A Classification of Variation Control Systems

Lukas Linsbauer
CD Lab MEVSS
Johannes Kepler University Linz
Austria

Thorsten Berger
Chalmers | University of Gothenburg
Sweden

Paul Grünbacher
CD Lab MEVSS
Johannes Kepler University Linz
Austria

Abstract

Version control systems are an integral part of today's software and systems development processes. They facilitate the management of *revisions* (sequential versions) and *variants* (concurrent versions) of a system under development and enable collaboration between developers. Revisions are commonly maintained either per file or for the whole system. Variants are supported via branching or forking mechanisms that conceptually clone the whole system under development. It is known that such cloning practices come with disadvantages. In fact, while short-lived branches for isolated development of new functionality (a.k.a. feature branches) are well supported, dealing with long-term and fine-grained system variants currently requires employing additional mechanisms, such as preprocessors, build systems or custom configuration tools. Interestingly, the literature describes a number of *variation* control systems, which provide a richer set of capabilities for handling fine-grained system variants compared to the version control systems widely used today. In this paper we present a classification and comparison of selected variation control systems to get an understanding of their capabilities and the advantages they can offer. We discuss problems of variation control systems, which may explain their comparably low popularity. We also propose research activities we regard as important to change this situation.

CCS Concepts • **Software and its engineering → Software configuration management and version control systems**;

Keywords Variability management, software product lines, configuration management, software repositories

ACM Reference Format:
Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *Proceedings of GPCE'17*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3136040.3136054

GPCE'17, October 23–24, 2017, Vancouver, BC, Canada

ACM ISBN 978-1-4503-5524-7/17/10…$15.00
https://doi.org/10.1145/3136040.3136054

1 Introduction

Managing sequential and concurrent versions of software artifacts of different types is a constant challenge in software engineering. Version control systems, such as Subversion or Git, are widely used for this purpose. They support handling sequential *revisions* at the file or system level. Concurrent versions—a.k.a. *variants*—of software systems are supported by cloning entire systems or creating branches for specific development tasks. However, variants are currently not handled at the levels of files or features. This impedes flexibility, since only a fixed set of variants may exist. Developers thus need to use mechanisms and tools like preprocessors, build systems such as *Make*, or even custom configuration solutions as for example in the Linux kernel in addition to version control systems. Such configurable systems (a.k.a. software product lines [3]) manage variants in terms of *features*—configuration options that are mapped to variation points (e.g., preprocessor directives), which allows deriving individual variants by selecting the desired features. It has been pointed out, however, that managing variation points manually is cumbersome and error-prone [19, 42, 56].

We consider variation control systems (VarCS) that aim at overcoming these limitations. A VarCS supports creating and editing views of development artifacts for specific system variants based on features. It thus reduces the complexity of changing variants and frees developers from manually maintaining variation points. Existing work on VarCS is not limited to product line research. We found that different research communities have been developing a wide range of VarCS solutions with capabilities for handling long-term and fine-grained system variants. Often, these systems are not widely known and there is also only little evidence demonstrating their success in real-world scenarios. Obviously, the terminology used to describe these approaches is often not consistent across the different research communities, which challenges their comparison.

We present a classification of existing VarCS to get an understanding of their capabilities and the advantages they offer. We further reveal possible reasons why VarCS never became as widely used as version control systems and what kind of research needs to be done to change this situation. Our paper is intended for both researchers and practitioners

interested in what shortcomings prevented the success of VarCS and what needs to be added to make them successful.

2 Background and Related Work

Software changes over time and often needs to exist in multiple variants to address varying stakeholder requirements, such as different hardware, functionality, or energy consumption. *Versions* represent states of evolving software artifacts that are put under version control. Versions are created with different intents: a version intended to supersede its predecessor is commonly called *revision*, while versions intended to coexist are called *variants* [13]. Two research communities have developed a wide range of approaches to handle software revisions and variants.

2.1 Software Configuration Management

The research community of *software configuration management (SCM)* distinguishes between extensional and intensional versioning [13]: extensional versioning means that previously constructed versions can be retrieved, e.g., by a unique number. This means that all versions are explicit and have been checked in once before. Intensional versioning is used when consistent versions of large spaces are created automatically in a flexible manner, that is, new combinations may be constructed on demand. The SCM community has mainly focused on managing the evolution of software artifacts by tracking revisions, largely sidestepping variant support [40, 41]. Although the community recognizes the need [62], the actual variant-management support is still limited in contemporary SCM tools. Variants are supported per system (i.e., per customer) via branching or forking mechanisms that conceptually clone the whole system. However, while such ad hoc management of variants using clone-and-own [18] is simple and cheap, it does not scale with the number of variants. Existing tools lack support for handling variants at the level of files or features. This limits their flexibility, as only a fixed set of variants can exist.

2.2 Software Variability Management

The need for managing variants has been recognized a long time ago in research on program families [48], later leading to the field of *software product line engineering (SPLE)* [12, 16], which provides methods and tools to effectively manage portfolios of system variants. In SPLE, variants are no longer managed at the level of customers, but at the level of features [3, 8, 27] with integrated and configurable platforms. These platforms rely on variability mechanisms, such as conditional compilation and configurable build systems, and allow deriving new variants by selecting desired features. This is achieved by mapping features to variation points, which are commonly realized using annotations such as conditional-compilation directives (e.g., `#ifdef`s). However, in complex systems, such annotations significantly clutter

```
1  uint8_t sort_order[SDSORT_LIMIT];
2
3  // Cache filenames to speed up SD menus.
4  #if SDSORT_USES_RAM
5
6    // If using dynamic RAM for names, allocate on heap
7    #if SDSORT_CACHE_NAMES
8      char sortshort[SDSORT_LIMIT][FILENAME_LENGTH];
9      char sortnames[SDSORT_LIMIT][FILENAME_LENGTH];
10   #elif !SDSORT_USES_STACK
11     char sortnames[SDSORT_LIMIT][FILENAME_LENGTH];
12   #endif
13
14   #if HAS_FOLDER_SORTING && SDSORT_CACHE_NAMES
15     uint8_t isDir[(SDSORT_LIMIT+7)>>3];
16   #endif
17
18 #endif // SDSORT_USES_RAM
```

Listing 1. Marlin code excerpt (slightly adapted from its original).

source code. This challenges program comprehension when many variants of the system need to be edited at the same time [42]. Listing 1 shows a code excerpt of the 3D-printer firmware Marlin, which exists in many variants. The excerpt shows code related to file and folder sorting on SD cards (explained shortly). Editing the code becomes difficult for developers who are only interested in one feature. Approaches such as C-CLR [54], CIDE [28], PEOPL [7], conditional SDGs [2] or GUPRO [31] can filter out irrelevant code, but do not consider the integrated handling of revisions and variants, leaving it to the developers to edit variation points.

Notably, SCM researchers also recognized the need for variant management, including the handling of variants at the level of user-facing, high-level properties (i.e., *features* in product-line terminology). For instance, Gulla et al. [24] emphasize that "selection based on property" is beneficial for non-developers (e.g., testers, customers, and sales experts). In fact, some systems addressing variant-management needs have been developed by SCM researchers, but without finding widespread adoption. Yet, recognizing increasing interest, these systems [25, 36, 60], referred to as VarCS in the remainder, and their concepts deserve a fresh look.

