Autotune start");
#endif
```

Mainline view:

```
SERIAL_ECHOLN(MSG_PID_AUTOTUNE_START);
```

Clone view:

```
SERIAL_ECHOLN("PID Autotune start");
```

Resolutions:

Note that the actual applied intention is `Keep`, denoted by `K`. The `R` is added by the tool after prompting the user. This is triggered because all the children of the if/else-block has a `Keep` intention applied - therefore the user is asked whether they want to automatically apply `Remove` to all children of the sibling if/else-block.

To reach the expected outcome, we can apply the `Keep` intention in either the integrated view or the clone view. For view clarity, we only show the integrated view here, but the same principle as above still stands.

Integrated view:

```
#if !defined(FORK)
K  SERIAL_ECHOLN(MSG_PID_AUTOTUNE_START); // Apply Keep to all
    nodes in this block - prompt user for Remove on all nodes in
    else-block.
#else
R  SERIAL_ECHOLN("PID Autotune start"); // All nodes in this
    block get Remove because we apply the Keep intention above!
#endif
```

Outcome:

```
SERIAL_ECHOLN(MSG_PID_AUTOTUNE_START);
```

Exclusive

Code taken from `373f3ec/Marlin_main.cpp`.

Case I: Mutually exclusive code

Expected outcome: Integrate Delta-specific changes from fork as features, and make non-delta-compatible code features.

Integrated view

```
#if !defined(FORK)
  // Do not use feedmultiply for E or Z only moves
  if( (current_position[X_AXIS] == destination [X_AXIS]) &&
      (current_position[Y_AXIS] == destination [Y_AXIS])) {
    plan_buffer_line(destination[X_AXIS], destination[Y_AXIS],
        destination[Z_AXIS], destination[E_AXIS], feedrate/60,
        active_extruder);
  }
#else
  float difference[NUM_AXIS];
  for (int8_t i=0; i < NUM_AXIS; i++) {
    difference[i] = destination[i] - current_position[i];
  }
#endif
```

Mainline view:

```
float difference[NUM_AXIS];
for (int8_t i=0; i < NUM_AXIS; i++) {
  difference[i] = destination[i] - current_position[i];
}
```

Clone view:

```
// Do not use feedmultiply for E or Z only moves
if( (current_position[X_AXIS] == destination [X_AXIS]) &&
    (current_position[Y_AXIS] == destination [Y_AXIS])) {
  plan_buffer_line(destination[X_AXIS], destination[Y_AXIS],
      destination[Z_AXIS], destination[E_AXIS], feedrate/60,
      active_extruder);
}
```

Resolution:

Apply `Keep` on both branches, followed by making each block mutually exclusive under the condition `!defined(DELTA)`. Let `K` denote `Keep` and `Xn` denote `Exclusive`, with `n={1,2}` to denote block.

Integrated view:

```
#if !defined(FORK)
K  // Do not use feedmultiply for E or Z only moves
K  if( (current_position[X_AXIS] == destination [X_AXIS]) &&
    (current_position[Y_AXIS] == destination [Y_AXIS])) {
K    plan_buffer_line(destination[X_AXIS], destination[Y_AXIS],
    destination[Z_AXIS], destination[E_AXIS], feedrate/60,
    active_extruder);
K  }
#else
K  float difference[NUM_AXIS];
K  for (int8_t i=0; i < NUM_AXIS; i++) {
K    difference[i] = destination[i] - current_position[i];
K  }
#endif
```

Intermediary view (after Keep is applied):

```
  // Do not use feedmultiply for E or Z only moves
  if( (current_position[X_AXIS] == destination [X_AXIS]) &&
      (current_position[Y_AXIS] == destination [Y_AXIS])) {
    plan_buffer_line(destination[X_AXIS], destination[Y_AXIS],
        destination[Z_AXIS], destination[E_AXIS], feedrate/60,
        active_extruder);
  }
  float difference[NUM_AXIS];
  for (int8_t i=0; i < NUM_AXIS; i++) {
    difference[i] = destination[i] - current_position[i];
  }
```

Integrated view (apply Exclusive):

```
X1  // Do not use feedmultiply for E or Z only moves
X1  if( (current_position[X_AXIS] == destination [X_AXIS]) &&
    (current_position[Y_AXIS] == destination [Y_AXIS])) {
X1    plan_buffer_line(destination[X_AXIS], destination[Y_AXIS],
    destination[Z_AXIS], destination[E_AXIS], feedrate/60,
    active_extruder);
X1  }
X2  float difference[NUM_AXIS];
X2  for (int8_t i=0; i < NUM_AXIS; i++) {
```

```
X2    difference[i] = destination[i] - current_position[i];
X2  }
```

Outcome:

```
#if !defined(DELTA)
  // Do not use feedmultiply for E or Z only moves
  if( (current_position[X_AXIS] == destination [X_AXIS]) &&
      (current_position[Y_AXIS] == destination [Y_AXIS])) {
    plan_buffer_line(destination[X_AXIS], destination[Y_AXIS],
        destination[Z_AXIS], destination[E_AXIS], feedrate/60,
        active_extruder);
  }
#else
  float difference[NUM_AXIS];
  for (int8_t i=0; i < NUM_AXIS; i++) {
    difference[i] = destination[i] - current_position[i];
  }
#endif
```

A.8 Variant Integration Exit Questionnaire

Exit Questionnaire

Please specify to which extent you agree with the following statements. Thereafter, please answer five open-ended questions. Please carefully elaborate, we are interested in your experiences and your opinion about the concept of intention-based integration, less about the tool UI or usability-related issues that are easy to fix in the INCLINE tool.

