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

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

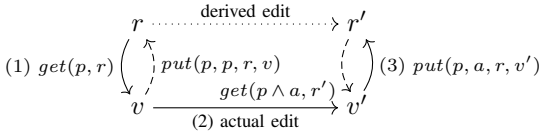

Fig. 2. Projection-based variational editing workflow and relationships. Symbol r represents the repository that contains source code, p is the projection that specifies how to obtain the view v from r using the get function. The ambition a specifies how should the changes from the edited view v' be 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 $(A \wedge B)\langle \texttt{2}, \texttt{3}\rangle$, 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.

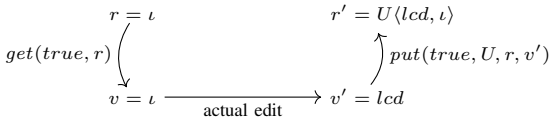

Fig. 4. P1 AddIfdef editing workflow.

*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.

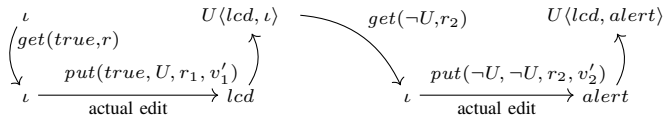

Fig. 5. P3 AddIfdefElse editing workflow.

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`.

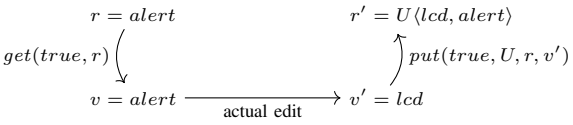

Fig. 6. P4 AddIfdefWrapElse editing workflow.

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.

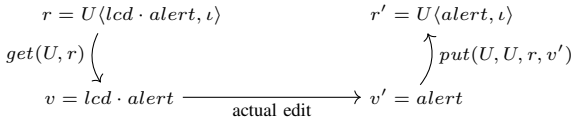

Fig. 7. P8 RemNormalCode editing workflow.

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.

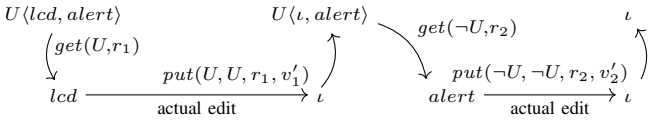

Fig. 8. P9 RemIfdef editing workflow.

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`).

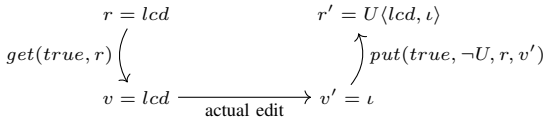

Fig. 9. P11 WrapCode editing workflow.

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	$F\langle e_1, e_2 \cdot e_3\rangle \to F\langle e_1 \cdot e_2, e_3\rangle$

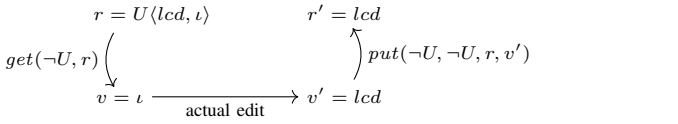

Fig. 10. P12 UnwrapCode editing workflow.

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.

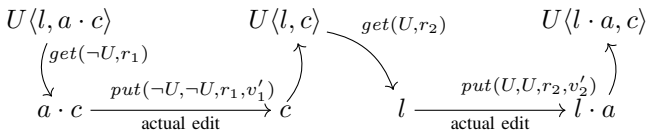

Fig. 11. P14 MoveElse editing workflow.

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

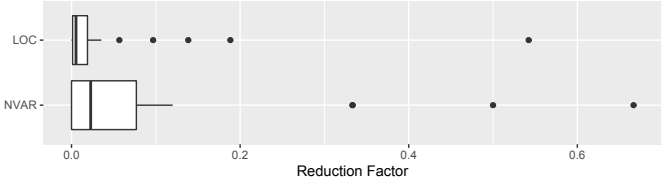

Fig. 12. Reduction factors for LOC and NVAR for the view

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.

References

[1] Jetbrains MPS. http://www.jetbrains.com/mps.

[2] I. Abal, C. Brabrand, and A. Wasowski. 42 variability bugs in the linux kernel: a qualitative analysis. In *ACM/IEEE International Conference on Automated Software Engineering, ASE, 2014*, pages 421–432, 2014.

[3] D. L. Atkins, T. Ball, T. L. Graves, and A. Mockus. Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor. *IEEE Trans. on Software Engineering*, 28(7):625–637, 2002.

[4] 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.

[5] M. Christerson and H. Kolk. Domain expert DSLs, 2009. talk at QCon London 2009, available at http://www.infoq.com/presentations/DSL-Magnus-Christerson-Henk-Kolk.

[6] M. C. Chu-Carroll, J. Wright, and D. Shields. Supporting aggregation in fine grained software configuration management. *ACM SIGSOFT Software Engineering Notes*, 27:99, 2002.

