Concepts of Variation Control Systems

Lukas Linsbauer[a], Felix Schwägerl[b], Thorsten Berger[c], Paul Grünbacher[d]

[a]*ISF, Technische Universität Braunschweig, Germany*
E-Mail: l.linsbauer@tu-braunschweig.de
[b]*Applied Computer Science I, University of Bayreuth, Germany*
E-Mail: felix.schwaegerl@uni-bayreuth.de
[c]*Chalmers | University of Gothenburg, Sweden*
E-Mail: thorsten.berger@cse.gu.se
[d]*Institute for Software Systems Engineering, Johannes Kepler University Linz, Austria*
E-Mail: paul.gruenbacher@jku.at

Abstract

Version control systems are an integral part of today's software engineering. They facilitate the collaborative management of *revisions* (sequential versions) and *variants* (concurrent versions) of software systems under development. Typical version control systems maintain revisions of files and variants of whole software systems. Variants are supported via branching or forking mechanisms that conceptually clone whole systems in a coarse-grained way. Unfortunately, such cloning leads to high maintenance efforts. To avoid these disadvantages and support fine-grained variation, developers need to employ custom configuration mechanisms, which leads to a misappropriation of tools and undesired context switches. Addressing this trade-off, a number of *variation* control systems has been conceived, providing a richer set of capabilities for handling variants. Variation control systems decompose a software system into finer-grained variable entities and offer high-level metaphors to automatically manage this variability. In this paper, we classify and compare variation control systems and illustrate their core concepts and characteristics. All investigated variation control systems offer an iterative (checkout-modify-commit) workflow, but there are essential differences affecting developers. We highlight challenges and discuss research perspectives for developing the next generation of version and variation control systems.

Keywords: variability management, software product lines, configuration management, version control, software repositories

1. Introduction

Managing *revisions* (sequential versions) and *variants* (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 *variants* are supported by cloning entire systems or creating branches. 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. Consequently, developers need to use mechanisms and tools [1] in addition to version control systems. For instance, they use preprocessors, build systems such as *Make*, or even custom configuration solutions as for example in the Linux kernel [2, 3]. Such configurable systems (a.k.a., software product lines [4]) 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 [5, 6, 7].

The *problems* and *challenges* of current practices in variant management can be described as follows:

- Existing mechanisms for dealing with variability require developers to manually edit variation points and to manually integrate changes (e.g., made in feature branches), which increases the complexity of managing variant-rich systems.
- Most existing mechanisms have a strong impact on the way software needs to be developed. This can range from adding annotations to code in the simplest case to employing new paradigms such as feature-oriented or aspect-oriented programming, in each case placing a burden on developers.
- Most mechanisms realize variability at the level of source code. This hinders developers to focus on the problem at hand, since variability increasingly tends to govern the structure and contents of source code just as much as the actual functionality. It further complicates the build process, as new build steps need to be added.
- The widely used variability mechanisms usually do not support revision management, which is usually taken care of by version control systems. This means that the two closely related concepts of variants and revisions as specializations of versions cannot be handled uniformly by developers.
- While the underlying concepts of variability mechanisms are often generic (e.g., aspect-oriented programming), their concrete realization is usually specific for a certain type of artifact (e.g., AspectJ for Java). For heterogeneous systems consisting of many different types of implementation artifacts this further complicates the build process, and requires developers to learn even more tools. Furthermore, keeping variability information consistent across different types of artifacts and tools is a manual and error-prone task.

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. As such, it reduces the complexity of changing variants and frees developers from manually maintaining variation points. We define a VarCS as follows:

Definition. *A* Variation Control System (VarCS) *supports managing variant-rich systems in terms of features. It supports editing variant subsets, which are represented by a selection of features, and it automatically integrates the edited variant subsets back into the variant-rich system in a transactional way.*

This definition emphasizes the variation of software systems using features, as opposed to variation per system (e.g., per customer) through cloning (or branching) of software systems. The definition also emphasizes automation, since VarCS alleviate developers from editing variation points manually and from integrating the changes manually. As such, a VarCS aims at reducing the complexity of editing feature-based variant-rich systems.

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 only little evidence demonstrating their success in real-world scenarios. Obviously, the terminology used to describe these systems is often not consistent across the different research communities, which further challenges their comparison. Furthermore, while all VarCS take away the responsibility from the developer to manually specify variation points (according to our definition of VarCS), there are major differences in how this is achieved. Finally, while VarCS have the potential to solve all the above mentioned problems of current practice, most of them only tackle a subset of them. For example, not all of them support revisions in addition to variations, or more than one type of implementation artifact.

In summary, the *motivation* and *goals* of this work are thus to

- *contribute to harmonizing research from different research communities* related to long-term and fine-grained variant management,
- *identify* and *compare* ancient and recent VarCS,
- *identify* and *understand* key *concepts and characteristics* of VarCS and provide a unified terminology, and to
- *identify remaining challenges* and *suggest opportunities for further research.*

We present a classification of VarCS to obtain an understanding of their concepts and characteristics. Our comparison reveals the key commonalities and differences between six selected subject systems. Our paper is intended for both researchers and practitioners interested in building, improving, and using VarCS.

An earlier version of this work appeared as a conference publication [8], which focused on identifying the core characteristics of VarCS. We have significantly extended this earlier paper in several directions: in particular, we extended and refined our classification of VarCS. We now also provide a running example from the open source Marlin firmware that is used throughout the paper to explain the basic VarCS concepts and the differences of the subject systems. We provide a completely new part that identifies essential differences between the VarCS regarding concepts, usage, and behavior based on a formal concept analysis and a scenario-based comparative analysis applying the three most recent subject VarCS to our running example.

In summary, the *contributions* of this work are:

- an *integrated and unified view on research* from the *software configuration management* and *software product line engineering* communities,
- a *definition* of *variation control systems* including their *characteristics and concepts*,
- an *illustration* of all characteristics and concepts based on a running example,
- a *classification and comparison* of existing variation control systems based on their characteristics and concepts, and a
- *discussion of challenges* of existing variation control systems.

We proceed by presenting background information and our running example in Sec. 2. We describe our research methodology in Sec. 3, then provide an overview of our six selected variation control systems in Sec. 4. We provide our classification and its instantiation for our subjects in Sec. 5, illustrated by the running example. In Sec. 6 we present results of the analyses we conducted to reveal essential differences between the VarCS: a formal concept analysis and a scenario-based behavioral analysis. We discuss research challenges and perspectives in Sec. 7, threats to validity in Sec. 8, and conclude in Sec. 9.

2. Background and Related Work

Software evolves 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* [9]. Two research communities have developed a wide range of approaches to handle revisions and variants.

2.1. Software Configuration Management

The research community of *software configuration management* distinguishes between extensional and intensional versioning [9]. Extensional versioning means that only previously constructed versions can be retrieved, typically identified by a unique number (e.g., revision). That is, 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, so new combinations may be constructed on demand. Research on software configuration management has mainly focused on managing the evolution of software artifacts by tracking revisions, largely sidestepping variant support [10, 11]. Some researchers recognized the need for variant management, including the handling of variants at the level of user-facing, high-level properties (i.e., *features* [12] in product-line terminology). For instance, Gulla et al. [13] 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 have been developed by researchers in software configuration management. Although the community recognizes the need [14], actual variant-management support is still limited in contemporary software configuration management 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 [15, 16, 17] is simple and cheap, it does not scale with the number of variants, leading to costly variant re-engineering and integration efforts [18, 19, 20]. Existing tools lack support for handling variants at the level of files (or below) based on features, which limits their flexibility, since only a fixed set of manually created 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 [21], later leading to the field of *software product line engineering* [22, 23], which provides methods and tools to effectively manage portfolios of system variants. Variants are no longer managed at the level of customers, but at the level of features [24, 12, 4], 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 the desired features. This flexibility is achieved by mapping features to variation points, which are commonly realized using annotations, such as conditional-compilation directives (e.g., `#ifdef`). However, in complex systems, such annotations significantly clutter source code, which challenges program comprehension when many variants need to be edited at the same time [5]. Listing 1 shows a code excerpt of the 3D-printer firmware Marlin, which exists in many variants. Editing the code becomes difficult for developers who are only interested in one feature. Dedicated product line engineering tools such as C-CLR [25], CIDE [26], PEoPL [27], conditional SDGs [28] or GUPRO [29] can filter out irrelevant code, but do not consider the reintegration of variants into the platform, leaving it to the developers to manually edit variation points [30].

There are many techniques and mechanisms for implementing variable systems [4]. Some techniques employ common mechanisms and tools that are widely known and not specific to variability (e.g., preprocessors as shown in Listing 1). We briefly review some other tools and mechanisms specifically designed for the development of variable systems:

Feature Toggles and Runtime Variability. Feature toggles [31, 32] are programming-language-specific conditional statements placed into source code to realize variability. The code is compiled as usual, packaged in a binary, and its toggles are

```
uint8_t sort_order[SDSORT_LIMIT];

// Cache filenames to speed up SD menus.
#if SDSORT_USES_RAM

  // If using dynamic ram for names, allocate on the heap.
  #if SDSORT_CACHE_NAMES
    char sortshort[SDSORT_LIMIT][FILENAME_LENGTH];
    char sortnames[SDSORT_LIMIT][FILENAME_LENGTH];
  #elif !SDSORT_USES_STACK
    char sortnames[SDSORT_LIMIT][FILENAME_LENGTH];
  #endif

  // Folder sorting uses an isDir array when caching items.
  #if HAS_FOLDER_SORTING && (SDSORT_CACHE_NAMES || !SDSORT_USES_STACK)
    uint8_t isDir[(SDSORT_LIMIT+7)>>3];
  #endif

#endif
```

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

evaluated at runtime (a.k.a., *runtime variability*). No additional tooling and no special training for developers is required, which explains the popularity of feature toggles in practice. However, source code easily becomes cluttered with feature toggles, challenging its comprehension and sometimes leading to dead (i.e., unused) feature code. Furthermore, the code related to features cannot easily be distinguished from other code. The binaries contain the code for all possible feature configurations, unnecessarily increasing the binary size, which can be problematic on devices with limited memory, such as microcontrollers and IoT devices. The presence of all features in binaries can also lead to security problems, unwanted feature interactions or intellectual property leaks.

Version Control Systems and Build Systems. Using mechanisms provided by version control systems (e.g., Git) and build systems (e.g., Make) is another common practice to realize variability. Current version control systems provide support for branches, which are essentially clones. On the one hand, branches can be used to maintain variants of a system explicitly as clones. However, this can hardly be considered a variability mechanism as it operates on the system as a whole and does not offer fine-granular variation of individual files or folders based on features or a similar concept. Rather, it should be considered as tool support for clone-and-own. On the other hand, a common practice in many popular branching models is the use of temporary feature branches. A feature branch is created when the development of a new feature starts. The new feature is then developed on this branch in isolation until it is finished. Ultimately, the feature branch is merged into its parent branch and the feature is integrated into the system. However, this is also not an actual variability mechanism, as the feature is no longer variable after the merge. Some build systems allow the customization of the build process to a degree where it can be used to control what files and folders shall be included during compilation based on a feature selection. However, this customization needs to be done manually by the developers as there is no native feature concept.