2.3 Variation Control Systems

We therefore study VarCS that can manage features, variants, and variation points in an integrated and uniform manner. VarCS ease or even eliminate the need to directly edit variation points, such as C preprocessor directives. VarCS allow working on one or multiple variants by providing views (or projections) that filter irrelevant details of configurable artifacts to facilitate their comprehension and to lower the cognitive complexity of editing them. Without tool support, a developer would be presented with all the code and presence conditions at once (see Listing 1), including those irrelevant to a given task, and also have to explicitly add or edit the presence conditions manually in the code. A VarCS would, for example, allow a developer to specify its intention once (e.g., adding a new feature to a new variant), and not only will all the irrelevant code for that intention be hidden, also new presence conditions will automatically be added and

existing ones updated in accordance with the specified intention. As such, VarCS aim at supporting the evolution and maintenance of systems with many variants, including software product lines and highly configurable systems [57].

2.4 Illustrative Example

To illustrate the use of a VarCS, we discuss the evolution of the code excerpt shown in Listing 1, which was taken from Marlin (slightly adapted from its original).[1] The developer wanted to add support for dynamic memory allocation (feature SDSORT_DYNAMIC_RAM) when sorting files and folders on SD cards. The final result is shown in Listing 4. Instead of manually editing the source code, let us assume that the developer uses a VarCS, such as one of our subjects, VTS (cf. Section 4.4) [57].

To focus only on the relevant code, the developer chooses in VTS to see only variants containing the features SDSORT_USES_RAM and SDSORT_CACHE_NAMES, since she knows based on her domain knowledge that the new feature SDSORT_DYNAMIC_RAM can only be realized in these variants. So she defines the partial configuration SDSORT_USES_RAM ∧ SDSORT_CACHE_NAMES to check out the relevant code from VTS, which provides her with the code in Listing 2.

```
1  uint8_t sort_order[SDSORT_LIMIT];
2
3  // Cache filenames to speed up SD menus.
4
5    // If using dynamic RAM for names, allocate on heap
6      char sortshort[SDSORT_LIMIT][FILENAME_LENGTH];
7      char sortnames[SDSORT_LIMIT][FILENAME_LENGTH];
8
9    #if HAS_FOLDER_SORTING
10     uint8_t isDir[(SDSORT_LIMIT+7)>>3];
11   #endif
```

Listing 2. A subset of the variants from Listing 1.

The code that is non-optional in these variants (lines 6–7) is shown without `#if` annotations, and the code that does explicitly not belong to these variants is hidden. So when editing, she can be sure that any modification of the code subset can never interact with the hidden code. Code that is optional in these variants (lines 9–11) remains wrapped by `#if` (yet, the expression is simpler in these variants).

The developer now changes lines 1, 6 and 7 to implement the new feature and obtains Listing 3.

```
1  uint8_t *sort_order;
2
3  // Cache filenames to speed up SD menus.
4
5    // If using dynamic RAM for names, allocate on heap
6        char **sortshort, **sortnames;
7
8    #if HAS_FOLDER_SORTING
9      uint8_t isDir[(SDSORT_LIMIT+7)>>3];
10   #endif
```

Listing 3. Modification of Listing 2 for implementing a new feature.

[1] https://github.com/MarlinFirmware/Marlin/commit/47f9883

```
1  // By default the sort index is static
2  #if SDSORT_DYNAMIC_RAM
3    uint8_t *sort_order;
4  #else
5    uint8_t sort_order[SDSORT_LIMIT];
6  #endif
7
8  // Cache filenames to speed up SD menus.
9  #if SDSORT_USES_RAM
10
11   // If using dynamic RAM for names, allocate on heap
12   #if SDSORT_CACHE_NAMES
13     #if SDSORT_DYNAMIC_RAM
14       char **sortshort, **sortnames;
15     #else
16       char sortshort[SDSORT_LIMIT][FILENAME_LENGTH];
17       char sortnames[SDSORT_LIMIT][FILENAME_LENGTH];
18     #endif
19   #elif SDSORT_USES_STACK
20     char sortnames[SDSORT_LIMIT][FILENAME_LENGTH];
21   #endif
22
23   #if !SDSORT_CACHE_NAMES && HAS_FOLDER_SORTING )
24     uint8_t isDir[(SDSORT_LIMIT+7)>>3];
25   #endif
26
27 #endif // SDSORT_USES_RAM
```

Listing 4. Marlin code excerpt after implementation of the new feature SDSORT_DYNAMIC_RAM.

Finally, she specifies that the changes belong to the new feature SDSORT_DYNAMIC_RAM and commits the code. VTS handles all variation points and generates the new internal representation containing all variants that is shown in Listing 4. Especially observe how the new feature code is realized as an alternative (lines 2–6 and 13–14) to the previous code, which is effective when SDSORT_DYNAMIC_RAM is not enabled. Details about the exact mechanism are provided by Stanciulescu et al [57]. All VarCS follow roughly this workflow, but differ in important characteristics we discuss in the remainder.

3 Research Methodology

Our research was carried out in four steps.

Literature Study. Based on our experience in software product lines and software configuration management, we started our search with existing survey papers [13, 21, 40, 58] and then used snowballing [11] to identify further relevant papers. We investigated the reference lists of the papers to identify possibly yet undiscovered papers we considered as relevant for our context. This was useful as some of the subject systems are only published in papers—some older than 30 years—and not accessible via today's digital libraries, making a systematic literature review or mapping study infeasible.

Selection of Subject Systems. We studied the collected subject systems and selected six systems for our comparison. Following our definition of VarCS we included systems capable of handling variants and variation points to hide the complexity of low-level variant management from the user. In several cases, the literature study revealed several generations of subject systems, for instance, when a more advanced system was developed based on a concept or early

prototype. In these cases, we chose the most mature generation for comparison. We excluded systems just providing visualizations for variation points (e.g., `#ifdef`s) or better editor support (e.g., quick fixes, intentions, refactorings) for manipulating variation points in software product lines. We also excluded version control systems such as CVS, Subversion, or Git as they do not allow to deal with concurrent variants at the level of features. We furthermore excluded several subject systems for which the publications were no longer accessible or their description was too superficial for the purpose of our comparison.

Definitions of Characteristics for Comparison. We started this task by identifying a core set of characteristics based on our experience as researchers in the subject matter. We also consulted existing surveys [13, 58] and iteratively refined our set of characteristics when reviewing and structuring the subject systems' capabilities. This also included harmonising the terminology, which was necessary as the papers are from different research communities. This process was also inspired by existing approaches on developing taxonomies (e.g., [46]). The main goal was to identify key characteristics for comparing VarCS, particularly considering their practical adoption.

Classification of Subject Systems. All authors then individually assessed the subject systems using the characteristics as a guideline. The individual classifications were then consolidated in a common classification. The authors carefully discussed all cases of disagreement to achieve consensus on the reported results. Often this required looking for additional background literature not found in the first round.

4 Subject Systems

We now introduce the VarCS selected for our study.

4.1 P-EDIT Editor

P-EDIT was presented in 1984 as a line-based editor for the VM/370 operating system [30] to facilitate the development of multi-version programs by supporting both 'sequences of versions' and 'concurrent versions'. It allows working with one or multiple lines using commands such as LOCATE, CHANGE, NEXT, UP, and INSERT. Line editors were originally developed for systems only providing a keyboard and a printer, but continued to exist for early screen-based systems due to their low memory footprint. P-EDIT used the screen to show the lines surrounding the currently edited line. The prime motivation for developing P-EDIT was the cluttering of source code with preprocessor directives (e.g., `#if`, `#ifdef`, `#ifndef`) [29]. The author also motivates the approach by arguing that representing variants (called concurrent versions) as deltas [49], i.e., instructions on how to modify a previous version, is not sufficient, as each variant would need to be represented by a tree of deltas, leading to substantial redundancy between variants. P-EDIT was implemented by Kruskal and Kosinski, both researchers at the IBM Watson Research Center. A Boolean formula simplifier from another IBM researcher, Sheridan, was incorporated. The tool is not available anymore, but had a small community of users at IBM. Apparently, an evaluation in a larger development project was planned, but we did not find any report about it. The author later described the concepts of P-EDIT at a workshop on virtual separation of concerns [29].