1. **Integration with INCLINE is faster than in Eclipse**
   Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

2. **I make fewer mistakes in INCLINE than in Eclipse**
   Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

3. **Mistakes are easier to notice in INCLINE than in Eclipse**
   Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

4. **Mistakes are easier to fix in INCLINE than in Eclipse**
   Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

5. **INCLINE is a mature tool**
   Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

6. **INCLINE is not mature enough**
   Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

7. **Intention-based integration is not complex**
Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

8. **The Keep intention is intuitive**
Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

9. **The Remove intention is intuitive**
Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

10. **The Implicit Keep/Remove intentions are intuitive**
Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

11. **The Keep as Feature intention is intuitive**
Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

12. **The Assign Feature intention is intuitive**
Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

13. **The Exclusive intention is intuitive**
Mark only one oval.

	1	2	3	4	5	
Strongly disagree	◯	◯	◯	◯	◯	Strongly agree

14. **Which intentions did you find the most useful? Please elaborate.**

15. **What are the advantages of using intentions for variant integration compared to manual integration with Eclipse? Please elaborate.**

16. **What are your perceived disadvantages of using intentions for variant integration compared to manual integration with Eclipse? Please elaborate.**

17. **Are there any possible improvements to intention-based integration or in particular INCLINE? Please elaborate.**

18. **How would you prefer to perform a variability-related integration? Please elaborate.**

A.9 Variant Integration Exit Questionnaire Answers

The full answers can be viewed at `https://docs.google.com/spreadsheets/d/1-hoXph_T7p9NvHX55rObh4RG17MECXZoIqLT_Qw5gko/edit?usp=sharing`

Timestamp	Integration with INCLINE	I make fewer mistakes in	Mistakes are easier to not	Mistakes are easier to fix	INCLINE is a mature tool	INCLINE is not mature en
2017-05-02 14.50.57	1	1	1	1	4	2
2017-05-03 09.54.38	3	2	4	4	3	2
2017-05-03 20.03.51	4	3	4	2	1	5
2017-05-04 17.47.17	4	4	5	5	2	5
2017-05-08 14.35.48	4	2	3	4	2	4
2017-05-08 14.36.18	4	4	5	4	2	4
2017-05-08 16.32.08	4	3	4	4	3	3
2017-05-09 11.22.06	4	4	4	4	3	3
2017-05-09 14.08.23	1	2	3	1	4	2
2017-05-09 15.58.42	2	4	4	2	3	3
2017-05-11 21.01.06	4	4	4	3	2	3
2017-05-12 14.30.16	1	1	2	2	3	4
2017-05-15 11.14.30	4	5	4	4	4	2
2017-05-15 18.24.39	3	1	3	2	2	3
2017-05-15 18.24.51	3	5	5	1	3	3
2017-05-16 18.39.11	4	4	5	4	3	

Intention-based integratio	The Keep intention is intu	The Remove intention is i	The Implicit Keep/Remov	The Keep as Feature inte	The Assign Feature intent	The Exclusive intention is
3	5	5	5	5	5	5
3	4	3	3	3	3	2
5	5	5	4	4	3	3
5	4	4	3	3	4	3
4	5	5	5	5	5	3
5	4	5	4	2	2	
4	4	4	3	4	4	3
3	4	4	4	5	5	
4	2	5	2		4	3
2	5	5	4	3	4	3
3	5	5	5	5	5	
1	3	3	2	3	3	
4	4	4	4	4	4	
4	4	4	3	4	3	3
4	4	4	4	4	5	
4	5	5	3	4	4	

Which intentions did you f	What are the advantages	What are your perceived (	Are there any possible im	How would you prefer to p
The Keep and remove we	it instantly adds the metho	I don't have the experienc	I would say add colors, lik	The github desktop tool se
Keep intention, i found it t	I found it faster and more	Manual you could change	Well there were some sma	Since Im not in the so call
Keep intention and keep a	It doesn't require manual (	It's a bit steeper learning (	Easy keybindings for faste	Today I'd probably do it m
Keep and remove	easier to see end result w	confusing to learn a new t	The ui could be more visu	incline, but I've never run
Keep and remove, easy to	Easier to keep track of wh	It might take longer to lea	The function (intentions, fe	I don't really prefer any wa
Keep and remove	All information is kept inta	Different way of thinking	It would be nice with bette	If it was a large scale proj
The assign as a feature a	It gives a better overview	To be able to write directly	Same as above, to be abl	A combination of both wou
The keep intention mostly	It's a much more powerful	Bigger hurdle to get starte	Not that I can think of (exc	I would definitely like usin
Assign feature	Seems faster in a larger s	It was hard to navigate to	The naming of functions w	Incline for longer periods (
Keep and Remove, very i	Easier to keep track of bu	Learning curve, mostly - u	Hotkeys for all the intentio	INCLINE would be nice fo
Impossible to say, becaus	You get the preview of the	Learning curve. And a slig	Not having different intent	INCLINE for any file large
No single one stood out a	Explicit concern given to t	No additional abstraction	Interleaved free-text editin	Three-way diff.