Annotative Mechanisms. Annotative mechanisms are specific to the type of artifact (e.g., text files or UML diagrams) and require the developer to manually place annotations (usually feature conditions) on the fragments of the artifacts (e.g., lines of text or diagram elements). In the case of textual artifacts, existing tools such as preprocessors or template engines are commonly used. For C and C++, for instance, their built-in preprocessor is the most common variability mechanism used (see Listing 1).

Compositional Mechanisms and Modularization. Compositional mechanisms are based on modularization principles. They allow developers to modularize their system (e.g., based on features) and then compose the individual modules into a whole system. However, variation points need to be placed manually without providing operations for doing so. Examples of mechanisms in this category are Feature-Oriented Programming with tools such as AHEAD[1] [33, 34] or FeatureHouse[2] [35, 36]), Aspect-Oriented Programming [37] (which works especially well for crosscutting features) with tools such as AspectJ,[3] Delta-Oriented Programming [38] or Context-Oriented Programming [39].

2.3. Variation Control Systems

While the above mechanisms allow creating variable systems (e.g., making a system configurable based on features), the added complexity of using these mechanisms weighs fully on the developers (e.g., the placement and specification of feature conditions). Developers still need to edit the variable system directly and manually in whatever way the used mechanism requires. This is the major difference to VarCS as we defined them and also explains why we did not consider the presented mechanisms and techniques as within the scope of our study. While some VarCS make use of existing variability mechanisms internally, they usually hide them (at least partially) from the developer and provide higher-level operations for automatically modifying them without showing them to the developer. However, such VarCS have not received much attention yet, not in research, and certainly not in practice.

We study VarCS, which manage features, variants, and variation points in a fine-granular, 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 that filter irrelevant details of variable artifacts to facilitate their comprehension and to lower the cognitive complexity of editing them. Without tool support, developers are presented with all the code and variation points at once (see Listing 1), including those irrelevant to a given task. Developers also need to manually add or edit the variation points in the code. With a VarCS, developers can specify their intention [20], for example, to add a feature to a new variant. The VarCS then hides all irrelevant code for that intention and automatically updates the variation points according to the intention. As such, VarCS support the

[1] http://www.cs.utexas.edu/users/schwartz/ATS.html
[2] https://www.infosun.fim.uni-passau.de/spl/apel/fh/
[3] https://www.eclipse.org/aspectj/

evolution and maintenance of systems with many variants, including software product lines and highly configurable systems.

2.4. Running Example

As an illustrative example throughout this paper we discuss the evolution of the code excerpt shown in Listing 1, which was taken from the 3D-printer firmware Marlin (slightly adapted from its original).[4] The excerpt is related to file and folder sorting on secure digital (SD) cards. The developer wanted to add support for dynamic memory allocation (feature SDSORT_DYNAMIC_RAM) when sorting files and folders on memory cards. The final result is shown in Listing 2. The developers added the new feature manually by adding new code, adding additional preprocessor directives, and changing the conditions of existing directives. VarCS can support this task by providing special operations for such scenarios.

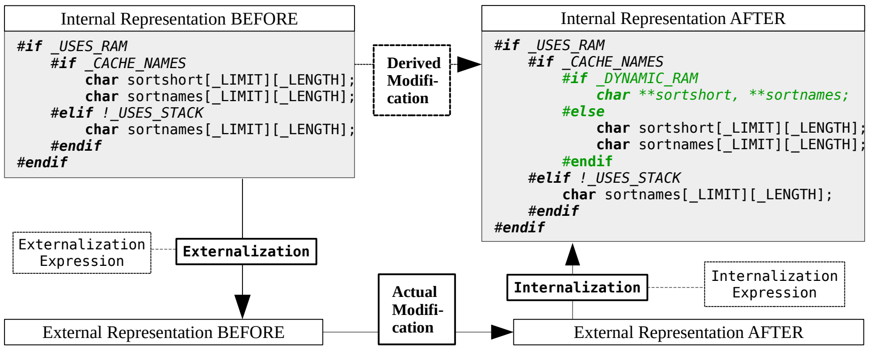
```
// By default the sort index is static
#if SDSORT_DYNAMIC_RAM
  uint8_t *sort_order;
#else
  uint8_t sort_order[SDSORT_LIMIT];
#endif

// Cache filenames to speed up SD menus.
#if SDSORT_USES_RAM

  // If using dynamic ram for names, allocate on the heap.
  #if SDSORT_CACHE_NAMES
    #if SDSORT_DYNAMIC_RAM
      char **sortshort, **sortnames;
    #else
      char sortshort[SDSORT_LIMIT][FILENAME_LENGTH];
      char sortnames[SDSORT_LIMIT][FILENAME_LENGTH];
    #endif
  #elif !SDSORT_USES_STACK
    char sortnames[SDSORT_LIMIT][FILENAME_LENGTH];
  #endif

  // Folder sorting uses an isDir array when caching items.
  #if HAS_FOLDER_SORTING
    #if SDSORT_DYNAMIC_RAM
      uint8_t *isDir;
    #elif SDSORT_CACHE_NAMES || !SDSORT_USES_STACK
      uint8_t isDir[(SDSORT_LIMIT+7)>>3];
    #endif
  #endif

#endif
```

Listing 2: Marlin code excerpt after implementation of the new feature SDSORT_DYNAMIC_RAM.

Figure 1 shows an iteration of the general workflow of VarCS. The *internal representation* stores the versioned artifacts and is usually not visible to the user. The *externalization* operation takes an *externalization expression* to generate an *external representation* the user can interact with and *modify*. Afterwards, the *internalization* operation modifies the *internal representation* according to a provided *internalization expression*. Every such editing cycle implies a *derived modification* that creates a new state of the internal representation. As obvious

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

from the figure, the derived modification affects both variability annotations and contents.


Internal Representation BEFORE
Derived Modification
Internal Representation AFTER
Externalization Expression
Externalization
Actual Modification
Internalization
Internalization Expression
External Representation BEFORE
External Representation AFTER

```
#if _USES_RAM
    #if _CACHE_NAMES
        char sortshort[_LIMIT][_LENGTH];
        char sortnames[_LIMIT][_LENGTH];
    #elif !_USES_STACK
        char sortnames[_LIMIT][_LENGTH];
    #endif
#endif
```

```
#if _USES_RAM
    #if _CACHE_NAMES
        #if _DYNAMIC_RAM
            char **sortshort, **sortnames;
        #else
            char sortshort[_LIMIT][_LENGTH];
            char sortnames[_LIMIT][_LENGTH];
        #endif
    #elif !_USES_STACK
        char sortnames[_LIMIT][_LENGTH];
    #endif
#endif
```

Figure 1: VarCS workflow shared by all subject systems. After the internalization (top right), the green text denotes modification through insertion.

The specific subject systems studied in this article differ with respect to the details of the workflow sketched in Fig. 1, which is why no concrete example of external and internal representations are shown. More precisely, we study the mechanisms provided for internalization and externalization, the properties of the expressions used for both operations, and the properties of the external representation the developer is exposed to during modification.

3. Methodology

Our research methodology comprised five main steps.

(1) Snowballing. Our aim was to study selected and relevant VarCS in depth. However, we intentionally did not conduct a full systematic literature review (e.g., following Kitchenham et al.'s method [40]). Based on our experience in software product lines and software configuration management, we sought existing survey papers [9, 41, 11, 42] and then used snowballing [43] 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 needed as some of the subject systems are only published in papers (some older than 30 years) and not accessible via digital libraries, making a systematic literature review or mapping study infeasible. A purely keyword-based search as done in systematic literature reviews would have been insufficient, since our target papers span many decades and are published in venues of very different quality, including technical reports, workshops, conferences, and journals. Nevertheless, we performed a retrospective search to exclude that we overlooked relevant literature (see step 3).

(2) Selection of Subject Systems. We inspected the publications about the collected subject systems and selected six for our comparison. Following our definition of VarCS, for a system to be in scope, it has to offer a workflow that can be mapped to the sequence of visible operations (*externalization*, *modification*, and *internalization*) in a transactional way, as shown in Fig. 1. When more than one generation of a VarCS existed, we chose the most mature version.

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. Obviously, contradicting our definition, we excluded typical version control systems (e.g., Concurrent Versions System, Subversion or Git). We also excluded subject systems for which the publications were no longer accessible or their description was too superficial for the purpose of our comparison.

(3) Thematic Synthesis of Characteristics for Comparison. To identify the concepts and characteristics that distinguish our subjects, we applied *thematic synthesis* [44], an approach that aims at identifying, interpreting, and explaining recurring themes from multiple studies. In particular we performed the following steps:

i. *Extract data.* We first annotated *excerpts* in the literature about VarCS that explain their internal and external data structures, concepts, and operations. For example, for VTS we recorded: "The operation *get* is used to checkout a particular working copy from the repository, and *put* is used to commit any changes back to the repository." [45]
ii. *Code data.* We then identified repeatedly occurring terms as well as terms highlighted in the literature as *codes*. For instance, from the above excerpt, we noted `get`, `checkout`, `working copy`, `repository`, `put`, `commit`, `changes`. We also consulted existing surveys [9, 41] and iteratively refined our set of codes.
iii. *Translate codes into characteristics.* Next, we compared the codes extracted for all subject systems and grouped them into *characteristics* based on their semantic properties. An example of a characteristics is `externalization operation`, which subsumes the codes `get`, `check[-]out`, `start transaction`, and `read`. This also included harmonizing the terminology, which was necessary as our subjects are from different research communities. Where adequate, we introduced a set of concrete values that a characteristic can take. For instance, the `externalization expression` is `partial` in P-EDIT and VTS, whereas it is `total` in the other subjects.
iv. *Group characteristics into categories.* Subsequently, we grouped characteristics into categories. An example is `externalization`, which groups, among others, the characteristics `externalization operation` and `externalization expression`.
v. *Assess the trustworthiness of the synthesis.* Finally, we ensured that the editing models explained in the literature for the different subject systems can be completely and unambiguously mapped to our catalog of characteristics.

Furthermore, we performed a retrospective Google Scholar search based on the codes collected in step (2) to ensure that the literature we included so far is accessible from these search terms, and to ensure we did not overlook any literature.

(4) Classification of Subject Systems. All authors then individually and independently assessed the subject systems using the characteristics and their possible values as a guideline. The individual classifications were then aligned and consolidated in a common classification with a common set of characteristics and their possible values, sufficient for characterizing systems in the domain of VarCS. The authors carefully discussed all cases of disagreement to reach consensus.

(5) Comparative Analyses. Finally, we performed detailed analyses to reveal essential differences between the VarCS in terms of the editing models and workflows of using the systems. First, we conducted a formal concept analysis [46, 47, 48] for deriving a concept hierarchy from the collection of subject systems and their properties to make clear their differences. Then we performed an in-depth scenario-based behavioural analysis to illustrate three subject systems from the perspective of the user of a VarCS.

4. Subject Systems

We now briefly introduce our subject systems before discussing their concepts and characteristics, and further illustrating them using our running example.