4.2 EPOS Version Control System

Change-oriented versioning (CoV) focuses on managing 'logical changes' of assets instead of managing whole versions of assets or systems [24, 43, 45]. The approach follows the intensional versioning paradigm (cf. Section 2.1) unlike most of today's version-control systems. According to Munch [43], who provided the first implementation, the concept of CoV was first described by Holager [26] in 1988 in a technical report, followed by Lie et al. [32, 33], who extended the concept and implemented CoV using a database, which became the basis for Munch's prototype. The EPOS[2] (Expert System for Program "og" System Development) prototype was later extended and evaluated on the GNU C compiler, which uses conditional-compilation directives to represent its many variants. It later became part of the EPOS Configuration Management (ECM) framework [44], where it was again extended (e.g., with cooperation support [61]) and combined with process modeling and process execution techniques.

CoV follows the typical checkout/commit workflow of version control systems, but instead of checking out revisions of the system (or individual files), the user specifies a configuration to checkout, internally handled by a configurator, similar to configuration-based product derivation in software product lines.

4.3 Leviathan File System

Software developers are using a wide range of development tools that are in most cases not variability-aware and cannot deal with variability mechanisms. Leviathan [25] addresses this problem by providing VarCS support at the level of file systems. Specifically, the authors present an implementation of variant views in specific Leviathan file systems, which can be mounted by developers working on specific variants. Besides activating and deactivating features developers can also express partial configurations by setting features as undecided. This approach allows the use of arbitrary tools without support for specific variability mechanisms, e.g., when debugging or maintaining different program variants. A typical development workflow in Leviathan is to specify a desired variant, to mount the Leviathan file system representing the variant, to modify the variant code, and to save the changes in the editor. The approach also supports automatically writing back changes to the configurable code base

[2] http://www.idi.ntnu.no/~epos/EPOS.html

after editing variant views if certain assumptions are satisfied. Otherwise developers need to double check if Leviathan applied the changes correctly. The approach has further limitations: for instance, it does not allow to change the inclusion condition of a conditional block in a mounted view. Further, Leviathan's CPP only considers constructs for conditional compilation, and cannot deal with expressions containing macros.

4.4 VTS Command-Line Tool

The variation tracking system (VTS) was developed as a prototype for evaluating various concepts of existing VarCS [57]. Specifically, it extends an approach called projectional editing of variational software [60] (not to be confused with the projectional-editing paradigm [10] for direct AST editing, a.k.a. structured editing or syntax-directed editing). Like its conceptual predecessor [60], the VTS approach is formalized using choice calculus, a formal representation of variation points, similar to conditional-compilation directives.

VTS is realized as a command-line tool and realizes a workflow known from version-control systems with a checkout/commit cycle similar to CoV. The prototype[3] can handle individual text files that use C conditional-compilation directives. It allows creating views based on an expression (similar to the choice expression of CoV), which are then edited and committed back to the original file based on another expression (similar to the ambition expression of CoV). The prototype has been evaluated by replaying parts of the history of the Marlin 3D-printer firmware, showing that the tool's capabilities are sufficient to handle a complex real-world evolution process. At the same time the evaluation revealed certain evolution scenarios that require multiple checkout/commit cycles in VTS (cf. Section 6).

4.5 ECCO Version Control System

ECCO[4] realizes a feature-oriented, distributed version control system. It started out as an approach to re-engineering variability from sets of cloned system variants [38, 39] by identifying traces from features to implementation artifacts in these variants, and then consolidating them into a product-line platform. Later, the same approach was used for incremental construction of product lines by supporting clone-and-own development with systematic and automated reuse [20, 37]. Similar to VTS, ECCO aims at combining the simplicity and flexibility of clone-and-own with the efficiency and scalability of structured product-line development. The original ECCO approach supported feature-based variation management. Later it was extended with revision support for individual features, and now evolved into a feature-oriented version and variation control system [34, 36]. Interestingly, the ECCO core still relies on the original algorithms that were used for re-engineering cloned variants.

ECCO now provides checkout and commit functionality for retrieving and updating the contents of its repository. Additionally, it has recently received experimental support for distributed development via fork, push and pull functionality for transferring features between different repositories. Developers can use a command-line tool and a graphical tool (both implemented based on ECCO's Java API) for accessing and modifying its repository.

Checking out a configuration provides the respective code and other artifacts in the file system. Developers can then work with arbitrary tools when adding new features or changing existing ones. Committing a new configuration updates the contents of the repository by automatically computing or updating the presence conditions of the affected artifacts. Internally, ECCO stores implementation artifacts as a generic tree structure where sub-trees are labeled with presence conditions. ECCO supports variability in any type of file for which a plugin is available that can translate it into ECCO's internal tree structure. In case of file types for which no specific plugin is available, variability is only supported at the level of entire files.

4.6 SuperMod Version Control System

SuperMod (Superimposition of Models)[5] [52, 53] aims at the integration of revisions and variants (called 'variability in time' and 'variability in space' by the authors) by integrating temporal and logical versioning approaches. The approach allows developers to better manage the complexity of handling logical variants for different revisions of a software system. The authors pursue a model-driven approach and use feature models to define logical variants and constraints in addition to a revision graph covering the evolution over time. SuperMod uses the well-known checkout/commit paradigm: software variants can be specified and checked out using feature configurations, which resolve the variability defined in the models. Developers can then make changes in tools of their choice. When committing changes developers need to define an ambition, a partial feature configuration defining the logical scope of the change. The approach is implemented using the Eclipse Modeling Framework (EMF) and available as a plugin[6] to the Eclipse IDE.

4.7 Excluded Subjects

We excluded several subject systems for which we could not find the publications anymore or if the description lacked details important for our classification and comparison. For instance, Conradi et al. [13] and Munch [43] refer to the editor MVPE [50]. However, since both authors report MVPE as an extension of P-EDIT, we believe that conceptually it

[3] https://bitbucket.org/modelsteam/2016-vcs-marlin/src/master/prototype/
[4] http://jku-isse.github.io/ecco/
[5] http://www.ai1.uni-bayreuth.de/de/projects/SuperMod
[6] http://btn1x4.inf.uni-bayreuth.de/supermod/update

does not differ too much from the P-EDIT system included in our comparison. Conradi et al. [13] also discuss PIE and DaSC. PIE [23] is probably the oldest system supporting fine-grained variability and was developed in the late 1970s [13]. DaSC [22, 40] is similar, relying on the concept of "selectors" that compose variants based on concurrent versions of assets. Developed for version and variation control of small teams, it supports the typical functionality of modern version control systems, including collaboration and consolidation, the latter referring to merge support. However, we could not find a detailed description and the tool is no longer available. Nevertheless, it would be very valuable to understand whether and how the consolidation support also covers integrating variants. Aide-de-Camp is reported to be similar, but we also could not find the papers [15, 55] anymore. Atkins et al. [6] evaluate Labs' VE (version editor) [5, 47], which reportedly has similar functionality as P-EDIT. Specifically, it also creates variation points automatically. VE has its roots in the so-called Delta System [14], which was also not accessible.

5 Classification of Variation Control Systems

Following our research process, all authors independently assessed the six identified subject systems. We then aligned and consolidated the individual views and defined a common set of characteristics and their possible values, sufficient to characterize systems in the domain of VarCS.