Keep intentions. It makes	It's much more intuitive ar	It's something that I haver	Someone would have to e Sometimes I was a bit los	I think the one with Incline I would be much less wor
Keep as Feature.	It was more clear how it a	Involves a learning curve	Too new to be able to give	Probably the copy and pa
Keep and remove	Less mistakes	Feels like drag and drop,	More keybindings	INCLINE
Keep Intention	You can preform the task	It takes more time.	Keybindings. Being able t	Incline

A.10 Controlled Experiment Answers

Timestamp	Integration with INCLINE i	I make fewer mistakes in	Mistakes are easier to not	Mistakes are easier to fix	INCLINE is a mature tool	INCLINE is not mature en
22/08/2017 13:51:04	5	4	4	3	3	3
22/08/2017 14:50:11	5	5	5	4	4	2
23/08/2017 03:11:35	3	4	4	2	3	4
23/08/2017 03:55:12	3	3	3	4	3	3
23/08/2017 04:17:01	4	2	4	2	3	3
23/08/2017 05:40:13	4	3	2	1	1	5
23/08/2017 09:02:40	3	3	4	2	2	4
24/08/2017 07:39:10	4	5	5	4	4	3
24/08/2017 13:41:09	3	5	5	5	3	3
26/08/2017 08:38:10	4	5	4	4	3	4

Intention-based integratio	The Keep intention is intu	The Remove intention is i	The Implicit Keep/Remov	The Keep as Feature inte	The Exclusive intention is	Which intentions did you f
4	5	5	5	5	4	Keep, remove, feature
4	5	5	4	5	4	feature intention, since its
5	5	5	5	4	4	Keep as feature
4	4	4	4	4	4	In my opinion, all of the in
4	5	5	5	5	5	Keep as Feature intention
4	5	5	4	5	4	
3	4	4	4	4	3	Keep and remove, becaus
5	5	5	5	4	3	Keep as Feature, Implicit
4	5	5	5	5	5	The implicit K/R intentions
5	5	5	4	5	4	Keep and remove

What are the advantages	What are your perceived d	Are there any possible im	How would you prefer to perform a variability-related integration? Please elaborate.
Intentions are much more	Have to be careful not to l	Maybe allow mouse text-s	INCLINE
less typing/manual copy&paste and thereby less err		existing features coud be UI could be polished, espe	as described before, extraction of existing features in mainline and fork would be helpful. E.g., for selectir
Makes it easier to merge t	Makes it easier to merge t	The tool had some bugs, especially when un-doing things.	
Faster integration based c	The only thing is that you	Log the actions and displa	I did not get the question.
It is much time saving than copy-pasting.		Sometimes code-blocks c	Your tool is better than Eclipse. It still seems to have some is issues/bugs.
Editing on the AST level i	Moving code seems to be	I have little experiences w	I would lean towards INCLINE, given that it still allowed me to manually adjust the code (some mistakes a
It's a specialised tool, in c	You might be simply more	Don't know, I don't know t	I'm not sure I understand the question.
Less effort and reduction	None, with maturity of he	Now I can't remember, pe	Through intention based integration. It was much faster for me once I got to know how to use the tool--IN
(1) INCLINE as a tool is b	There is a learning curve to the semantics and the keyboard shortcuts of the intentions and, of course, the tool must be sound. But one it matures I can't see a		
-> Straightforward to deci -> Easy to check whether	Sometimes it would be nic	Support for merging direc	Typically, I integrate variants using IntelliJ. INCLINE would be a nice addition on top.