4.1. P-EDIT Editor

P-EDIT was presented in 1984 as a line-based editor for the VM/370 operating system [49] 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. It incorporates a Boolean formula simplifier for convenience. 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`) [50]. The author also motivates the approach by arguing that representing variants (called concurrent versions) as deltas [51], 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. The tool is not available anymore. 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 [50].

4.2. EPOS Version Control System

Expert System for Program "og" System Development (EPOS) supports change-oriented versioning, which aims at managing 'logical changes' of assets instead of managing whole versions of assets or systems [13, 52, 53]. It 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. The approach thus supports the intensional versioning paradigm (cf. Sec. 2.1), unlike most of today's version control systems. According to Munch [53], who provided the first implementation, the concept of change-oriented versioning was first described by Holager [54] in 1988 in a technical report, followed by Lie et al. [55, 56], who extended the concept and implemented change-oriented versioning using a database, which became the basis for Munch's prototype. The EPOS 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 framework [57], where it was again extended (e.g., with cooperation support [58]) and combined with process modeling and process execution techniques.

4.3. Leviathan File System

Software development tools are typically not variability-aware and cannot deal with variability mechanisms. Leviathan [59] addresses this problem by providing VarCS support at the file-system level. The Leviathan file system can be mounted by specifying a feature configuration (or a partial configuration where features are set as undecided), and thereby provides variant views for developers working on specific variants of a codebase. Leviathan allows using arbitrary tools without support for specific variability mechanisms, e.g., when debugging or maintaining different program variants. A typical development workflow 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 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 internally used preprocessor only supports 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 [45]. Specifically, it extends an approach called projectional editing of variational software [60] (not to be confused with the projectional-editing paradigm [61] for direct AST editing, a.k.a., structured editing or syntax-directed editing). Like its conceptual predecessor [60], VTS is formalized using the choice calculus [62], 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 change-oriented versioning. The prototype[5] 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 change-oriented versioning), which are then edited and committed back to the original file based on another expression (similar to the ambition expression of change-oriented versioning). 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.

4.5. ECCO Version Control System

ECCO[6] realizes a feature-oriented, distributed version control system. It started out as an approach to re-engineering variability from sets of cloned system variants [63, 64] 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 [65, 66]. 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 [67, 68].

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 (e.g., text files or Java source code) that can translate it into ECCO's internal tree structure. In case of file types for which no specific plugin is

[5] https://bitbucket.org/modelsteam/2016-vcs-marlin/src/master/prototype/
[6] https://github.com/jku-isse/ecco/

available, variability is only supported at the level of entire files (e.g., in case of binary files).

4.6. SuperMod Version Control System

SuperMod (Superimposition of Models)[7] [69, 70] 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 and available as a plugin to the Eclipse IDE. SuperMod and EPOS share conceptual properties due to their common ancestry, the change-oriented versioning paradigm [52], which in turn was specialized by the uniform version model [14]. There existed a preliminary implementation of the uniform version model based on EPOS, which is no longer available. SuperMod in turn is based on retrospective considerations of applying change-oriented versioning and the uniform version model approach to software product lines [71].

4.7. Excluded Subjects

We excluded several subject systems for which we could not find the publications anymore or the description lacked the necessary details. For instance, Conradi and Westfechtel [9] and Munch [53] refer to the editor MVPE [72]. However, since both authors report MVPE as an extension of P-EDIT, we believe that conceptually it does not differ too much from the P-EDIT system included in our comparison. Conradi and Westfechtel [9] also discuss PIE and DaSC. PIE [73] is probably the oldest system supporting fine-grained variability and was developed in the late 1970s [9]. DaSC [74, 11] 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 [75, 76] anymore. Atkins et al. [77] evaluate Labs' VE (version editor) [78, 79], which reportedly has similar functionality as

[7] http://www.ai1.uni-bayreuth.de/de/projects/SuperMod

P-EDIT. Specifically, it also creates variation points automatically. VE has its roots in the so-called Delta System [80], which was also not accessible.

5. Concepts of Variation Control Systems

We now describe the concepts and characteristics of VarCS 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)? What 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). Table 1 summarizes these characteristics.

We then explain how the selected VarCS create the external from the internal representation, and how changes made by the users in the external representation are integrated into the internal representation. This includes the alignment of changes in the external representation with the contents of the internal representation and the editing of constraints over features along with the implementation artifacts. Table 2 summarizes these characteristics.

We then proceed with describing the support offered for collaboration among developers. Finally, we discuss how each subject system was implemented and to what extent it was evaluated. Table 5 summarizes these characteristics.

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 (e.g., *Boolean*, *integer*, *enumeration*). They are user-visible and mapped to *variation points* in *variable artifacts* via a mapping.

For instance, P-EDIT uses *options* that 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.

When referring back to the example introduced in Sec. 2.4, the configuration options processed by the preprocessor are Boolean entities as shown in Fig. 2a, which can be toggled on or off during conditional compilation. Fig. 2b and Fig. 2c show examples of what integer and enumeration entities respectively could look like.

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,

Table 1: Characteristics Part 1: Concepts and Internal and External Representation.

			P-EDIT	EPOS	Leviathan	VTS	ECCO	SuperMod
Entities		Boolean	●	●	●	●	●	●
		Integer	●		●			
		Enumeration	●					
Constr.		None	●		●	●	●	
		Set of Constraints		●				
		Variability Model						●
Variable Artifacts		Files and Folders		●	●		●	●
	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		●	●	●	●	●
	State	Fixed	●	●	●	●	●	●
		Variable	●		●	●		

$\texttt{SDSORT\_USES\_STACK} \in \{true, false\}$
$\texttt{SDSORT\_CACHE\_NAMES} \in \{true, false\}$
$\texttt{HAS\_FOLDER\_SORTING} \in \{true, false\}$

(a) Boolean

$\texttt{SD\_MAXSIZE\_MB} \in \{128, \ldots, 8192\}$
$\texttt{USB\_STANDARD} \in \{1, 2, 3, 4\}$

(b) Integer

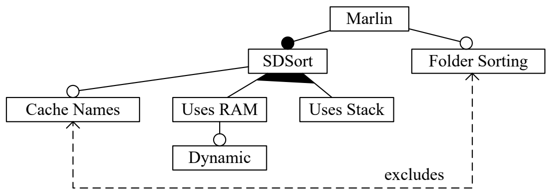
$\texttt{SDSORT\_RAM} \in \{NONE, STATIC, DYNAMIC\}$
$\texttt{SD\_VENDOR} \in \{INTEL, AMD\}$

(c) Enumeration

Figure 2: Different types of variability entities by example. Boolean entities were introduced in the running example, whereas integer and enumeration entities shown here do not appear in the original version history.

```
SDSORT_USES_RAM
|| SDSORT_USES_STACK

!SDSORT_DYNAMIC_RAM
|| SDSORT_USES_RAM


!(SDSORT_CACHE_NAMES
&& HAS_FOLDER_SORTING)
```

(a) Sets of constraints



(b) Variability model

Figure 3: Sets of constraints and variability model, referring to the running example. The dashed arrow in the feature model indicates a mutual exclusion between the adjacent features.

or feature constraints. The difference between two of the categories described below, namely sets of constraints and variability model, is sketched in Fig. 3.

None. Without constraints, every combination of options is allowed. 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.

Set of Constraints. A second option that systems might realize is logical constraints that must evaluate to true for a combination of options to be valid.

Variability Model. Feature models [24, 81, 82] and decision models [83] have been proposed for defining and managing the commonalities and variabilities in software product lines [84]. 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.

Four of our six subject systems do not support declaring constraints. EPOS allows specifying sets of constraints (Fig. 3a) among options, called rules (validities, constraints, preferences, defaults) [13]. Interestingly, these rules can also be used to define access rights in order to control which users have permission to access (i.e., read or externalize) or modify (i.e., write or internalize) which choices (i.e., variants). SuperMod uses feature models (Fig. 3b) to define logical variants and constraints.

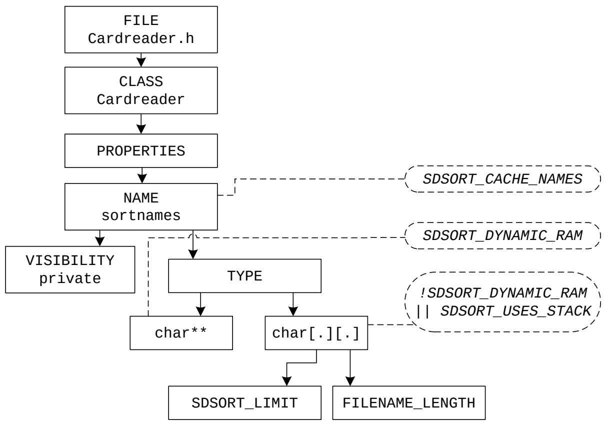
5.3. Variable Artifacts

This characteristic addresses the types and granularity 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 and variability at the level of folders and files. While variability at file-level is independent of the file type, we observe different file-format limitations and variation granularities for variability within files. Recall that we exclude the system level granularity according to our definition of VarCS.

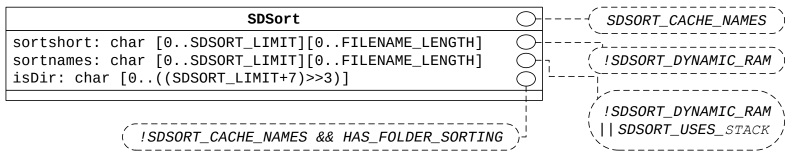
File-Level Granularity. Except for P-EDIT and VTS, all systems support file-level granularity for all kinds of files (including binary). P-EDIT and VTS only handle variability within text files and do not use a repository with folders and files.

```
  char **sortshort, **sortnames; | SDSORT_CACHE_NAMES && SDSORT_DYNAMIC_RAM
  char sortshort[SDSORT_LIMIT][FILENAME_LENGTH]; | SDSORT_CACHE_NAMES && !SDSORT_DYNAMIC_RAM
  char sortnames[SDSORT_LIMIT][FILENAME_LENGTH]; | SDSORT_CACHE_NAMES && !SDSORT_DYNAMIC_RAM
  char sortnames[SDSORT_LIMIT][FILENAME_LENGTH]; | SDSORT_CACHE_NAMES && SDSORT_USES_STACK
  uint8_t isDir[(SDSORT_LIMIT+7)>>3]; | !SDSORT_CACHE_NAMES && HAS_FOLDER_SORTING
```

(a) Text (line by line)



(b) Abstract syntax tree


SDSort
sortshort: char [0..SDSORT_LIMIT][0..FILENAME_LENGTH]
sortnames: char [0..SDSORT_LIMIT][0..FILENAME_LENGTH]
isDir: char [0..((SDSORT_LIMIT+7)>>3)]
SDSORT_CACHE_NAMES
!SDSORT_DYNAMIC_RAM
!SDSORT_DYNAMIC_RAM
|| SDSORT_USES_STACK
!SDSORT_CACHE_NAMES && HAS_FOLDER_SORTING

(c) Models (Ecore)

Figure 4: Sub-file artifact granularity by example. For both AST and model, we assume that annotated presence conditions are applied hierarchically to all sub-elements.