We now describe the resulting characteristics and illustrate them with concrete realizations and examples of the selected subject systems. We first discuss the general representation of variability in the systems: which abstractions are used to represent functionality belonging to different variants (commonly called *features* in SPLE)? Which artifacts can be variable by extending them with variation points? What are the general characteristics of these variation points? We then discuss how variable artifacts are presented to its users (external representation) and how the systems store all the variants and potentially their history (internal representation). We then discuss how the selected VarCS create the external from the internal representation, and how changes made by the users in the external representation are persisted in the internal representation. Table 1 illustrates these characteristics for our subjects.

We then proceed with describing the support offered for collaboration among developers and for aligning changes with the source artifacts. Finally, we discuss how each subject system was implemented and to what extent it was evaluated. Table 2 summarizes these characteristics.

Table 1. Characteristics of the VarCS.

			Subject Systems					
			P-EDIT	EPOS	Leviathan	VTS	ECCO	SuperMod
Entities	Boolean		●	●	●	●	●	●
	Integer		●		●			
	Enumeration		●					
Constraints	None		●		●	●	●	
	Set of Constraints			●				
	Variability Model							●
Var. Artifacts	Whole File			●	●		●	●
	Sub-File	Text (Lines)	●	●	●	●	●	
		Code (AST Nodes)					●	
		Models (Ecore)						●
		Any (Plugins)					●	
Revisions	None				●	●		
	Whole System		●	●				●
	Per Feature						●	
Internal	Storage	File System	●		●	●		
		Database		●			●	●
	VPs	Annotative	●	●	●	●		●
		Modular					●	
External	Type	Virtual	●					
		Materialized		●	●	●	●	●
	Spec.	Manual	●	●	●	●	●	●
		Automatic	●					
External-ization	Partial Configuration		●		●	●		
	Consistency Check						●	
Internal-ization	Expression	Weaker					●	
		Same	●	●	●	●	●	●
		Stronger		●		●	●	●

5.1 Variability Entities

The subject systems use different types of entities to abstractly describe and express variants. These entities can take values of different data types (*Boolean*, *integer*, *enumerations*, etc.). They are user-visible and mapped to *variation points* in *variable artifacts* via a mapping. For instance, P-EDIT uses *options* which can take Boolean, numeric, or enumerated values. EPOS also uses *options*, but only allows Boolean values. Leviathan, SuperMod, VTS, and ECCO use the notion of *features*, which are restricted to Boolean values. Despite the different names we did not find any conceptual differences between these entities, as they are all labels representing variable functionality, which is either modularized (mapped to one artifact) or cross-cutting (mapped to multiple artifacts).

5.2 Constraints over Variability Entities

This characteristic covers the different ways for declaring constraints over the variability entities, a.k.a. configuration constraints, composition constraints, or feature constraints.

None. Most of our systems do not support declaring constraints. This can be accounted for in smaller projects by using experts for configuring the system, or by making the mapping between features and artifacts more complex (see presence conditions in Section 5.3).

Set of Constraints. CoV allows specifying constraints among options, called rules (validities, constraints, preferences, defaults) [24]. Interestingly, these rules can also be used to define access rights.

Variability Model. Feature models [9, 27] and decision models [51] have been proposed for defining and managing the commonalities and variabilities in software product lines [17]. Feature models allow organizing features and constraints graphically using elements such as feature groups, hierarchy constraints, mandatory and optional features. This helps developers to keep a better overview of the system and to more easily evolve it. Only SuperMod currently uses feature modeling to define logical variants and constraints.

5.3 Variable Artifacts

This characteristic addresses the types of artifacts that can be managed by each VarCS. Specifically, this involves the artifact types that can be made variable by introducing variation points and the granularity of these variation points. Another distinguishing property is whether the VarCS also supports non-variable artifacts.

Variation Granularity. This characteristic describes the granularity at which variability in artifacts is supported. Recall that we exclude the system level according to our definition of VarCS. While file-level variability is independent of the file type, we observe different file-format limitations and variation granularities for variability within files.

File-Level Granularity. Except for P-EDIT and VTS, all systems support file-level granularity for all kinds of files (including binary). P-EDIT and VTS only handle variability within text files and do not use a repository with folders and files.

Sub-File-Level Granularity. This granularity is supported by all our VarCS for different types of implementation artifacts. P-EDIT, Leviathan, VTS, and EPOS support variability in arbitrary text files via variation points, regardless of the actual text format (e.g., code in various programming languages, documentation, or help pages). The granularity is text lines for VTS, EPOS, and P-EDIT. For EPOS, it could potentially be at character-level, but we could not find detailed information about the granularity of artifact fragments, beyond some speculation from the main EPOS developer: "There may be good arguments for trying smaller syntactical units (words, statements, language tokens)" [43]. The evaluation, however, is based on importing an `#ifdef`-based system, indicating that at least text lines are the finest level of granularity. SuperMod supports variation points within Ecore models by annotating model elements with presence conditions (cf. Section 5.5). ECCO can be extended with plugins to support variability for different types of files. The granularity of variation is determined by each respective plugin. Currently, plugins exist for whole files (a fallback for non-textual file types for which no specific plugin exists), line-based text files (a fallback for textual files like source code for which no specific plugin exists), and Java (based on the AST nodes rather than lines of code). Other plugins for C/C++, Ecore models, or STEP files (3D CAD drawings) are under development.

5.4 Revisions

This characteristic addresses the support of the VarCS for supporting revisions, i.e., versions intended to represent the sequential evolution, at different levels of granularity. The latter classifies into *system level* (to track changes of the whole system), *feature level* (to track changes of individual features), or *no* revisions at all.

All systems except Leviathan and ECCO allow creating revisions of the *whole system*. For instance, P-EDIT allows revisions of the whole system via artificial integer options (VERSION, TIME, RELEASE) that store the revision number. A similar approach also works for VTS, although the system lacks dedicated support. ECCO supports revisions at the level of individual features, thus allowing versions of 'snippets' within files.

5.5 Internal Representation

This characteristic describes the internal representation the subject systems use for storing artifacts, variation points, and the mapping to variability entities.

Artifact Storage. The selected VarCS use different ways to store the artifacts and their variants: EPOS and ECCO use a database, while Leviathan, VTS, and P-EDIT rely on the file system for loading and saving artifacts. ECCO's modular architecture provides more flexibility and additional persistency mechanisms can easily be added.

Variation Points. All systems use the notion of presence condition to represent variation points. Specifically, a presence condition is a Boolean expression over variability entities declared for artifacts (or parts of these). It specifies to which variant an artifact belongs to, thus controlling if it should be included when obtaining an external representation.

P-EDIT uses *custom annotations* inside text files that contain variability. In a variable text file the presence condition expressions are appended at the end of each line, delimited by special characters (double blank), to determine in which variants the line is to be included. VTS relies on *standard C-preprocessor*[7] *directives* for conditional compilation (e.g., `#ifdef` or `#if`) to annotate textual artifacts. Leviathan can use

[7] https://gcc.gnu.org/onlinedocs/cpp/

both C-preprocessor directives and M4[8] macros. SuperMod also uses custom annotations within Ecore model files.

EPOS does not annotate variable artifacts directly in their files. Instead, it decomposes files into fragments (e.g., consecutive text lines) that are then stored in a database. These fragments are mapped to presence conditions (called visibility conditions [45]), i.e., propositional logic expressions over options that are also stored in the database. This could be seen as a kind of custom annotation, but since fragments with the same presence condition are arranged close to each other in the database, it could also be seen as a kind of modularized storage, similar to feature modules.