ıg from existing features when applying a feature intention or for a feature-base

are just quicker to fix by moving code around).		
CLINE		
any disadvantages.		

Bibliography

[mps,] Jetbrains MPS. `http://www.jetbrains.com/mps`.

[Abal et al., 2014] Abal, I., Brabrand, C., and Wasowski, A. (2014). 42 variability bugs in the linux kernel: a qualitative analysis. In *ACM/IEEE International Conference on Automated Software Engineering, ASE, 2014*, pages 421–432.

[Abal et al., 2017] Abal, I., Melo, J., Stănciulescu, Ş., Ribeiro, M., Brabrand, C., and Wąsowski, A. (2017). Variability Bugs in Highly-Configurable Systems: A Qualitative Analysis. TOSEM.

[Antkiewicz et al., 2014] Antkiewicz, M., Ji, W., Berger, T., Czarnecki, K., Schmorleiz, T., Lämmel, R., Stanciulescu, S., Wasowski, A., and Schaefer, I. (2014). Flexible product line engineering with a virtual platform. In *36th International Conference on Software Engineering, ICSE '14, Companion Proceedings*.

[Assunção et al., 2017] Assunção, W. K. G., Lopez-Herrejon, R. E., Linsbauer, L., Vergilio, S. R., and Egyed, A. (2017). Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering*, pages 1–45.

[Atkins et al., 2002] Atkins, D. L., Ball, T., Graves, T. L., and Mockus, A. (2002). Using version control data to evaluate the impact of software tools: A case study of the version editor. *IEEE Transactions on Software Engineering*, 28(7):625–637.

[Babich, 1986] Babich, W. A. (1986). *Software Configuration Management: Coordination for Team Productivity*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Batory et al., 2004] Batory, D., Sarvela, J. N., and Rauschmayer, A. (2004). Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371.

[Bécan, 2016] Bécan, G. (2016). *Metamodels and Feature Models: Complementary Approaches to Formalize Product Comparison Matrices*. PhD thesis, Université Rennes 1.

[Berger et al., 2014a] Berger, T., Nair, D., Rublack, R., Atlee, J. M., Czarnecki, K., and Wąsowski, A. (2014a). Three Cases of Feature-Based Variability Modeling in Industry. In *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, pages 302–319.

[Berger et al., 2013] Berger, T., She, S., Lotufo, R., Wąsowski, A., and Czarnecki, K. (2013). A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Trans. Software Eng.*, 39(12):1611–1640.

[Berger et al., 2014b] Berger, T., Stănciulescu, Ş., Øgård, O., Haugen, Ø., Larsen, B., and Wąsowski, A. (2014b). To connect or not to connect: experiences from modeling topological variability. In *18th International Software Product Line Conference*.

[Berger et al., 2016] Berger, T., Völter, M., Jensen, H. P., Dangprasert, T., and Siegmund, J. (2016). Efficiency of Projectional Editing: A Controlled Experiment. In *24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*.

[Blendinger, 2010] Blendinger, F. (2010). A Filesystem-Based Approach to Support Product Line Development with Editable Views. Master's thesis, Friedrich-Alexander University Erlangen-Nuremberg.

[Christerson and Kolk, 2009] Christerson, M. and Kolk, H. (2009). Domain expert DSLs. talk at QCon London 2009, available at `http://www.infoq.com/presentations/DSL-Magnus-Christerson-Henk-Kolk`.

[Chu-Carroll et al., 2002] Chu-Carroll, M. C., Wright, J., and Shields, D. (2002). Supporting aggregation in fine grained software configuration management. *ACM SIGSOFT Software Engineering Notes*, 27:99.

[Clements and Northrop, 2002] Clements, P. and Northrop, L. (2002). *Software product lines*. Addison-Wesley.

[Cordy, 2003] Cordy, J. R. (2003). Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation. In *11th IEEE International Workshop on Program Comprehension*.

[Czarnecki and Eisenecker, 2000] Czarnecki, K. and Eisenecker, U. W. (2000). *Generative programming - methods, tools and applications*. Addison-Wesley.

[Di Iorio et al., 2009] Di Iorio, A., Schirinzi, M., Vitali, F., and Marchetti, C. (2009). A natural and multi-layered approach to detect changes in tree-based textual documents. In *Proceedings of the 11th International Conference on Enterprise Information Systems*, pages 90–101, Berlin, Heidelberg. Springer.

[Dintzner et al., 2016] Dintzner, N., van Deursen, A., and Pinzger, M. (2016). FEVER: Extracting Feature-oriented Changes from Commits. In *13th International Conference on Mining Software Repositories (MSR)*.