Sub-File-Level Granularity. All our VarCS support sub-file-level granularity, albeit at different levels of detail (see Fig. 4 for a side-by-side comparison). P-EDIT, Leviathan, VTS, and EPOS support variability in arbitrary text files, regardless of the actual text format (e.g., code in various programming languages, documentation, or help pages). The granularity is at the level of distinct 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)" [53]. 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. It also supports line-based variability in text files by internally transforming text files into models as well. ECCO can be extended with plugins to support variability for different types of files. The granularity of variations is determined by each respective plugin. Currently, plugins exist for whole files (a fallback for unknown 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 IEC languages [85, 86], C/C++ code, Ecore models or STEP files (3D CAD drawings) are under development.

5.4. Revisions

This characteristic addresses the support of the VarCS for revisions, i.e., versions intended to represent the sequential evolution, at different levels of granularity. The latter classifies into *system level* (to track evolution of the whole system), *feature level* (to track evolution of individual features), or *no* revisions at all (see Fig. 5).

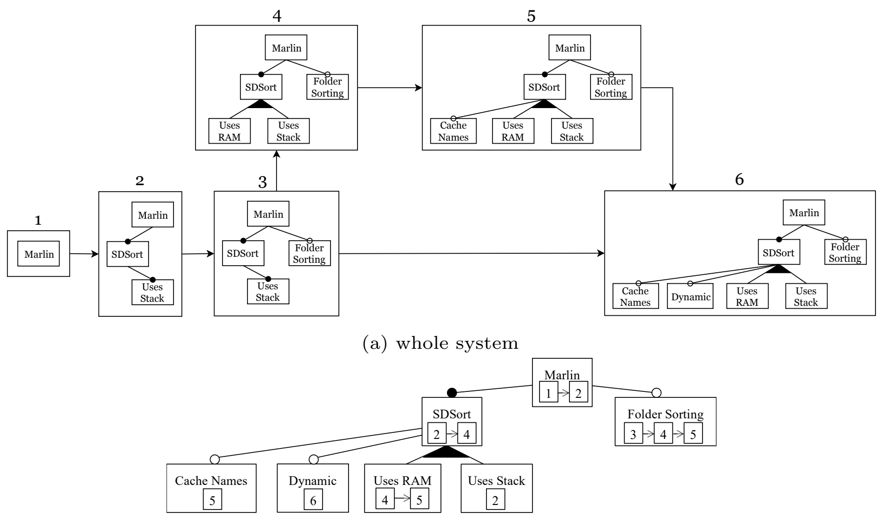
All systems except Leviathan and ECCO support revisions of the *whole system*. For instance, P-EDIT allows revisions of the whole system via artificial integer options (VERSION, TIME, RELEASE) that store the revision number. A similar approach would also work for VTS, although the system lacks dedicated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

5.5. Internal Representation

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

5.5.1. Artifact Storage

Our VarCS use different ways to store the artifacts and their variants: EPOS uses a database, while Leviathan, VTS, P-EDIT, and SuperMod rely on the file system for loading and saving artifacts. ECCO's plugin architecture allows adding custom storage mechanisms that can choose freely whether to use a database or simple files for persistence.



(a) whole system

(b) per feature

Figure 5: System vs. feature level revision graphs by example. The notation used in the per-feature example is inspired by *hyper feature models* [87].

5.5.2. Variation Points

All systems use presence conditions 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, thus, controlling if it should be included when obtaining an external representation. Concerning the manifestation of variation points, we can distinguish two approaches: The *annotative* way assumes that the internal storage superimposes all variants and attaches presence conditions to conditionally visible elements, whereas *modular* approaches assume a core variant, to which specific elements are conditionally added by composition.

P-EDIT uses *custom annotations* inside text files that contain variability. In a variable text file the presence conditions 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*[8] *directives* for conditional compilation (e.g., `#ifdef` or `#if` inside the text files) to directly annotate textual artifacts with presence conditions. Leviathan can use both C-preprocessor directives and M4[9] macros. SuperMod 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

[8]https://gcc.gnu.org/onlinedocs/cpp/
[9]https://www.gnu.org/software/m4/m4.html

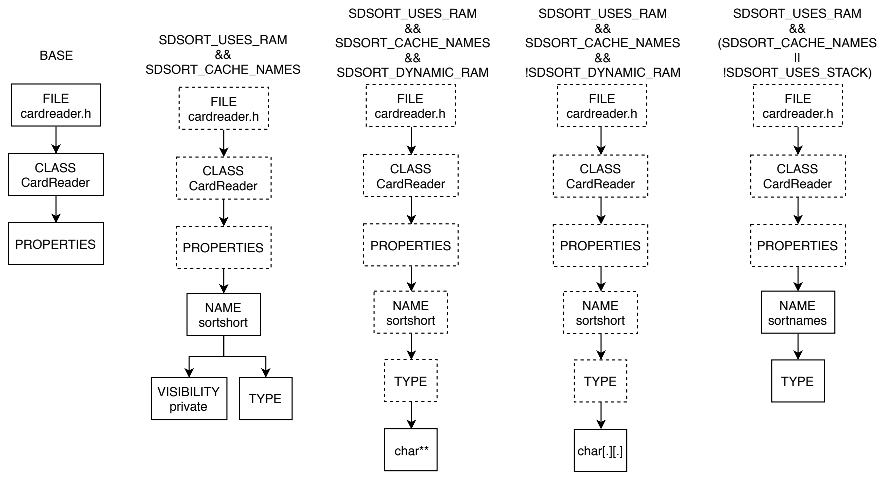


Figure 6: Modular internal representation of the source code (AST nodes) from the running example (as opposed to annotated internal representation shown in Fig. 4) as used by ECCO. Solid nodes are part of the respective module. Dashed nodes are not part of the module but necessary to preserve the location of solid child nodes.

visibility conditions [52]), 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.

ECCO has a custom data model resembling modules known from the paradigm of Feature-Oriented Software Development [88]. 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).

The running example in Fig. 1 uses standard C preprocessor directives to annotate the source code with presence conditions. Fig. 4 shows different internal representations (text, AST, model) annotated with custom presence conditions. For example, Fig. 4a shows text annotated with custom annotations to the right as P-EDIT does it. Fig. 6 depicts a decomposition of parts of the source code of the running example into *modules* as internally used by ECCO. It depicts the same AST as is shown with annotations in Fig. 4b decomposed into modules.

5.6. External Representation

This characteristic describes the interface the VarCS offers 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.

5.6.1. Type

A core characteristic is whether the external representation is *materialized* (that is, copied to a file system where it can be edited like ordinary files) or *virtual* (that is, only shown in a custom editor via which it must be edited).

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

5.6.2. State

The *state* of the external representation can either be *fixed* or *variable*. It is *fixed* if it represents a single, concrete variant without any remaining variability (i.e., every feature is either included or excluded). A *fixed* external representation does not require a variability mechanism. It is *variable* if it contains remaining variability and thus represents a set of variants. This is the case when not every feature has a value assigned (i.e., some features are undecided). This requires a variability mechanism (e.g., preprocessor annotations) in the external representation that the user can edit directly.

All systems support *fixed* external representations. P-EDIT, Leviathan, and VTS also support *variable* external representations. P-EDIT uses its custom annotations and color highlighting to mark fixed and variable parts of the code for the user. Leviathan and VTS simply leave ordinary preprocessor annotations (which they use anyway in their internal representation) in the code to mark variable parts. These annotations can be edited directly by the user.

5.7. Editing of Constraints

This property reflects if and how the subject systems support editing of constraints over the variability entities. Fig. 7 summarizes the viable strategies as state charts. Constraints editing may either be applicable to the internal representation (*internalized* constraints editing) or embedded into the externalization/internalization cycle (*externalized* constraints editing).

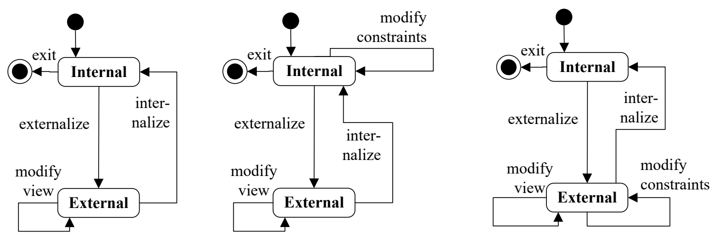
P-EDIT, Leviathan, VTS, and ECCO do not support version constraints; therefore, their editing is not applicable. In EPOS, both the set of *options* (Boolean entities) and the set of constraints is available for modification in between externalization/internalization cycles, when no view is active, such that they are considered for the subsequent cycle. SuperMod offers externalized editing of the variability model, allowing to define and realize a feature in one and the same cycle. This introduces new consistency problems referring to the relationship between, e.g., the variability model and the IE themselves [89].

5.8. Externalization

The externalization operation retrieves an external representation from the internal one.

Table 2: Characteristics Part 2: Mapping between Internal and External Representation.

			P-EDIT	EPOS	Leviathan	VTS	ECCO	SuperMod
Constr. Editing	N/A		●		●	●	●	
	Internalized			●				
	Externalized							●
Externalization	Expression	Arbitrary	●			●		
		Conjunction	●	●	●	●	●	●
		Partial	●		●	●		
		Full	●	●	●	●	●	●
	Specif.	Manual	●	●	●	●	●	●
		Assisted	●					●
		Automatic	●					
	Artifacts Consistency						●	●
Internalization	Expression	Arbitrary	●			●		
		Conjunction	●	●	●	●	●	●
		Partial	●	●	●	●		●
		Full	●	●	●	●	●	●
	Relation with EE	Weaker		●				●
		Same	●	●	●	●		●
		Stronger				●		
	Representativity Check					●	●	●
Alignment	Not Necessary		●			●		
	Textual			●	●		●	
	Structural						●	●



(a) None (b) Internalized (c) Externalized

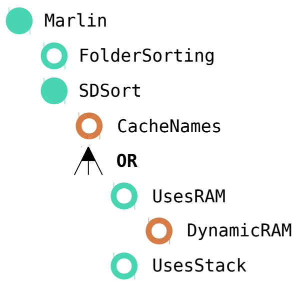
Figure 7: Viable strategies for integrating the editing of constraints into the VarCS workflow.

`defined(SDSORT_USES_STACK) &&`
`!defined(SDSORT_CACHE_NAMES)`

(a) Projection expression (VTS)

`VERSION > 4 && SD_MAXSIZE_MB > 1024 &&`
`!SDSORT_USES_STACK`

(b) Mask (P-EDIT)



```
● Marlin
  ○ FolderSorting
  ● SDSort
      ○ CacheNames
      OR
        ○ UsesRAM
            ○ DynamicRAM
        ○ UsesStack
```

(c) Feature configuration (SuperMod)

Figure 8: Running example: specification of different variants as offered by different subject systems. In the screenshot in Fig. 8c, filled circles represent mandatory, empty circles optional features; cyan denotes positive, and magenta negative selection.

Table 3: Examples of externalization expressions with different characteristics represented in preprocessor-like syntax.