ECCO has a custom data model resembling feature modules known from the paradigm of Feature-Oriented Software Development [4]. The model uses a generic tree structure for representing artifacts. Sub-trees of the implementation of a system under development are labeled with presence conditions (Boolean expressions over features).

5.6 External Representation

This characteristic describes the interface the VarCS offer to the user—in other words, how artifacts of a software system are presented. For instance, a VarCS might resolve all variability in the software system and only show full variants (with all artifacts being non-optional) or allow unresolved variability, meaning that multiple system variants will be shown to the user with variation points exposed.

Type. A core characteristic is whether the external representation (a variant or multiple variants) is *materialized*—that is, copied to a file system where it can be edited like ordinary files—or *virtual*—that is, the individual variants are only shown in the editor but need to be saved to be materialized.

Leviathan, EPOS, VTS, ECCO, and SuperMod produce a materialized external representation, while P-EDIT's external representation is virtual.

Specification. In all systems, the external representation is specified by a logical expression over the variability entities, a.k.a. configuration. This expression is commonly specified *manually* by the developer. A configuration is fully specified if all entities are decided. It is partially specified if variability is still left undecided in the external representation.

Different terms are used to denote a configuration in the VarCS and the syntactic rules slightly differ. Leviathan calls it *variant* (conjunction of features), EPOS and SuperMod call it *choice* (conjunction of options/features), VTS calls it *projection expression* (arbitrary Boolean expression over features), while ECCO uses the term *configuration* (conjunction of features). Note that SuperMod provides graphical support by allowing to manually create a configuration expression by selecting features in the graphical feature model. In P-EDIT a configuration is an expression over options. It is named *mask* and also allows arithmetic comparison operators, which can be used for selecting versions.

Interestingly, P-EDIT is the only subject system that does not require the users to specify the externalization expression manually. Instead, users can point to text lines and use them as a reference to automatically use their mask, thus obtaining the external representation "through the text" [29]. P-EDIT in this way specifies the external representation *automatically* by generating it from the users' artifact selections.

5.7 Externalization

This characteristic refers to the strategy of obtaining an external representation from the internal one. A core discriminator among the systems is whether partial configurations are supported. This influences the mechanism for obtaining and presenting the external representation to the user.

Partial Configuration. ECCO, EPOS, and SuperMod require a full configuration with all variability entities (e.g., features) decided. In this case, the external representation does not contain any remaining variation points.

Leviathan, VTS and P-EDIT support partial configurations, leaving variation points in the external representation. Variation points are then represented using conditional compilation directives, as in VTS, or using text highlighting, as in P-EDIT. Let us illustrate both mechanisms.

P-EDIT checks for every artifact with a variation point in the internal representation if its presence condition is incompatible with the mask (i.e., the conjunction of both is false). In this case, the asset is completely invisible ("as though its code did not exist" [30]). If the presence condition is implied by the mask (i.e., their conjunction evaluates to true), it is shown as ordinary text (called "fixed" by the author [30]). If the conjunction of the mask and the presence condition evaluates to neither true nor false, the artifacts are "displayed bright" (called "unfixed" by the author [30]).

VTS has a similar approach. An artifact becomes invisible, if the projection expression is incompatible with the artifact's presence condition. If the presence condition is not determined, then the conditional-compilation directives (e.g., `#if`) remain visible; yet, they are simplified by removing the part of the projection expression. If the presence condition is implied by the mask, then the artifact is shown as ordinary text not wrapped by conditional-compilation directives.

Consistency Checking. Finally, recall that concrete variants (external representations) could be inconsistent. Such inconsistencies can be caused by specifying invalid externalization expressions that violate constraints over the variability entities. For instance, a variable declaration might be missing if the feature containing it has not been included in the externalization expression. A VarCS could check the consistency upon obtaining the external representation. However, the VarCS aim at being oblivious to the underlying artifact formats and do not perform such consistency checks. An

[8] https://www.gnu.org/software/m4/m4.html

exception is ECCO, which can use the knowledge available in file-type-specific plugins for simple checks.

5.8 Internalization

This characteristic describes the mechanism by which an external representation can be transitioned into and merged with the current internal representation, or in other words, how an internal representation can be refined based on an external representation.

Internalization Expression. The most important and distinguishing characteristic is how changes made by modifying the external representation are applied to the internal representation. All VarCS follow a typical workflow: externalization (i.e., creating a view representing one or multiple variants), editing (changing the artifacts belonging to the view), and internalization (applying changes back consistently). While all VarCS represent the application of changes as an expression (full or partial configuration), they mainly differ in the freedom of specifying this expression in relation to the externalization expression.

Restrictions on Internalization Expression. In P-EDIT and Leviathan, the internalization expression, which represents the scope of changes, cannot be set by the user. It is the same as the externalization expression and therefore determined during externalization. Changing it requires a new cycle in the VarCS workflow.

EPOS, VTS, and SuperMod are more flexible and allow the user to set the internalization expression. All three approaches call this expression "ambition," obviously inspired by the CoV paradigm. However, the ambition can only be stronger or the same as the externalization expression, which is a limitation. In fact, being able to specify a weaker expression, which would allow to apply changes to variants not visible in the view, is an unsolved problem requiring further investigation. Also recall that EPOS and SuperMod require full configurations, while only VTS allows the ambition to be partial. ECCO allows arbitrary internalization expression regardless of the previously used externalization expression.

5.9 Alignment Strategy

This characteristic describes the strategies our VarCS employ for aligning changes with the existing artifacts (which potentially belong to different variants) upon internalization.

We learned that alignment can be difficult based on the externalization and internalization strategy employed. Specifically, alignment problems can occur when the externalization is hiding artifacts that are not contradicted by the internalization expression used to insert artifacts at the same location where the hidden artifacts would be. This can happen in any VarCS that does not enforce internalization expressions that are at least as strong as the externalization expression that was used to produce the current view. In this case, if a developer adds new code, this code could affect variants that are currently not visible. A consequence of this could for example be that surrounding code which is active in other variants might be hidden in the developer's view, making the position of the new code ambiguous in those variants. In this case, manual or automated alignment needs to be performed. If the alignment is done in the wrong way the syntax or semantics of the artifact could be violated (e.g., statements would be put in the wrong order).

Leviathan performs such alignment based on heuristics and, if specified, on manual annotations that can be created by users to instruct Leviathan on aligning changes. ECCO performs structured merging during internalization based on its internal tree structure in combination with partial-order-relations for merging nodes on the same tree level. For EPOS and SuperMod, we could not identify any alignment strategy.

P-EDIT and VTS do not need dedicated alignment support, given their externalization strategy. Specifically, only artifacts that contradict the externalization expression are hidden. So, the hidden artifacts can never appear together with the current changes. Artifacts that are still variable are just highlighted (P-EDIT) or still appear within conditional-compilation directives (VTS). This, of course, only works when the internalization expression cannot be weaker than

Table 2. Characteristics of the VarCS.

			Subject Systems					
			P-EDIT	EPOS	Leviathan	VTS	ECCO	SuperMod
Alignment	Not Necessary		●			●		●
	Textual			●	●			
	Structural						●	
Collaboration	None/Local		●		●	●		
	Centralized			●				
	Distributed						●	●
Implementation	Modality	Editor	●					
		Version Control		●		●	●	●
		File System			●			
	Interface	Textual	●	●		●	●	
		Graphical					●	●
	Intrusiveness	Internal	●	●			●	●
		External	●					
	Available	Binary				●	●	●
		Source					●	
Evaluation	Exemplary			●		●		●
	Qualitative					●		
	Quantitative						●	
	Formal							

the externalization expression, which is enforced by both systems. In fact, it is not clear whether allowing weaker internalization expressions can be realized together with respective alignment strategies, and what the benefit of this flexibility would be. Investigating this is valuable future work.