[7] N. Dintzner, A. van Deursen, and M. Pinzger. Fever: Extracting feature-oriented changes from commits. In *13th International Conference on Mining Software Repositories (MSR)*, 2016.

[8] J. Ebert, R. Gimnich, H. Stasch, and A. Winter. Gupro - generische umgebung zum programmverstehen. Koblenzer Schriften zur Informatik. Folbach, Koblenz, 1998.

[9] N. Eén and N. Sörensson. *An Extensible SAT-solver*. 2004.

[10] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 21(1):6:1–6:27, 2011.

[11] M. Erwig and E. Walkingshaw. *Generative and Transformational Techniques in Software Engineering IV: International Summer School, GTTSE 2011*, chapter Variation Programming with the Choice Calculus, pages 55–100. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[12] J. Favre. Preprocessors from an abstract point of view. In *International Conference on Software Maintenance (ICSM)*, 1996.

[13] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem. *ACM Trans. on Programming Languages and Systems*, 29(3), 2007.

[14] B. Gulla, E.-A. Karlsson, and D. Yeh. Change-oriented version descriptions in EPOS. *Software Engineering Journal*, 6(6):378–386, 1991.

[15] W. Hofer, C. Elsner, F. Blendinger, W. Schröder-Preikschat, and D. Lohmann. Toolchain-independent variant management with the leviathan filesystem. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development (FOSD)*, 2010.

[16] D. Janzen and K. De Volder. Programming with crosscutting effective views. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP)*. 2004.

[17] W. Ji, T. Berger, M. Antkiewicz, and K. Czarnecki. Maintaining feature traceability with embedded annotations. In *Proc. SPLC*, 2015.

[18] C. Kästner, S. Trujillo, and S. Apel. Visualizing software product line variabilities in source code. In *ViSPLE*, 2008.

[19] B. W. Kernighan and D. M. Ritchie. *The C programming language*, volume 78. 1988.

[20] V. Kruskal. Managing multi-version programs with an editor. *IBM Journal of Research and Development*, 28(1):74–81, Jan 1984.

[21] V. Kruskal. A blast from the past: Using p-edit for multidimensional editing. In *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, 2000.

[22] B. Kullbach and V. Riediger. Folding: an approach to enable program understanding of preprocessed languages. In *Proceedings of 8th Working Conference on Reverse Engineering (WCRE)*, 2001.

[23] D. Le, E. Walkingshaw, and M. Erwig. #ifdef confirmed harmful: Promoting understandable software variation. In *Proceedings of Symposium on Visual Languages and Human Centric Computing (VL/HCC)*, 2011.

[24] A. Lie, R. Conradi, T. M. Didriksen, and E.-A. Karlsson. Change oriented versioning in a software engineering database. *SIGSOFT Softw. Eng. Notes*, 14(7):56–65, 1989.

[25] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski. Evolution of the linux kernel variability model. In *Proceedings of 14th International Conference, (SPLC)*, 2010.

[26] J. Melo, C. Brabrand, and A. Wąsowski. How Does the Degree of Variability Affect Bug-Finding? In *International Conference on Software Engineering (ICSE)*, 2016.

[27] B. P. Munch. *Versioning in a Software Engineering Database — the Change Oriented Way*. PhD thesis, Norwegian Institute of Technology, Division of Computer Systems and Telematics, 1993.

[28] L. Passos, L. Teixeira, D. Nicolas, S. Apel, A. Wąsowski, K. Czarnecki, P. Borba, and J. Guo. Coevolution of variability models and related software artifacts: A fresh look at evolution patterns in the linux kernel. *Empirical Software Engineering, Springer*, 2015.

[29] F. Schwägerl, T. Buchmann, and B. Westfechtel. 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, 2015.

[30] C. Simonyi, M. Christerson, and S. Clifford. Intentional Software. In *Proceedings of OOPSLA*, 2006.

[31] N. Singh, C. Gibbs, and Y. Coady. 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)*, 2007.

[32] H. Spencer and C. Geoff. #ifdef Considered Harmful, or Portability Experience With C News. In *USENIX Summer Technical Conference*, pages 185–198, 1992.

[33] W. F. Tichy. RCS — A System for Version Control. 7(July 1985):637–654, 1985.

[34] Ş. 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.

[35] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. Towards user-friendly projectional editors. In *7th International Conference on Software Language Engineering (SLE)*, 2014.

[36] E. Walkingshaw. *The Choice Calculus: A Formal Language of Variation*. PhD thesis, Oregon State University, USA, 2013.

[37] E. Walkingshaw and M. Erwig. A calculus for modeling and implementing variation. *SIGPLAN Not.*, 48(3):132–140, Sept. 2012.

[38] E. Walkingshaw and K. Ostermann. Projectional editing of variational software. In *Generative Programming: Concepts and Experiences (GPCE)*, 2014.

[39] B. Westfechtel, B. P. Munch, and R. Conradi. A layered architecture for uniform version management. *IEEE TSE*, 27(12):1111–1133, Dec. 2001.