	All variables	`SDSORT_CACHE_NAMES`, `SDSORT_DYNAMIC_RAM`, `SDSORT_USES_STACK`
Arbitrary	Partial	`SDSORT_CACHE_NAMES || !SDSORT_USES_STACK`
	Full	`(SDSORT_CACHE_NAMES || SDSORT_USES_STACK) && !SDSORT_DYNAMIC_RAM`
Conjunction	Partial	`!SDSORT_CACHE_NAMES && SDSORT_DYNAMIC_RAM`
	Full	`SDSORT_CACHE_NAMES && SDSORT_DYNAMIC_RAM && !SDSORT_USES_STACK`

5.8.1. Externalization Expression (EE)

The *EE* refers to version entities and determines which variable artifacts, or which parts thereof, are presented externally to the user. We identified two distinguishing characteristics, indicating whether *partial* expressions are supported, and whether the expression is necessarily a *conjunction* of version entities.

The EE is defined over the variability entities. It is *fully* specified if all entities are decided. It is *partially* specified if variability is still left undecided in the external representation. Figure 8 accompanies the textual descriptions given in the following. Different terms are used to denote the EE in the VarCS and the syntactic rules slightly differ. Leviathan calls it *variant*, EPOS and SuperMod call it *choice*, VTS calls it *projection expression* (cf. example in Fig. 8a), ECCO uses the term *configuration*. P-EDIT refers to the EE as *mask* and allows arithmetic comparison operators which can be used for selecting versions (Fig. 8b).

Table 3 provides four concrete examples of EEs covering all combinations of cases: *arbitrary expressions* or *conjunctions* (i.e., configurations) each as *partial* or *full* expressions.

Arbitrary Expression. In the most general case, the EE is instance of ordinary propositional logic over the version entities; the user may freely define this expression, e.g., based on text input. When creating the external representation, all elements whose presence conditions imply the expression are externalized. VTS accepts an arbitrary logical expression. P-EDIT allows for arbitrary expressions over the Boolean, integer, or enumeration entities.

Conjunction. EPOS, Leviathan, ECCO, and SuperMod are more restrictive and require the EE to be a conjunction of entities. In the case of Boolean-valued entities, such a conjunction can be straightforwardly derived from a configuration provided by the user. A configuration is an assignment of the values *true* or *false* to every entity, which is essentially a *conjunction* of positive or negative references to the entities.

Partial. Leviathan, VTS and P-EDIT support *partial expressions*, 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. 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" [49]). 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 [49]). 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 [49]). 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 projection expression, then the artifact is shown as ordinary text not wrapped by conditional compilation directives.

Full. The internal complexity of many subject VarCS is significantly reduced by requiring the EE to be a full configuration where every entity needs a value assigned. EPOS, ECCO, 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 and represents a single, concrete variant.

5.8.2. Specification

Some of the subjects *assist* the user in specification of the EE. In EPOS, *preferences* and *defaults* may contain derivation rules for undecided configuration options. SuperMod, on the one hand, provides graphical support by allowing to manually create a configuration expression by selecting features in the graphical feature model (Fig. 8c). On the other hand, feature selection is propagated in certain situations, e.g., negative selections are applied hierarchically to child features.

Interestingly, P-EDIT is the only subject system that does not require the users to specify the 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" [50]. P-EDIT in this way specifies the EE *automatically* by generating it from the users' artifact selections.

5.8.3. Artifacts Consistency

Even for subject systems that support constraints over version entities, their successful validation is not a necessary precondition for the syntactical (nor semantical) consistency of concrete variants (external representations). For instance, the *optional feature dilemma* [90] may occur when conflictingly combining two features/options that have been realized in isolation, or problems between declarations and applications of artifacts can appear.

A VarCS could check the consistency upon obtaining the external representation. However, most VarCS aim at being oblivious to the underlying artifact formats and do not perform such consistency checks. Exceptions are ECCO, which can use the knowledge available in file-type-specific plugins for simple checks, and SuperMod which can validate its models against their respective meta-models (without taking context-sensitive rules, e.g., OCL constraints, into account).

5.9. Internalization

The internalization operation refines an internal representation based on an external representation.

5.9.1. Internalization Expression (IE)

All subject VarCS provide an *IE* for delineating the variant(s) to which the change performed in the external view shall be made effective internally. We here discuss the same properties as for the EE (*arbitrary expression* or *conjunction* and *partial or full*), before we discuss the relationship between IE and EE.

EPOS, VTS, and SuperMod call the IE "ambition," obviously inspired by the change-oriented versioning paradigm.

Conjunction. Although there is no dependency in theory, here, subjects requiring the EE to be a conjunction (EPOS, Leviathan, ECCO, and SuperMod) require the same property to hold for the IE. The motivation to apply this restriction is analogous: a conjunction can be derived easily from a (partial) configuration of version entities.

Partial. Five of the six subject systems (P-EDIT, EPOS, Leviathan, VTS, and SuperMod) allow that the IE be *partial*, i.e., it may contain unresolved configuration decisions and therefore refer to a set of variants rather than to a single variant. Based on this selection, the presence conditions of elements modified in the external view are updated internally.

Full. ECCO represents an exception here as the IE is required to be a *full configuration*. This means that only concrete variants can be internalized. The IE (the variant's configuration) is used together with the artifacts in the external representation (the variant's implementation) to automatically refine presence conditions for artifacts in the internal representation. Every variant that is internalized allows ECCO to incrementally refine presence conditions of artifacts that can also affect other variants. Therefore, although no partial configuration is allowed, a single internalization operation can affect multiple variants. The major difference to the other subject systems is, that the developer does not

Table 4: Relationship between externalization expression and internalization expression by example.

	Externalization Expression	Viable Internalization Expressions
Weaker	`SDSORT_CACHE_NAMES && !SDSORT_DYNAMIC_RAM`	`SDSORT_CACHE_NAMES`
		`!SDSORT_DYNAMIC_RAM`
Same	`SDSORT_CACHE_NAMES && !SDSORT_DYNAMIC_RAM`	`SDSORT_CACHE_NAMES && !SDSORT_DYNAMIC_RAM`
	`SDSORT_USES_STACK`	`SDSORT_USES_STACK`
Stronger	`SDSORT_USES_STACK`	`SDSORT_USES_STACK && SDSORT_CACHE_NAMES`
		`SDSORT_USES_STACK && !SDSORT_DYNAMIC_RAM`

directly specify which variants are to be affected in what way. Instead, ECCO decides what artifacts need to be affected in what way in order to reflect the new information learned from the new variant.

5.9.2. Relationship with Externalization 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 (see Figure 1): externalization (i.e., creating a view representing one or multiple variants), modification (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 formal relation to the EE. Table 4 provides examples for the viable strategies *same*, *stronger*, and *weaker*, presented subsequently.

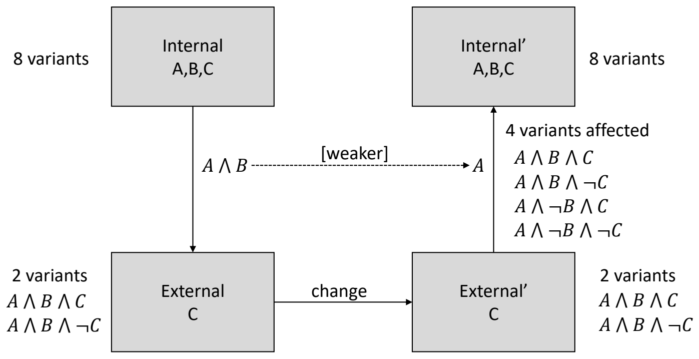
Same. In P-EDIT and Leviathan, the IE, which represents the scope of changes, cannot be set by the user. It is the same as the EE and therefore determined during externalization. Changing it requires a new cycle in the VarCS workflow. Please notice that this strategy can only be applied in a meaningful way as long as partial IEs/EEs are allowed.

Stronger. For VTS the IE can only be stronger or the same as the EE, which is a limitation. *Stronger* means that the configuration decisions made in the IE must include all corresponding decisions previously fixed in the EE ($IE \Rightarrow EE$ in the propositional logical sense).

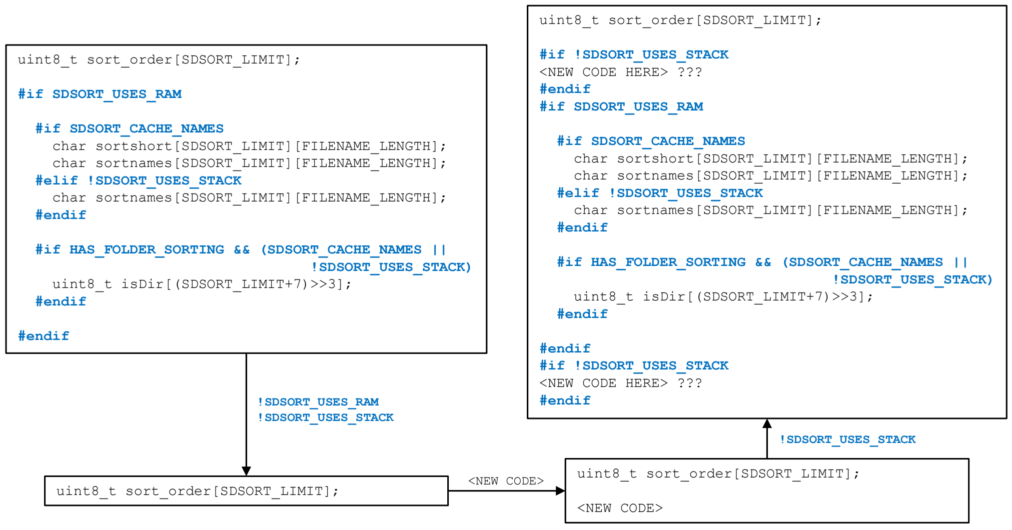
Weaker. For EPOS and SuperMod the IE can be weaker or the same as the EE, where *weaker* denotes a subset of the configuration decision ($EE \Rightarrow IE$). Being able to specify a weaker expression, which makes it possible to apply changes to variants that are not visible in the view, can cause, e.g., representativity or alignment issues (see below).

No relation. ECCO allows arbitrary IEs regardless of the previously used EE, both need to be full configurations though.