[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 *17th European Conference on Software Maintenance and Reengineering*.

[Duc et al., 2014] Duc, A. N., Mockus, A., Hackbarth, R., and Palframan, J. (2014). Forking and Coordination in Multi-platform Development: A Case Study. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 59–68, New York, NY, USA. ACM.

[Ebert et al., 1998] Ebert, J., Gimnich, R., Stasch, H., and Winter, A. (1998). GUPRO - Generische Umgebung zum Programmverstehen. Koblenzer Schriften zur Informatik. Folbach, Koblenz.

[Erwig and Walkingshaw, 2011a] Erwig, M. and Walkingshaw, E. (2011a). The choice calculus: A representation for software variation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(1):6.

[Erwig and Walkingshaw, 2011b] Erwig, M. and Walkingshaw, E. (2011b). The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 21(1):6:1–6:27.

[Faust and Verhoef, 2003] Faust, D. and Verhoef, C. (2003). Software Product Line Migration and Deployment. In *Software Product Line Migration and Deployment*, volume 33, pages 933–955.

[Favre, 1996] Favre, J. (1996). Preprocessors from an abstract point of view. In *International Conference on Software Maintenance (ICSM)*.

[Fenske et al., 2017] Fenske, W., Meinicke, J., Schulze, S., Schulze, S., and Saake, G. (2017). Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line. In *24th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 316–326. IEEE.

[Fenske and Schulze, 2015] Fenske, W. and Schulze, S. (2015). Code smells revisited: A variability perspective. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*, page 3. ACM.

[Fenske et al., 2015] Fenske, W., Schulze, S., Meyer, D., and Saake, G. (2015). When Code Smells Twice as Much: Metric-Based Detection of Variability-Aware Code

Smells. In *Source Code Analysis and Manipulation (SCAM), 2015 15th IEEE International Working Conference on*, pages 171–180. IEEE.

[Fenske et al., 2014] Fenske, W., Thüm, T., and Saake, G. (2014). A taxonomy of software product line reengineering. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, page 4. ACM.

[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.

[Fischer et al., 2015] Fischer, S., Linsbauer, L., Lopez-Herrejon, R. E., and Egyed, A. (2015). The ECCO tool: Extraction and composition for clone-and-own. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 665–668. IEEE Press.

[Foster et al., 2007] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. (2007). Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3):17.

[Ganesan et al., 2009] Ganesan, D., Lindvall, M., Ackermann, C., McComas, D., and Bartholomew, M. (2009). Verifying Architectural Design Rules of the Flight Software Product Line. In *13th International Software Product Line Conference*.

[Gousios, 2013] Gousios, G. (2013). The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA. IEEE Press.

[Gousios et al., 2014] Gousios, G., Pinzger, M., and van Deursen, A. (2014). An Exploratory Study of the Pull-based Software Development Model. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 345–355.

[Gulla et al., 1991] Gulla, B., Karlsson, E.-A., and Yeh, D. (1991). Change-oriented version descriptions in EPOS. *Software Engineering Journal*, 6(6):378–386.

[Hetrick et al., 2006] Hetrick, W. A., Krueger, C. W., and Moore, J. G. (2006). Incremental return on incremental investment: Engenio's transition to software product line practice. In *OOPSLA'06*.

[Hofer et al., 2010] Hofer, W., Elsner, C., Blendinger, F., Schröder-Preikschat, W., and Lohmann, D. (2010). Toolchain-independent Variant Management with the Leviathan Filesystem. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development (FOSD)*.

[Hunsen et al., 2015] Hunsen, C., Zhang, B., Siegmund, J., Kästner, C., Leßenich, O., Becker, M., and Apel, S. (2015). Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering*.

[Iosif-Lazar et al., 2017] Iosif-Lazar, A. F., Melo, J., Dimovski, A. S., Brabrand, C., and Wasowski, A. (2017). Effective Analysis of C Programs by Rewriting Variability. *Programming*.

[Jang et al., 2012] Jang, J., Agrawal, A., and Brumley, D. (2012). ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 48–62.

[Janzen and De Volder, 2004] Janzen, D. and De Volder, K. (2004). Programming with crosscutting effective views. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP)*.

[Jepsen and Beuche, 2009] Jepsen, H. P. and Beuche, D. (2009). Running a Software Product Line: Standing Still is Going Backwards. In *13th International Software Product Line Conference*.

[Jepsen et al., 2007] Jepsen, H. P., Dall, J. G., and Beuche, D. (2007). Minimally Invasive Migration to Software Product Lines. In *11th International Software Product Line Conference*.