5.10 Collaboration

This characteristic describes how the subject systems support collaboration: not at all (local), centralized, or decentralized/distributed.

P-EDIT supports no collaboration and works only locally. Leviathan also does not support collaboration, however, it is implemented as a file system and could in principle support centralized collaboration if implemented similar to a network file system. EPOS behaves similarly to a database system and thus supports centralized collaboration. VTS does not support any collaboration. ECCO and SuperMod allow for decentralized (i.e., distributed) collaboration.

5.11 Implementation and Tool Support

We also investigated if the subject systems currently have—or at some point had—an implementation or tool support for their theoretical concepts. This is important as many of the other characteristics require a concrete implementation of a concept to be answered. However, many papers provide no or only limited information to what extent they have been implemented and if tools supporting the approach are (publicly) available. As some of the VarCS were first published long time ago, most of the tools and frameworks discussed in the papers are not available to the public. In several cases tools existed at some point in time, which are no longer available or maintained.

We also compared the tools regarding their nature, kind of user interface, and integration in the development environment.

Modality. We distinguish different modalities of the tools, for example *editor*, *version control system*, or *file system*. For instance, P-EDIT was implemented as an editor. Leviathan has been realized as a virtual file system. EPOS, VTS, ECCO, and SuperMod use the checkout and commit metaphor from current version control systems. Yet, although less explicit for the latter three, all follow the same workflow (cf. Section 5.8).

User Interface. We distinguish textual and graphical user interfaces. All subject systems except SuperMod have a textual interface in form of a CLI or query language. In addition, ECCO also provides a graphical interface to navigate in the version control space. SuperMod is available as an Eclipse plugin with a focus on rich and interactive feature modeling, so no CLI is provided.

Intrusive. This characteristic addresses how the subject systems integrate into the development process, i.e. how intrusive the tools are when it comes to editing their internal and external representations. P-EDIT is intrusive on both internal and external representation. It assumes using its specific editor and does not allow any other editor or tool to modify the system artifacts (which would need to be exported first). EPOS stores the source code in its database and full variants can only be checked out using the tool. However, once a variant has been checked out it can be edited using existing tools. Leviathan and VTS are both non-intrusive as they use common annotations (C preprocessor or M4 macros) in both their internal and external representations. They do not require any specific tools for editing their internal or external representations. In case of ECCO, it depends on the storage plugin. The storage can be implemented using a database, but its feature modules could also be transformed into an annotative, file-based representation compatible with the C preprocessor, for instance. SuperMod requires its Eclipse based feature model editor to be used when performing its versioning operations. It is thus intrusive regarding the internal representation. However, after a version has been retrieved, existing tools can be used for editing the files in the retrieved external representation.

Availability. We checked if and in what form the implementation is available. ECCO's source code is publicly available on GitHub, but no precompiled binaries are provided. SuperMod is available as a binary plugin to the Eclipse IDE, but we could not find the sources. The VTS prototype is available as a binary, but the publication of its sources is planned. For all other subject systems the implementation was not available.

5.12 Evaluation

This characteristic assesses the degree and rigor of the scientific evaluation of the subject systems. This assessment is important for identifying shortcomings of VarCS and understanding the reasons for their limited impact on practice. The investigated VarCS are research-oriented prototypes and thus their validation must be framed in this context. Different kinds of evaluations are reported in the survey literature including *exemplary*, *qualitative*, *quantitative*, and *formal* methods, which we adopt for our classification. Many approaches have only been assessed through simple examples or through "friendly-enough" systems. Only in a few cases the evaluations were conducted on realistic open source systems. Specifically, VTS was evaluated by replaying the evolution of the open-source project Marlin (cf. Section 2), showing the applicability of the approach, although multiple checkout/commit cycles were required to realize certain kinds of variability. This was the case, for example, when at the same time adding two variants, which are represented by a conditional-compilation directive with an else branch. The correctness of ECCO was evaluated by replaying the development and evolution of open source product lines and measuring the correctness and usefulness of the results. It was also evaluated using an industrial system [35]. EPOS

was evaluated by importing the sources of the GNU C compiler into the EPOS database, and processing the conditional-compilation directives defining its many variants.

6 Discussion

A key motivation of our work was to better understand why VarCS, despite their advanced support for managing variants, did not achieve wide-spread adoption. We discuss problems of VarCS, which may have prevented their success. We further derive research activities.

Cognitive complexity. VarCS use logical expressions to handle variants of a system with different variability entities (e.g., features). Due to the high number of revisions and variants, this task becomes cognitively extremely demanding. For instance, creating externalization expressions is difficult for developers who think in terms of code and not in terms of variation points. A key for success is to improve developer support for working with complex logical expressions. Partial configurations, for instance, as supported by several VarCS, may help. Likewise, generating the externalization expression by letting users point to artifacts that should be in the variant, likely also helps, as the example of P-EDIT shows. This allows developers to avoid going via the abstraction (i.e., thinking in terms of options instead of code), which can be demanding even for small changes only affecting one or few artifacts. Furthermore, useful abstractions seem essential to facilitate the use of VarCS in this regard. For instance, feature models may help to significantly reduce the cognitive load by providing a higher-level and hierarchically-organized graphical perspective on a system. Developers can be supported by creating externalization expressions based on feature models, as the SuperMod system shows. However, besides the technical challenges of creating such a feature-based front-ends, user studies are needed to better understand how developers can cope with the complexity.

Change impact of updating variants. VarCS support developers by filtering details of configurable artifacts that are not part of the variant a developer is working on. Such views (or projections) ease the comprehension of these artifacts. At the same time it is very challenging to understand the scope of changes made in such views on other variants not shown in the view. Our study showed that while the investigated systems have found different solutions for this issue, the workflow is still rather complex from a developer's point of view. Although for VTS the evaluation confirmed that the capabilities are sufficient to handle a complex real-world evolution, the updating of variants was still complicated and sometimes required multiple checkout/commit cycles in the tool. In case of Leviathan changes to variants can be written back to the configurable code base automatically only if certain assumptions are met, meaning that developers need to manually double check if Leviathan applied the changes correctly. These findings suggest the integration of existing research on variability-aware change impact analysis that exists for instance in the area of program analysis [1]. Such techniques allow the development of tools visualizing the variants affected by a change to understand and assess its impact.

Locked-in syndrome. Developers are usually reluctant to commit to a proprietary repository technology for managing their software artifacts. Existing implementations of VarCS use diverse model-based approaches, various database technologies, and a wide range of mechanisms for representing variation points to manage the complex version space and artifacts. Overall, this increases the risk of becoming locked-in with no easy way to escape if technologies evolve. This problem may explain why the annotation-based preprocessors are still the most popular variability mechanism. A basic requirement for every VarCS should thus be the ability to export its content to such an annotation-based representation, and to populate its repositories by importing content from such a format. For systems such as VTS and Leviathan, which already use such formats as their internal representation, this should be possible with minimal effort. An even more advanced approach would be to use transformations to and from artifact-specific representations that require no additional variability mechanism at all. An example is the variability-encoding approach by Rhein et al., which transforms compile-time variability into load-time variability [59].