Figure 9 illustrates the consequences of an IE that is weaker than the EE. The changes affect more variants than were visible while applying them as shown in Figure 9a. In other words, the changes made to the external representation affect parts of the internal representation that were not visible when the changes were made. This can cause problems when merging the external representation



(a) abstract example



```
uint8_t sort_order[SDSORT_LIMIT];

#if SDSORT_USES_RAM

  #if SDSORT_CACHE_NAMES
    char sortshort[SDSORT_LIMIT][FILENAME_LENGTH];
    char sortnames[SDSORT_LIMIT][FILENAME_LENGTH];
  #elif !SDSORT_USES_STACK
    char sortnames[SDSORT_LIMIT][FILENAME_LENGTH];
  #endif

  #if HAS_FOLDER_SORTING && (SDSORT_CACHE_NAMES ||
                              !SDSORT_USES_STACK)
    uint8_t isDir[(SDSORT_LIMIT+7)>>3];
  #endif

#endif
```

```
uint8_t sort_order[SDSORT_LIMIT];

#if !SDSORT_USES_STACK
<NEW CODE HERE> ???
#endif
#if SDSORT_USES_RAM

  #if SDSORT_CACHE_NAMES
    char sortshort[SDSORT_LIMIT][FILENAME_LENGTH];
    char sortnames[SDSORT_LIMIT][FILENAME_LENGTH];
  #elif !SDSORT_USES_STACK
    char sortnames[SDSORT_LIMIT][FILENAME_LENGTH];
  #endif

  #if HAS_FOLDER_SORTING && (SDSORT_CACHE_NAMES ||
                              !SDSORT_USES_STACK)
    uint8_t isDir[(SDSORT_LIMIT+7)>>3];
  #endif

#endif
#if !SDSORT_USES_STACK
<NEW CODE HERE> ???
#endif
```

```
uint8_t sort_order[SDSORT_LIMIT];
```

```
uint8_t sort_order[SDSORT_LIMIT];

<NEW CODE>
```

(b) concrete example also illustrating alignment problem

Figure 9: Illustration of consequences of internalization expression that is weaker than externalization expression.

into the internal representation as multiple alignments may be possible as shown in Figure 9b.

5.9.3. *Representativity Check*

VarCS let the user *representatively* modify a variant or a set of variants, while the changes are made effective for a set of variants that may be larger (see *weaker* in the item above) or smaller (*stronger*). Here, the term *representativity* tacitly implies that the change performed in a specific view could have been equally applied in all other views that are affected by the change. Changes that do not meet this condition may create different kinds of inconsistencies. The subject systems differ in how this condition is checked and/or enforced.

This characteristic is not relevant for P-EDIT and Leviathan, since the IE and the EE are equal for those.

VTS has been designed around the *edit isolation principle* and therefore applies representativity checks by construction. When updating the presence condition of modified artifacts, “the only variants that change in the source are those that can be reached from the view” [60], where “source” denotes the internal representation.

SuperMod checks, but does not ultimately enforce, representativity. If, during internalization (*commit*) a change is detected which affects elements that are not available in all variants represented by the IE (*ambition*), a corresponding warning is shown to the user, who may either accept the risk of creating inconsistencies, or select a stronger ambition that satisfies the condition.

ECCO deals with the issue of representativity differently. The IE is not used directly as presence condition but merely represents the configuration of the current variant. The presence conditions are automatically derived based on a comparison between the current internal representation and the current external representation and the provided configuration. ECCO makes no assumptions and only guarantees that the exact variant represented by the provided configuration works as intended (extensional versioning). Any new variant (e.g., containing new features or new combinations of existing features) that has never been internalized before is constructed on a best effort basis (e.g., existing feature and feature interaction implementations known from previous variants are reused) and accompanied by *warnings* that point out that some of the individual artifacts that are relevant for the new variant might not (yet) work in combination (*intensional versioning*).

5.10. *Alignment Strategy*

This characteristic describes the strategies the VarCS employ for aligning changes in the external representation with the existing artifacts in the internal representation (which potentially belong to different variants) upon internalization as is illustrated in Figure 9b.

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 incorrectly syntax or semantics could be violated (e.g., statements could be put in the wrong order). Such alignment problems can occur when the externalization is hiding artifacts that are not contradicted by the IE used to insert artifacts at the same location where the hidden artifacts would be. This can happen in any VarCS that does not enforce IEs that are at least as strong as the EE that was used to produce the current view.

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 ordered nodes on the same tree level. SuperMod is based on Ecore and uses, similarly to ECCO, tree-based merging and for ordered sequences a directed graph. Albeit, the tools differ in the matching of source code artifacts: SuperMod performs a line-oriented alignment, whereas ECCO relies on the abstract syntax tree for fine-grained alignment. For EPOS 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 EE are hidden. Because both systems enforce that the internalization cannot be weaker than the externalization 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).

5.11. Collaboration

A subset of the systems investigated explicitly supports multi-user editing. We classify collaboration along two dimensions.

Paradigm. 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 does not support collaboration either, 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.

Synchronization. Traditional version control systems support multi-user collaboration based on two distinct paradigms: optimistic (i.e., modify-merge) or pessimistic (i.e., lock-modify-unlock).

When transferred to the here considered VarCS, the distinction becomes relevant only for those subjects that support collaboration; therefore, neither P-EDIT nor Leviathan nor VTS require synchronization. Apart from these, EPOS mandatorily requires locks, either at product-level or at ambition-level granularity, which ensure that the same artifacts are not modified concurrently by different transactions. SuperMod follows a two-step optimistic synchronization process. First, overlapping changes are merged non-interactively. In a second, interactive

Table 5: Characteristics Part 3: Tool Support and Validation.

			P-EDIT	EPOS	Leviathan	VTS	ECCO	SuperMod
Collaboration	Paradigm	None/Local	●		●	●		
		Centralized		●				
		Distributed					●	●
	Synchro-nization	N/A	●		●	●		
		Pessimistic		●				
		Optimistic					●	●
Implementation	Modality	Editor	●					
		Version Control		●		●	●	●
		File System			●		●	
	Inter-face	Textual	●	●		●		
		Graphical					●	●
	Intru-sive-ness	Internal	●	●			●	●
		External	●					
	Avail.	Binary				●	●	●
		Source					●	●
Evaluation	Exemplary			●		●		●
	Qualitative					●		●
	Quantitative						●	
	Formal							

step, the user may revert merge decisions based on a single-version view. ECCO requires no synchronization as it defers merging to the time of externalization. As ECCO persists revisions per feature it simply stores concurrent changes to the same feature as two parallel revisions of that feature and only requires that they are merged once the EE explicitly expresses that.

5.12. 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. Sec. 5.9 and Figure 1).

User Interface. We distinguish textual and graphical user interfaces. All subject systems except SuperMod have a textual interface in form of a command-line interface 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.

Intrusiveness. This characteristic describes to which extent the subject systems integrate into the development process and enviroment. Highly intrusive tools force their users to use customized (but also optimized) operations and user interfaces most operations. Non-intrusive tools integrate better with foreign editing tools and make less assumptions about the development environment. The characateristic is applicable to the internal (e.g., constraints editing) and external (e.g., editing of artifacts) perspective. 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). 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. 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. ECCO is non-intrusive on the external representation as the retrieved files can be edited with existing tools. The internal representation is not intended to be edited directly, however, depending on the used storage plugin, it could use 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, arbitrary 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 and binaries are publicly available on GitHub. SuperMod is available as a binary plugin to the Eclipse IDE; the server-side application is provided as Tomcat web application. The source code of both components of SuperMod is hosted on GitHub. The VTS prototype is available as a binary. For all other subject systems the implementation was not available.

5.13. 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. Sec. 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 [91, 92]. 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. SuperMod's qualitative evaluation based on three academic case studies is presented in [93].

6. Essential Differences Between the Subject Systems

Based on our classification and illustration of VarCS discussed above, we present *essential differences* between the subject systems. First, we provide results of a formal concept analysis [46, 47, 48] to improve the clarity of the differences related to the editing models and workflows of using the systems. Second, we perform a scenario-based behaviour analysis to illustrate three subject systems from the perspective of a software engineer, i.e., the primary user of a VarCS.

6.1. Formal Concept Analysis of Editing Models

The properties discussed in Sec. 5 are connected to design decisions made by the originators of the respective tools. Many of these properties reflect optional building blocks that can be transferred to the other systems with reasonable effort. For instance, the usage of a variability model for the higher-level representation of constraints and entities provides an advantage for the end user, but does not significantly affect the theoretic capabilities of the models.

The other properties, however, constitute mutually exclusive subject-level variation points that must be carefully discussed in terms of advantages and disadvantages to the end user. In this regard, we extracted an essential set of attributes that are functionally relevant to the *editing model* of the respective systems, i.e., decisive characteristics that substantially affect the workflow presented in Fig. 1. These characteristics are concerned with the *internalization expression (IE)* and *externalization expression (EE)* and their relation:

IE/EE Conjunction. The IEs and EEs are *conjunctions* as opposed to arbitrary Boolean expressions. On the one hand, requiring version specifications to be stated as conjunctive Boolean terms seems to constrain version construction. On the other hand, such expressions are much easier to handle for the user, e.g., when being derived from sets of configuration options or from feature configurations.

EE Partial. The EE can be *partial.* This property is decisive because it dictates whether the external view contains variation points (e.g, `#ifdef` directives in textual approaches) or whether it presents a single variant that is edited in a representative way (which is indispensable in model-driven approaches like SuperMod because there exists no generic multi-variant syntax resembling the `#ifdef` approach).

IE Partial. The IE can be *partial.* When this attribute is active, the modified contents to be internalized are in general associated with a set of variants rather than with a single variant specification. In the case of a total IE (EPOS), the presence conditions of modified elements are not updated based on the IE itself, but based on the differences between EE and IE.

IE $<$ EE. The IE can be *weaker* than the EE.

IE $=$ EE. The IE and the EE can be the *same.*

IE $>$ EE. The IE can be *stronger* than the EE.

The last three properties are decisive inasmuch as they impose additional constraints onto the relationship between EE and IE. Depending on the allowed relationships, editing cycles must be structured in a significantly different way when comparing the different subject systems with each other (see illustration in Sec. 6.2). In some scenarios, e.g., the minimum number of editing cycles necessary to realize a specific change may vary.

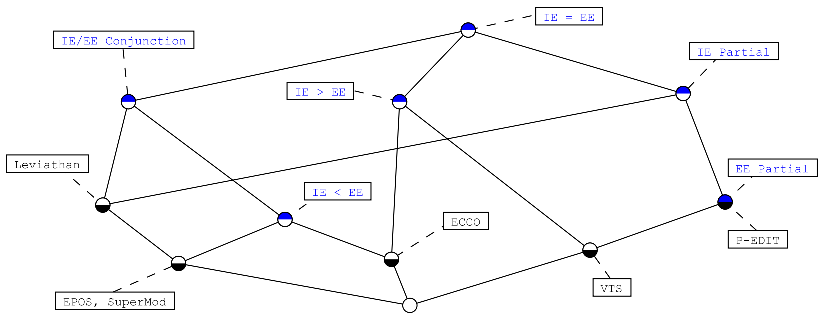


Figure 10: Concept lattice for VarCS properties that immediately reflect the editing model, obtained by formal concept analysis.

With a *formal concept analysis* (FCA) [46, 47, 48] we aligned these six Boolean attributes with the subject systems. FCA is an algebraic theory for binary relations which identifies all maximal rectangles in the table. We defined a cross table containing the attributes and the subject systems (object in FCA terminology) that imply them. The rectangles form a hierarchy that can be displayed as a so-called *concept lattice*, which we display in Fig. 10 for the FCA performed here.

From the lattice, we can learn two things, namely how closely the subject systems are related and by which decisive properties they are distinguished. Second, we can deduce implication relationships between attributes. For instance, EPOS and SuperMod have exactly the same valuation for the considered attributes, which is presumably due to their common ancestry (change-oriented versioning and unified version model). All subject systems allow to work with the same IE and EE, but only P-EDIT and Leviathan strictly require to do so. VTS differs from the other systems as it is the only system that allows the IE to be stronger than the EE; and ECCO is also unique as it allows for an arbitrary relationship between IE and EE as well as by requiring both the IE and the EE to be total (i.e., full configurations). The attribute *EE Partial* implies *IE Partial*.