[Johnson and Foote, 1988] Johnson, R. E. and Foote, B. (1988). Designing reusable classes. *Journal of object-oriented programming*, 1(2):22–35.

[Juergens et al., 2009] Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S. (2009). Do code clones matter? In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 485–495. IEEE.

[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.

[Kapser and Godfrey, 2006a] Kapser, C. and Godfrey, M. W. (2006a). 'Cloning Considered Harmful' Considered Harmful. In *13th Working Conference on Reverse Engineering*.

[Kapser and Godfrey, 2006b] Kapser, C. J. and Godfrey, M. W. (2006b). Supporting the Analysis of Clones in Software Systems: Research Articles. *J. Softw. Maint. Evol.*, 18(2):61–82.

[Kästner, 2010] Kästner, C. (2010). *Virtual separation of concerns*. PhD thesis, University of Magdeburg.

[Kästner and Apel, 2009] Kästner, C. and Apel, S. (2009). Virtual separation of concerns-a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78.

[Kästner et al., 2007] Kästner, C., Apel, S., and Batory, D. S. (2007). A Case Study Implementing Features Using AspectJ. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*, pages 223–232.

[Kästner et al., 2008] Kästner, C., Trujillo, S., and Apel, S. (2008). Visualizing Software Product Line Variabilities in Source Code. In *ViSPLE*.

[Kernighan and Ritchie, 1988] Kernighan, B. W. and Ritchie, D. M. (1988). *The C programming language*, volume 78.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. *ECOOP'97—Object-oriented programming*, pages 220–242.

[Kim et al., 2004] Kim, M., Bergman, L., Lau, T., and Notkin, D. (2004). An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, ISESE '04, pages 83–92, Washington, DC, USA. IEEE Computer Society.

[Kim et al., 2005] Kim, M., Sazawal, V., Notkin, D., and Murphy, G. (2005). An Empirical Study of Code Clone Genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196.

[Klatt et al., 2013] Klatt, B., Küster, M., and Krogmann, K. (2013). A graph-based analysis concept to derive a variation point design from product copies. *Proc of REVE*, 13.

[Kolb et al., 2006] Kolb, R., Muthig, D., Patzke, T., and Yamauchi, K. (2006). Refactoring a legacy component for reuse in a software product line: a case study. *Journal of Software Maintenance*, 18(2):109–132.

[Krinke, 2008] Krinke, J. (2008). Is cloned code more stable than non-cloned code? In *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, pages 57–66. IEEE.

[Kruskal, 1984] Kruskal, V. (1984). Managing Multi-Version Programs with an Editor. *IBM Journal of Research and Development*, 28(1):74–81.

[Kruskal, 2000] Kruskal, V. (2000). A blast from the past: Using P-EDIT for multidimensional editing. In *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*.

[Kullbach and Riediger, 2001] Kullbach, B. and Riediger, V. (2001). Folding: an approach to enable program understanding of preprocessed languages. In *Proceedings of 8th Working Conference on Reverse Engineering (WCRE)*.

[Le et al., 2011] Le, D., Walkingshaw, E., and Erwig, M. (2011). #ifdef confirmed harmful: Promoting understandable software variation. In *Proceedings of Symposium on Visual Languages and Human Centric Computing (VL/HCC)*.

[Lie et al., 1989] Lie, A., Conradi, R., Didriksen, T. M., and Karlsson, E.-A. (1989). Change oriented versioning in a software engineering database. *SIGSOFT Softw. Eng. Notes*, 14(7):56–65.

[Liebig, 2015] Liebig, J. (2015). *Analysis and Transformation of Configurable Systems*. PhD thesis, University of Passau, Germany.

[Liebig et al., 2015] Liebig, J., Janker, A., Garbe, F., Apel, S., and Lengauer, C. (2015). Morpheus: Variability-aware Refactoring in the Wild. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15.

[Lillack et al., 2017] Lillack, M., Stănciulescu, Ş., Hedmann, W., Berger, T., and Wąsowski, A. (2017). Intention-Based Integration of Variants. Manuscript under review. Available upon request.

[Lotufo et al., 2010] Lotufo, R., She, S., Berger, T., Czarnecki, K., and Wasowski, A. (2010). Evolution of the linux kernel variability model. In *Proceedings of 14th International Conference, (SPLC)*.

[Martinez et al., 2015] Martinez, J., Ziadi, T., Bissyandé, T. F., Klein, J., and Traon, Y. L. (2015). Bottom-up adoption of software product lines: a generic and extensible approach. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 101–110.

[McVoy, 2015] McVoy, L. (2015). Preliminary Product Line Support in BitKeeper. In *19th International Software Product Line Conference, SPLC*.