Adoption and migration barrier. In practice, systems are rarely planned with a high degree of variability from the beginning. In ad hoc reuse developers use available variability mechanisms (e.g., C-preprocessor when writing C code) or clone-and-own practices, usually leading to many independently maintained variants. By the time a VarCS system would pay off, migrating a system may already be difficult. VarCS systems should thus offer a mechanism to populate a repository from a set of clone-and-own variants to ease adoption and to enable migration from systems like Subversion or Git, where variants are maintained in separate branches. The same mechanism could be used to migrate between different VarCS systems by first creating all variants from one VarCS system (assuming they are not too many) and then importing them into another, thus reducing the locked-in syndrome mentioned above. Similarly, as already mentioned for the locked-in syndrome, artifact-specific options for migrating from common variability mechanisms would be beneficial, such as importing from preprocessor annotated text files.

Lack of collaboration support. The studied VarCS significantly improve the variation aspects regular version control systems are lacking. However, at the same time, many of the subject systems seem to neglect equally important aspects existing version control systems already support very well. In particular, an important aspect of version control systems is their support for collaboration among developers. Distributed development has become very popular in modern version control systems, especially in the development

of open source software. This could be a great opportunity for variation control systems to shine, as the independent development of individual functionality (i.e., features) is well suited for distributed development. This becomes evident when looking at popular Git branching models that already facilitate so-called *feature branches*[9], i.e., temporary branches that live as long as it takes to develop a new feature until they are eventually merged back into their parent branches. VarCS need to evolve towards distributed platforms for development and evolution support. Current version control systems support cloning of entire repositories, but lack support for handling variants at the level of features. We envision clone operations that will be based on specific feature selections and include only the features needed for a specific development task. Further, it should be possible to push, pull, or transfer features between platforms. For instance, a push feature operation may allow transferring a feature back to its original platform to make it generally available.

Low tool maturity, availability, and rigor of evaluations. Finally, our study showed that a lot needs to be done regarding the availability and maturity of the VarCS tools. ECCO, VTS, and SuperMod are the only implementations currently available. Another issue is the low maturity of the reported evaluations. Although replaying existing version histories of open source systems is a promising first step to demonstrate the feasibility of VarCS there is a strong need for case studies with industrial partners, which also need to elicit relevant usage scenarios of VarCS to measure their benefits.

7 Threats to Validity

A threat to the external validity of our study is that we might have missed important VarCS. To mitigate this problem, we studied the literature, starting with well-cited and large surveys of the SCM literature [13]. One author also talked to SCM researchers to make sure that we did not miss any VarCS relevant for our work. Another threat to the external validity is that the systems unavailable for our classification encompass important concepts and technologies. To mitigate this issue, we consulted secondary literature describing these systems to get a coarse overview and to understand their key ideas. We did not find any hints to concepts not supported by the other investigated systems. As such, we are confident that our studied systems are characteristic examples of VarCS.

A threat to the internal validity is that we might have misclassified the systems, especially those that have been developed by SCM researchers. Given that some papers were almost 40 years old, it was not always easy to understand the terminology used by the community at that time [23]. We therefore started with individual classification and worked to achieve consensus on the degree of support of features, often in several rounds. Finally, our classification might not be complete, i.e., we might have missed important concepts that are needed to realize new and better systems. However, we created the classification with the clear goal to identify concepts that are important for the practical use and adoption of VarCS, so overall we are confident that we identified the most relevant concepts. Still, contrasting it with systems that will be developed in the future is valuable further work. Finally, we were ourselves involved in the development of VTS and ECCO, which could potentially bias our classification. However, one author started surveying existing VarCS before either of both systems was developed, thus mitigating this potential bias.

[9] http://nvie.com/posts/a-successful-git-branching-model/

8 Conclusion

This paper presented a classification of VarCS, which aim to integrate the management of revisions of software artifacts and the handling of software variants at different levels of granularity. Our study provides a classification of six VarCS and shows that they use concepts and approaches developed in the areas of both software configuration management and software product line engineering. The results show that the investigated VarCS share a common core of capabilities although they were developed in different research communities, for a different purpose, and in a time span covering several decades. The results also reveal particular strengths of individual VarCS.

Based on these findings, we discussed reasons that may have prevented the wide-spread use of VarCS: these include the cognitive complexity of handling logical variants for different revisions of features, the complex workflow needed to consistently write back changes made to variants to the shared artifact repository, and the risk of become locked-in a particular style of artifact repository. Only some evaluations of existing VarCS are convincing. Furthermore, only few robust implementations exist today, which makes the transition to industrial practice difficult.

Acknowledgments

This work has been supported by the Christian Doppler Forschungsgesellschaft Austria, KEBA AG Austria, the Swedish Research Council, and Vinnova Sweden.

References

[1] Florian Angerer, Andreas Grimmer, Herbert Prähofer, and Paul Grünbacher. 2015. Configuration-Aware Change Impact Analysis. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. 385–395.

[2] Florian Angerer, Herbert Prähofer, Daniela Lettner, Andreas Grimmer, and Paul Grünbacher. 2014. Identifying Inactive Code in Product Lines with Configuration-Aware System Dependence Graphs. In *Proceedings 18th International Software Product Line Conference (SPLC 2014)*. ACM, New York, NY, USA, Florence, Italy, 52–61.

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer, Berlin Heidelberg.

[4] Sven Apel and Christian Kästner. 2009. An Overview of Feature-Oriented Software Development. *J. Object Techn.* 8, 5 (2009), 49–84.

[5] David L. Atkins. 1998. Version Sensitive Editing: Change History As a Programming Tool. In *Proceedings of the SCM-8 Symposium on System Configuration Management (ECOOP '98)*. Springer Verlag, London, UK, 146–157.

[6] David L. Atkins, Thomas Ball, Todd L. Graves, and Audris Mockus. 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 (2002), 625–637.

[7] Benjamin Behringer, Jochen Palz, and Thorsten Berger. 2017. PEoPL: projectional editing of product lines. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28*. 563–574.

[8] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *Proceedings 19th International Software Product Line Conference (SPLC'15)*. ACM, 16–25.

[9] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wasowski. 2014. Three Cases of Feature-Based Variability Modeling in Industry. In *Proceedings 17th International Conference Model-Driven Engineering Languages and Systems (MODELS'14), Valencia, Spain, September 28 – October 3*. Springer International Publishing, 302–319.

[10] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. 2016. Efficiency of Projectional Editing: A Controlled Experiment. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 763–774.

[11] David Budgen, Mark Turner, Pearl Brereton, and Barbara Kitchenham. 2008. Using mapping studies in software engineering. In *Proc. of PPIG*, Vol. 8. Lancaster University, 195–204.

[12] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA.

[13] Reidar Conradi and Bernhard Westfechtel. 1998. Version Models for Software Configuration Management. *ACM Comput. Surv.* 30, 2 (1998), 232–282.

[14] J. O. Coplien, D. L. DeBruler, and M. B. Thompson. 1987. The Delta System: A Nontraditional Approach to Software Version Management. In *AT&T Technical Papers, International Switching Symposium*.

[15] Randall D. Cronk. 1992. Tributaries and Deltas. *BYTE* 17, 1 (1992), 177–186.

[16] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, MA.

[17] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '12)*. ACM, New York, NY, USA, 173–182.

[18] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *Proceedings 17th European Conference on Software Maintenance and Reengineering*. 25–34.

[19] J. M. Favre. 1996. Preprocessors from an abstract point of view. In *Proceedings of the Third Working Conference on Reverse Engineering*. 287–296.

[20] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *Proceedings 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3*. 391–400.

[21] Christopher W. Fraser and Eugene W. Myers. 1987. An Editor for Revision Control. *ACM Trans. Program. Lang. Syst.* 9, 2 (March 1987), 277–295.