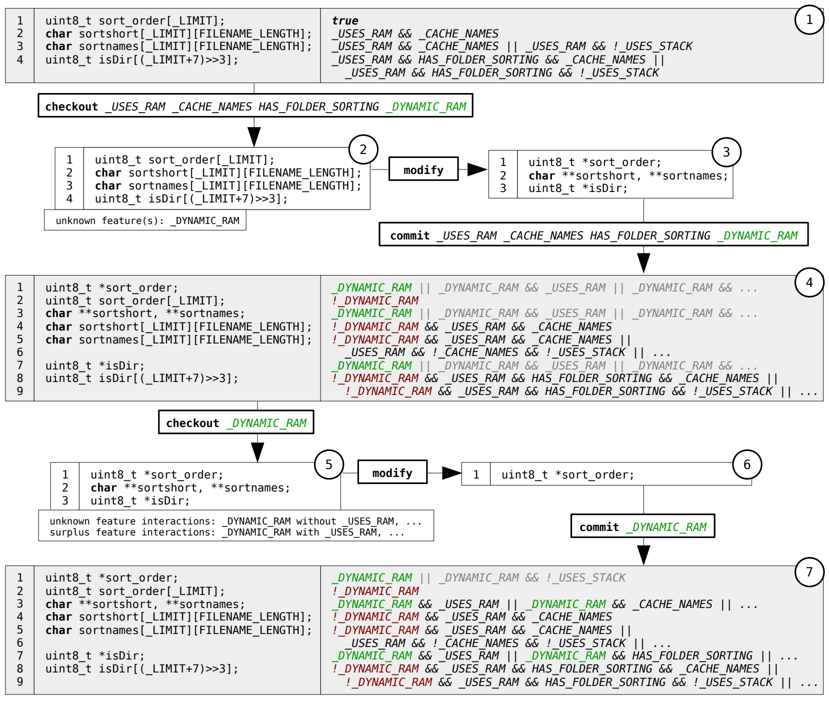
To sum up, although the considered subject systems share a common workflow pattern (see Fig. 1), there are fundamental semantic differences in the interpretation of IE and EE. It is hard to select among the subject systems one editing model that "rules them all"; rather, they all imply their individual properties that emerge from specific design decisions and drastically affect how the user works with the respective tools.

6.2. Scenario-Based Behavioral Analysis

We performed a scenario-based evaluation using the running example from Sec. 2.4. Our analysis covers the three most recent and according to Sec. 6.1 also most diverse subject systems for which a running implementation could be obtained: ECCO, SuperMod and VTS. We show differences of the editing model and workflow from the perspective of the software engineering working with the different systems.

6.2.1. ECCO

ECCO requires a full configuration as an IE or EE. Consequently, developers always work on concrete, fully configured variants without any remaining variability. Developers can focus solely on the concrete variant they decided to work on and do not need to worry about other variants or exact presence conditions of every line of code they add, modify or delete. ECCO automatically computes the presence conditions from the provided IE (full configurations) and automatically adds or refines them where necessary.



1
checkout _USES_RAM _CACHE_NAMES HAS_FOLDER_SORTING _DYNAMIC_RAM
2
modify
3
commit _USES_RAM _CACHE_NAMES HAS_FOLDER_SORTING _DYNAMIC_RAM
4
checkout _DYNAMIC_RAM
5
modify
6
commit _DYNAMIC_RAM
7

```
uint8_t sort_order[_LIMIT];                      true
char sortshort[_LIMIT][FILENAME_LENGTH];         _USES_RAM && _CACHE_NAMES
char sortnames[_LIMIT][FILENAME_LENGTH];         _USES_RAM && _CACHE_NAMES || _USES_RAM && !_USES_STACK
uint8_t isDir[(_LIMIT+7)>>3];                    _USES_RAM && HAS_FOLDER_SORTING && _CACHE_NAMES ||
                                                   _USES_RAM && HAS_FOLDER_SORTING && !_USES_STACK
```

```
uint8_t sort_order[_LIMIT];
char sortshort[_LIMIT][FILENAME_LENGTH];
char sortnames[_LIMIT][FILENAME_LENGTH];
uint8_t isDir[(_LIMIT+7)>>3];
```

```
unknown feature(s): _DYNAMIC_RAM
```

```
uint8_t *sort_order;
char **sortshort, **sortnames;
uint8_t *isDir;
```

```
uint8_t *sort_order;                             _DYNAMIC_RAM || _DYNAMIC_RAM && _USES_RAM || _DYNAMIC_RAM && ...
uint8_t sort_order[_LIMIT];                      !_DYNAMIC_RAM
char **sortshort, **sortnames;                   _DYNAMIC_RAM || _DYNAMIC_RAM && _USES_RAM || _DYNAMIC_RAM && ...
char sortshort[_LIMIT][FILENAME_LENGTH];         !_DYNAMIC_RAM && _USES_RAM && _CACHE_NAMES
char sortnames[_LIMIT][FILENAME_LENGTH];         !_DYNAMIC_RAM && _USES_RAM && _CACHE_NAMES ||
                                                   _USES_RAM && !_CACHE_NAMES && !_USES_STACK || ...
uint8_t *isDir;                                  _DYNAMIC_RAM || _DYNAMIC_RAM && _USES_RAM || _DYNAMIC_RAM && ...
uint8_t isDir[(_LIMIT+7)>>3];                    !_DYNAMIC_RAM && _USES_RAM && HAS_FOLDER_SORTING && _CACHE_NAMES ||
                                                   !_DYNAMIC_RAM && _USES_RAM && HAS_FOLDER_SORTING && !_USES_STACK || ...
```

```
uint8_t *sort_order;
char **sortshort, **sortnames;
uint8_t *isDir;
```

```
unknown feature interactions: _DYNAMIC_RAM without _USES_RAM, ...
surplus feature interactions: _DYNAMIC_RAM with _USES_RAM, ...
```

```
uint8_t *sort_order;
```

```
uint8_t *sort_order;                             _DYNAMIC_RAM || _DYNAMIC_RAM && !_USES_STACK
uint8_t sort_order[_LIMIT];                      !_DYNAMIC_RAM
char **sortshort, **sortnames;                   _DYNAMIC_RAM && _USES_RAM || _DYNAMIC_RAM && _CACHE_NAMES || ...
char sortshort[_LIMIT][FILENAME_LENGTH];         !_DYNAMIC_RAM && _USES_RAM && _CACHE_NAMES
char sortnames[_LIMIT][FILENAME_LENGTH];         !_DYNAMIC_RAM && _USES_RAM && _CACHE_NAMES ||
                                                   _USES_RAM && !_CACHE_NAMES && !_USES_STACK || ...
uint8_t *isDir;                                  _DYNAMIC_RAM && _USES_RAM || _DYNAMIC_RAM && HAS_FOLDER_SORTING || ...
uint8_t isDir[(_LIMIT+7)>>3];                    !_DYNAMIC_RAM && _USES_RAM && HAS_FOLDER_SORTING && _CACHE_NAMES ||
                                                   !_DYNAMIC_RAM && _USES_RAM && HAS_FOLDER_SORTING && !_USES_STACK || ...
```

Figure 11: The running example of Sec. 2.4 performed with ECCO. The internal representations (i.e., repository contents) are depicted with grey background and in a simplified form (textually with presence conditions annotated on the right) instead of the actual, more complex form (modular trees, cf. Fig. 6). The external representations (i.e., workspace contents) have a white background. User actions are placed along arrows and represented with a bold border. Configurations are given as a comma separated list of selected (+) or deselected (-) features. For brevity, the prefix SDSORT_ is replaced by a simple _.

Figure 11 shows the running example of Sec. 2.4 performed with ECCO:

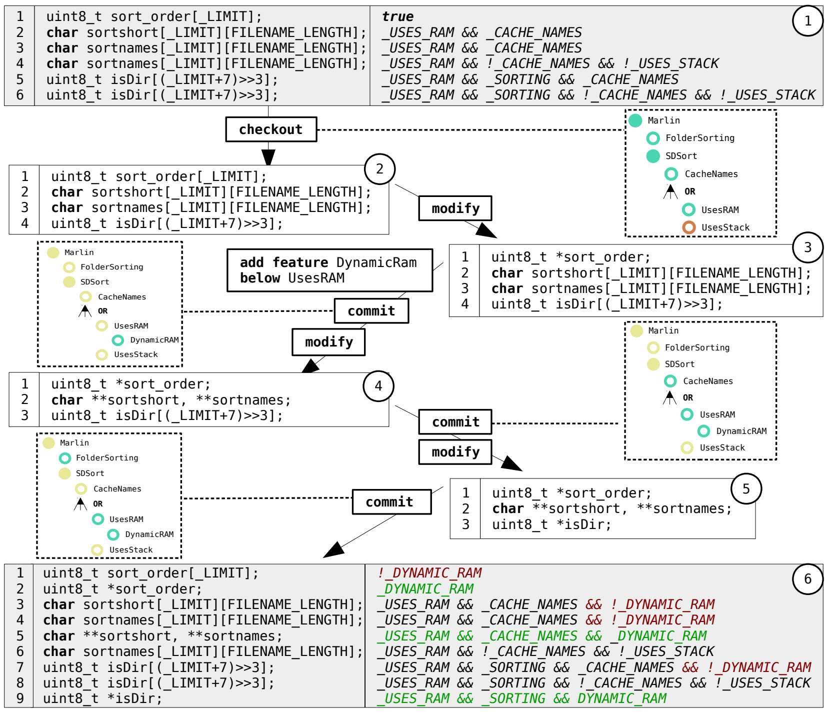
1. The internal representation depicted with code on the left and presence condition for each line to the right. Recall that the actual internal representation is a modular tree structure (cf. Fig. 6). Also notice that ECCO does not duplicate artifacts (as was done by the Marlin developers in Lines 9 and 11 of Listing 1) and instead assigns a presence condition with a logical *OR* to the artifact in such cases.

2. The external representation (a concrete variant) obtained by externalization with an EE (configuration) that is equivalent to the formula $\_USES\_RAM \wedge \_CACHE\_NAMES \wedge \neg \_USES\_STACK \wedge HAS\_FOLDER\_SORTING \wedge \_DYNAMIC\_RAM$, including a warning that the new feature `_DYNAMIC_RAM` is unknown.
3. The external representation after having addressed the warnings (i.e., in this case, after having added the new feature in the context of the concrete variant).
4. The resulting internal representation after internalization with the same IE that was used for externalization. Note that the new feature is added negatively (in red) to presence conditions of lines that were removed and positively (in green) to lines that were added. For added lines, all possible feature interactions are added as extra clauses (even if redundant at this time, which is why higher order feature interactions are grayed out if lower order ones are present) as they may be needed in the future for refining the presence conditions and also for providing warnings.
5. The following steps are optional and only necessary when additional variants are needed. Assume that another variant is externalized by providing the EE $\neg \_USES\_RAM \wedge \neg \_CACHE\_NAMES \wedge \neg \_USES\_STACK \wedge \neg HAS\_FOLDER\_SORTING \wedge \_DYNAMIC\_RAM$. The external representation of the desired variant may be incomplete as explained by the provided warnings. One warning explains that feature `_DYNAMIC_RAM` has never existed without feature `_USED_RAM` and corresponding code (if any) does not exist in the repository yet (in this case no such code is required and the warning can be ignored). Another warning says that there may be surplus code that belongs to the interaction of the features `_DYNAMIC_RAM` and `_USED_RAM` that may need to be removed (in this case the last two lines of code).
6. The external representation after having addressed the warnings (i.e., in this case, after the developer removed the surplus code as suggested by the warning).
7. The resulting internal representation after internalization with the same IE as was used for externalization. Note that since no new features were added the existing presence conditions were refined and made more precise by removing unnecessary clauses, i.e., clauses that were redundant are no longer redundant.