[Mebane and Ohta, 2007] Mebane, H. and Ohta, J. (2007). Dynamic Complexity and the Owen Firmware Product Line Program. In *11th International Software Product Line Conference*.

[Melo et al., 2016] Melo, J., Brabrand, C., and Wąsowski, A. (2016). How Does the Degree of Variability Affect Bug-Finding? In *International Conference on Software Engineering (ICSE)*.

[Mens, 2002] Mens, T. (2002). A State-of-the-Art Survey on Software Merging. *IEEE Trans. Software Eng.*, 28(5):449–462.

[Mikkonen and Nyman, 2011] Mikkonen, T. and Nyman, L. (2011). To Fork or Not to Fork: Fork Motivations in SourceForge Projects. *Int. J. Open Source Softw. Process.*, 3(3):1–9.

[Montalvillo and Díaz, 2015] Montalvillo, L. and Díaz, O. (2015). Tuning GitHub for SPL development: branching models & repository operations for product engineers. In *Proceedings of the 19th International Conference on Software Product Line*, pages 111–120. ACM.

[Moura and Bjørner, 2008] Moura, L. D. and Bjørner, N. (2008). Z3: An efficient smt solver. In *Proc. Int'l Conf. on Tools and algorithms for the construction and analysis of systems (TACAS/ETAPS)*, pages 337–340, Berlin, Heidelberg. Springer.

[Munch, 1993] Munch, B. P. (1993). *Versioning in a Software Engineering Database — the Change Oriented Way*. PhD thesis, Norwegian Institute of Technology, Division of Computer Systems and Telematics.

[Nguyen et al., 2012] Nguyen, H. A., Nguyen, T. T., Pham, N. H., Al-Kofahi, J., and Nguyen, T. N. (2012). Clone management for evolving software. *IEEE transactions on software engineering*, 38(5):1008–1026.

[Passos et al., 2015] Passos, L., Teixeira, L., Nicolas, D., Apel, S., Wąsowski, A., Czarnecki, K., Borba, P., and Guo, J. (2015). Coevolution of Variability Models and Related Software Artifacts: A Fresh Look at Evolution Patterns in the Linux Kernel. *Empirical Software Engineering, Springer*.

[Pech et al., 2009] Pech, D., Knodel, J., Carbon, R., Schitter, C., and Hein, D. (2009). Variability Management in Small Development Organizations: Experiences and Lessons Learned from a Case Study. In *13th International Software Product Line Conference*, SPLC '09.

[Pfofe et al., 2016] Pfofe, T., Thüm, T., Schulze, S., Fenske, W., and Schaefer, I. (2016). Synchronizing Software Variants with Variantsync. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 329–332, New York, NY, USA. ACM.

[Pohl et al., 2005] Pohl, K., Böckle, G., and Linden, F. J. v. d. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Prehofer, 1997] Prehofer, C. (1997). Feature-oriented programming: A fresh look at objects. *ECOOP'97—Object-Oriented Programming*, pages 419–443.

[Rabiser et al., 2016] Rabiser, D., Grünbacher, P., Prähofer, H., and Angerer, F. (2016). A Prototype-based Approach for Managing Clones in Clone-and-own Product Lines. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 35–44, New York, NY, USA. ACM.

[Raymond, 1999] Raymond, E. S. (1999). Homesteading the Noosphere. In O'Reilly, T., editor, *The Cathedral and the Bazaar*. O'Reilly & Associates, Inc.

[Robles and González-Barahona, 2012] Robles, G. and González-Barahona, J. M. (2012). A Comprehensive Study of Software Forks: Dates, Reasons and Outcomes. In *Open Source Systems: Long-Term Sustainability - 8th IFIP WG 2.13 International Conference, OSS 2012, Hammamet, Tunisia, September 10-13, 2012. Proceedings*, pages 1–14.

[Roy and Cordy, 2007] Roy, C. K. and Cordy, J. R. (2007). A Survey on Software Clone Detection Research. *Technical Report No. 2007-54, School of Computing, Queen's University, Kingston Canada*, 115.

[Rubin and Chechik, 2013a] Rubin, J. and Chechik, M. (2013a). A Framework for Managing Cloned Product Variants. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1233–1236, Piscataway, NJ, USA. IEEE Press.

[Rubin and Chechik, 2013b] Rubin, J. and Chechik, M. (2013b). N-way model merging. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 301–311. ACM.

[Rubin et al., 2013] Rubin, J., Czarnecki, K., and Chechik, M. (2013). Managing Cloned Variants: A Framework and Experience. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 101–110, New York, NY, USA. ACM.