[22] W. M. Gentleman, A. MacKay, and D. A. Stewart. 1989. Commercial Realtime Software Needs Different Configuation Management. In *Proceedings of the 2nd International Workshop on Software Configuration Management (SCM '89)*. ACM, New York, NY, USA, 152–161.

[23] Ira P Goldstein and Daniel G Bobrow. 1980. *A layered approach to software design*. Technical Report CSL-80-5. Xerox. Palo Alto Research Center.

[24] Bjørn Gulla, Even-André Karlsson, and Dashing Yeh. 1991. Change-oriented Version Descriptions in EPOS. *Softw. Eng. J.* 6, 6 (Nov. 1991), 378–386.

[25] Wanja Hofer, Christoph Elsner, Frank Blendinger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2010. Toolchain-independent Variant Management with the Leviathan Filesystem. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development (FOSD '10)*. ACM, New York, NY, USA, 18–24.

[26] Per Holager. 1988. *Elements of the design of a change oriented configuration management tool*. Technical Report STF44-A88023. ELAB, SINTEF, Trondheim, Norway.

[27] Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report SEI-90-TR-21. CMU.

[28] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 311–320.

[29] Vincent Kruskal. 2000. A blast from the past: Using P-EDIT for multi-dimensional editing. In *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*.

[30] Vincent J. Kruskal. 1984. Managing Multi-Version Programs with an Editor. *IBM Journal of Research and Development* 28, 1 (1984), 74–81.

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

[32] Anund Lie. 1990. *Versioning in Software Engineering Databases*. Ph.D. Dissertation. The Norwegian Institute of Technology.

[33] Anund Lie, Reidar Conradi, Tor Didriksen, and Even-André Karlsson. 1989. Change Oriented Versioning in a Software Engineering Database. In *Proceedings of the 2nd International Workshop on Software Configuration Management (SCM), Princeton, NJ, USA, October 24*. 56–65.

[34] Lukas Linsbauer. 2016. *Managing and Engineering Variability Intensive Systems*. Ph.D. Dissertation. Johannes Kepler University Linz.

[35] Lukas Linsbauer, Florian Angerer, Paul Grünbacher, Daniela Lettner, Herbert Prähofer, Roberto Lopez-Herrejon, and Alexander Egyed. 2014. Recovering Feature-to-Code Mappings in Mixed-Variability Software Systems. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME'14)*. 426–430.

[36] Lukas Linsbauer, Alexander Egyed, and Roberto Erick Lopez-Herrejon. 2016. A variability aware configuration management and revision control platform. In *Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, May 14-22, 2016 – Companion Volume (ICSE 2016)*. 803–806.

[37] Lukas Linsbauer, Stefan Fischer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. Using Traceability for Incremental Construction and Evolution of Software Product Portfolios. In *Proceedings IEEE/ACM 8th International Symposium on Software and Systems Traceability (SST)*. IEEE Computer Society, 57–60.

[38] Lukas Linsbauer, E. Roberto Lopez-Herrejon, and Alexander Egyed. 2013. Recovering Traceability Between Features and Code in Product Variants. In *Proceedings of the 17th International Software Product Line Conference (SPLC'13)*. ACM, New York, NY, USA, 131–140.

[39] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2016. Variability extraction and modeling for product variants. *Software & Systems Modeling* (29 Jan 2016).

[40] Stephen A. MacKay. 1995. The State of the Art in Concurrent, Distributed Configuration Management. In *Selected Papers from the ICSE SCM-4 and SCM-5 Workshops on Software Configuration Management*.

[41] Axel Mahler. 1995. Configuration Management. John Wiley & Sons, Inc., New York, NY, USA, Chapter Variants: Keeping Things Together and Telling Them Apart, 73–97.

[42] Jean Melo, Claus Brabrand, and Andrzej Wąsowski. 2016. How Does the Degree of Variability Affect Bug Finding?. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 679–690.

[43] Bjørn P. Munch. 1993. *Versioning in a Software Engineering Database – The Change Oriented Way*. Ph.D. Dissertation. The Norwegian Institute of Technology.

[44] Bjørn P. Munch, Reidar Conradi, Jens-Otto Larsen, Minh N. Nguyen, and Per H. Westby. 1996. Integrated Product and Process Management in EPOS. *Integr. Comput.-Aided Eng.* 3, 1 (1996), 5–19.

[45] Bjørn P. Munch, Jens-Otto Larsen, Bjørn Gulla, Reidar Conradi, and Even-André Karlsson. 1993. Uniform Versioning: The Change-Oriented Model. In *Proceedings of the Fourth International Workshop on Software Configuration Management (SCM-4), May 21–22*. 188–196.

[46] Robert C. Nickerson, Upkar Varshney, and Jan Muntermann. 2013. A method for taxonomy development and its application in information systems. *EJIS* 22, 3 (2013), 336–359.

[47] A. A. Pal and M. B. Thompson. 1989. An advanced interface to a switching software version management system. In *Proceedings of the 7th International Conference on Software Engineering for Telecommunication Switching Systems (SETSS)*.

[48] David Parnas. 1976. On the design and development of program families. *IEEE Transactions on Software Engineering* SE-2, 1 (July 1976), 1–9.

[49] Marc J. Rochkind. 1975. The Source Code Control System. *IEEE Trans. Softw. Eng.* 1, 1 (March 1975), 364–370.

[50] N. Sarnak, R. Bernstein, and V Kruskal. 1988. Creation and maintenance of multiple versions. In *Workshop on Software Version and Configuration Control*.

[51] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. 2011. A Comparison of Decision Modeling Approaches in Product Lines. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '11)*. ACM, New York, NY, USA, 119–126.

[52] Felix Schwägerl and Bernhard Westfechtel. 2016. Collaborative and Distributed Management of Versioned Model-driven Software Product Lines. In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) – Volume 2: ICSOFT-PT, Lisbon, Portugal, July 24-26*. 83–94.

[53] Felix Schwägerl and Bernhard Westfechtel. 2016. SuperMod: tool support for collaborative filtered model-driven software product line engineering. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, September 3-7 (ASE 2016)*. 822–827.

[54] Nieraj Singh, Celina Gibbs, and Yvonne Coady. 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)*.

[55] Software Maintenance & Development Systems, Inc. 1990. Aide de Camp Product Overview. Concord, Massachusetts. (Sept. 1990).

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

[57] Stefan Stanciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wasowski. 2016. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *Proceedings IEEE International Conference on Software Maintenance and Evolution, Raleigh, NC, USA, October 2-7 (ICSME 2016,)*. 323–333.

[58] Walter Tichy. 1988. *Software Configuration Management Overview*. Technical Report.

[59] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. 2016. Variability encoding: From compile-time to load-time variability. *Journal of Logical and Algebraic Methods in Programming* 85, 1 (2016), 125–145. Formal Methods for Software Product Line Engineering.

[60] Eric Walkingshaw and Klaus Ostermann. 2014. Projectional editing of variational software. In *Generative Programming: Concepts and Experiences, GPCE'14, Västerås, Sweden, September 15-16*. 29–38.

[61] Alf Inge Wang, Jens-Otto Larsen, Reidar Conradi, and Bjørn P. Munch. 1998. Improving Cooperation Support in the EPOS CM System. In *Proceedings 6th European Workshop on Software Process Technology, Weybridge, UK, September 16–18 (EWSPT'98)*. Springer Berlin Heidelberg, 75–91.

[62] Bernhard Westfechtel, Bjørn P. Munch, and Reidar Conradi. 2001. A Layered Architecture for Uniform Version Management. *IEEE Trans. Software Eng.* 27, 12 (2001), 1111–1133.