[Rubin et al., 2015] Rubin, J., Czarnecki, K., and Chechik, M. (2015). Cloned product variants: from ad-hoc to managed software product lines. *STTT*, 17(5):627–646.

[Schmorleiz, 2015] Schmorleiz, T. (2015). An Annotation-centric Approach to Similarity Management. Master's thesis, University of Koblenz Landau, Germany.

[Schmorleiz and Lämmel, 2016] Schmorleiz, T. and Lämmel, R. (2016). Similarity management of'cloned and owned'variants. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1466–1471. ACM.

[Schulze et al., 2012] Schulze, S., Thüm, T., Kuhlemann, M., and Saake, G. (2012). Variant-preserving Refactoring in Feature-oriented Software Product Lines. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12.

[Schwarz et al., 2012] Schwarz, N., Lungu, M., and Robbes, R. (2012). On how often code is cloned across repositories. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 1289–1292.

[Schwägerl et al., 2015] Schwägerl, F., Buchmann, T., and Westfechtel, B. (2015). SuperMod — A Model-Driven Tool that Combines Version Control and Software Product Line Engineering. In *Proceedings of the 10th International Conference on Software Paradigm Trends*, pages 5–18.

[She, 2013] She, S. (2013). *Feature model synthesis*. PhD thesis, University of Waterloo.

[She et al., 2010] She, S., Lotufo, R., Berger, T., Wasowski, A., and Czarnecki, K. (2010). The Variability Model of The Linux Kernel. In *4th International Workshop on Variability Modelling of Software-intensive Systems*.

[Singh et al., 2007] Singh, N., Gibbs, C., and Coady, Y. (2007). C-CLR: A Tool for Navigating Highly Configurable System Software. In *Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*.

[Spencer and Geoff, 1992] Spencer, H. and Geoff, C. (1992). #ifdef Considered Harmful, or Portability Experience With C News. In *USENIX Summer Technical Conference*, pages 185–198.

[Stănciulescu et al., 2016a] Stănciulescu, Ş., Berger, T., Walkingshaw, E., and Wąsowski, A. (2016a). Concepts, Operations and Feasibility of a Projection-Based Variation Control Systems. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*, ICSME'16.

[Stănciulescu et al., 2016b] Stănciulescu, Ş., Rabiser, D., and Seidl, C. (2016b). A Technology-Neutral Role-Based Collaboration Model for Software Ecosystems. In *Proceedings of the 7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, ISoLA'16.

[Stănciulescu et al., 2015] Stănciulescu, Ş., Schulze, S., and Wąsowski, A. (2015). Forked and Integrated Variants in an Open-Source Firmware Project. In *31st International Conference on Software Maintenance and Evolution (ICSME)*.

[Tichy, 1985] Tichy, W. F. (1985). RCS — A System for Version Control. 7(July 1985):637–654.

[Voelter et al., 2014] Voelter, M., Siegmund, J., Berger, T., and Kolb, B. (2014). Towards User-Friendly Projectional Editors. In *7th International Conference on Software Language Engineering (SLE)*.

[Völter et al., 2014] Völter, M., Siegmund, J., Berger, T., and Kolb, B. (2014). Towards User-Friendly Projectional Editors. In *SLE*.

[Walkingshaw, 2013] Walkingshaw, E. (2013). *The Choice Calculus: A Formal Language of Variation*. PhD thesis, Oregon State University, USA.

[Walkingshaw and Erwig, 2012] Walkingshaw, E. and Erwig, M. (2012). A Calculus for Modeling and Implementing Variation. *SIGPLAN Not.*, 48(3):132–140.

[Walkingshaw and Ostermann, 2014] Walkingshaw, E. and Ostermann, K. (2014). Projectional editing of variational software. In *Generative Programming: Concepts and Experiences (GPCE)*.

[Westfechtel et al., 2001] Westfechtel, B., Munch, B. P., and Conradi, R. (2001). A Layered Architecture for Uniform Version Management. *IEEE TSE*, 27(12):1111–1133.

[Zhang et al., 2013] Zhang, B., Becker, M., Patzke, T., Sierszecki, K., and Savolainen, J. E. (2013). Variability Evolution and Erosion in Industrial Product Lines: A Case Study. In *17th International Software Product Line Conference*.

[Zhou et al., 2018] Zhou, S., Stănciulescu, Ş., Leßenich, O., Xiong, Y., Kästner, C., and Wąsowski, A. (2018). INFOX: Identifying Features from Forks. In *Manuscript under review*.

[Ziadi et al., 2014] Ziadi, T., Henard, C., Papadakis, M., Ziane, M., and Le Traon, Y. (2014). Towards a Language-independent Approach for Reverse-engineering of Software Product Lines. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1064–1071, New York, NY, USA. ACM.