6.2.2. SuperMod

SuperMod assumes, like ECCO, that the developer works on a single variant, represented by a full configuration, without seeing variability annotations. A key difference, however, is that the IE specified during commit is partial and shall refer only to those features for which the change is relevant. Also, SuperMod does not rely on a preprocessor-based representation for variability annotations like VTS, but utilizes a custom storage format (optimized for Ecore models, but also capable of text).

Figure 12 depicts how the running example can be replayed using SuperMod. The figure reveals that the user may perform the necessary changes



```
uint8_t sort_order[_LIMIT];                      true
char sortshort[_LIMIT][FILENAME_LENGTH];         _USES_RAM && _CACHE_NAMES
char sortnames[_LIMIT][FILENAME_LENGTH];         _USES_RAM && _CACHE_NAMES
char sortnames[_LIMIT][FILENAME_LENGTH];         _USES_RAM && !_CACHE_NAMES && !_USES_STACK
uint8_t isDir[(_LIMIT+7)>>3];                    _USES_RAM && _SORTING && _CACHE_NAMES
uint8_t isDir[(_LIMIT+7)>>3];                    _USES_RAM && _SORTING && !_CACHE_NAMES && !_USES_STACK
```

```
uint8_t sort_order[_LIMIT];
char sortshort[_LIMIT][FILENAME_LENGTH];
char sortnames[_LIMIT][FILENAME_LENGTH];
uint8_t isDir[(_LIMIT+7)>>3];
```

```
uint8_t *sort_order;
char sortshort[_LIMIT][FILENAME_LENGTH];
char sortnames[_LIMIT][FILENAME_LENGTH];
uint8_t isDir[(_LIMIT+7)>>3];
```

```
uint8_t *sort_order;
char **sortshort, **sortnames;
uint8_t isDir[(_LIMIT+7)>>3];
```

```
uint8_t *sort_order;
char **sortshort, **sortnames;
uint8_t *isDir;
```

```
uint8_t sort_order[_LIMIT];                      !_DYNAMIC_RAM
uint8_t *sort_order;                             _DYNAMIC_RAM
char sortshort[_LIMIT][FILENAME_LENGTH];         _USES_RAM && _CACHE_NAMES && !_DYNAMIC_RAM
char sortnames[_LIMIT][FILENAME_LENGTH];         _USES_RAM && _CACHE_NAMES && !_DYNAMIC_RAM
char **sortshort, **sortnames;                   _USES_RAM && _CACHE_NAMES && _DYNAMIC_RAM
char sortnames[_LIMIT][FILENAME_LENGTH];         _USES_RAM && !_CACHE_NAMES && !_USES_STACK
uint8_t isDir[(_LIMIT+7)>>3];                    _USES_RAM && _SORTING && _CACHE_NAMES && !_DYNAMIC_RAM
uint8_t isDir[(_LIMIT+7)>>3];                    _USES_RAM && _SORTING && !_CACHE_NAMES && !_USES_STACK
uint8_t *isDir;                                  _USES_RAM && _SORTING && DYNAMIC_RAM
```

Figure 12: The running example of Sec. 2.4 performed with SuperMod. The internal representations (i.e., repository contents) are shown in a simplified form (textually with variability annotations on the right) with a grey background color, where the presence condition of every line of text is presented as conjunction in the right box; for reasons of compactness, the prefixes SDSORT_ and HAS_FOLDER_ have been replaced by a simple _. Different states of the external representation are shown in white. User actions are represented with a bold border. Choices and ambitions are depicted as trees representing complete and partial feature configurations, respectively; cyan denotes selection, orange deselection, and yellow is for an undefined state (ambitions only).

in a single view (i.e., external representation), but also that three subsequent commits, which are connected to three different feature ambitions, are necessary. The reason is that the modifications are supposed to be connected to different change scopes. While all changes are connected to the new feature SDSORT_DYNAMIC_RAM, the second change is relevant only when this feature interacts with both SDSORT_CACHE_NAMES and SDSORT_USES_RAM, and the third change assumes HAS_FOLDER_SORTING and SDSORT_USES_RAM in addition. The figure also illustrates to which extent the user is supported in specifying internalization/EEs by means of a feature model.

1. The initial internal representation depicted with code on the left and presence condition for each line on the right. Note, that the actual internal representation used by SuperMod is an Ecore model where each line is represented by model elements.
2. The external representation created by using the EE (full configuration)

HAS_FOLDER_SORTING $\wedge$ _CACHE_NAMES $\wedge$ _USES_RAM $\wedge$ $\neg$_USES_STACK.

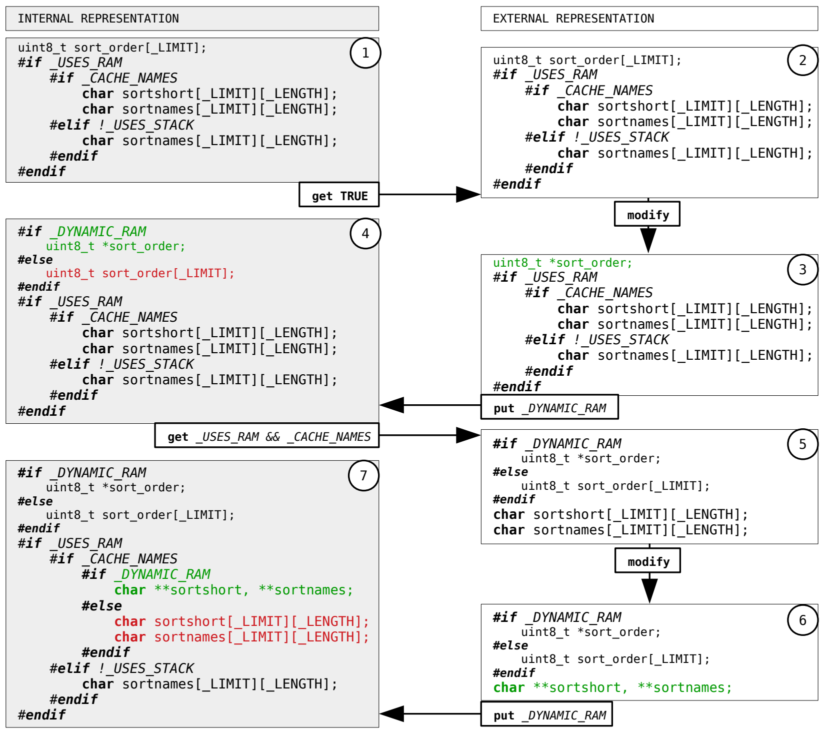
3. The external representation after being modified (first line changed) for the first internalization with expression _DYNAMIC_RAM. In addition, it is specified, that the newly added feature _DYNAMIC_RAM is to be placed under the existing feature _USES_RAM in the feature model.
4. The external representation after having been modified further (without another externalization operation) by replacing lines two and three. The second internalization operation is performed with expression _CACHE_NAMES $\wedge$ _USES_RAM $\wedge$ _DYNAMIC_RAM.
5. The external representation after its final modification (last line changed) for the final internalization with expression HAS_FOLDER_SORTING$\wedge$_USES_RAM$\wedge$ $\neg$_DYNAMIC_RAM
6. The final internal representation after the three performed internalization operations. Lines that were removed had their presence condition extended via conjunction with negated IE (red). Lines that were added received the positive IE and presence condition (green).

6.2.3. VTS

VTS allows both EEs and IEs to be partial. This makes it possible to leave variability in the external representation. Therefore, internal as well as external representation consist of text files containing C preprocessor annotations. This makes VTS the system that puts the user closest to the actual variation points as it allows to modify them directly in the external representation if desired. Presence conditions for added lines are computed as the conjunction of the EE and the IE (EE $\wedge$ IE), and for removed lines as the conjunction of the EE and the negation of the IE (EE $\wedge$ $\neg$IE). This can make it quite difficult to perform the desired changes using the internalization and externalization operations as both expressions need to be chosen correctly, which can sometimes make it the better choice to edit variation points directly. This is in contrast to SuperMod where only the (also partial) IE is used to compute presence conditions for changes.

Figure 13 shows the running example of Sec. 2.4 performed with VTS. It was simplified slightly by removing the last code block that implements the feature HAS_FOLDER_SORTING as it would otherwise have required several more editing cycles. Also, it was performed by applying the concepts and editing model of VTS theoretically, without using the actual implementation. While there is an implementation of VTS publicly available, in its current rudimentary state it was unfortunately not possible to apply it to the posed scenario.

1. The initial state of the internal representation. VTS uses plain text files annotated with C preprocessor directives.
2. The external representation after obtained with EE TRUE. This leaves all variability in the external representation which is necessary in this case as we want the following change to only get the new feature _DYNAMIC_RAM as presence condition and therefore the EE must be empty.
3. The external representation after having replaced the first line of code that shall only belong to feature _DYNAMIC_RAM.



```
uint8_t sort_order[_LIMIT];
#if _USES_RAM
    #if _CACHE_NAMES
        char sortshort[_LIMIT][_LENGTH];
        char sortnames[_LIMIT][_LENGTH];
    #elif !_USES_STACK
        char sortnames[_LIMIT][_LENGTH];
    #endif
#endif
```

```
uint8_t sort_order[_LIMIT];
#if _USES_RAM
    #if _CACHE_NAMES
        char sortshort[_LIMIT][_LENGTH];
        char sortnames[_LIMIT][_LENGTH];
    #elif !_USES_STACK
        char sortnames[_LIMIT][_LENGTH];
    #endif
#endif
```

```
uint8_t *sort_order;
#if _USES_RAM
    #if _CACHE_NAMES
        char sortshort[_LIMIT][_LENGTH];
        char sortnames[_LIMIT][_LENGTH];
    #elif !_USES_STACK
        char sortnames[_LIMIT][_LENGTH];
    #endif
#endif
```

```
#if _DYNAMIC_RAM
    uint8_t *sort_order;
#else
    uint8_t sort_order[_LIMIT];
#endif
#if _USES_RAM
    #if _CACHE_NAMES
        char sortshort[_LIMIT][_LENGTH];
        char sortnames[_LIMIT][_LENGTH];
    #elif !_USES_STACK
        char sortnames[_LIMIT][_LENGTH];
    #endif
#endif
```

```
#if _DYNAMIC_RAM
    uint8_t *sort_order;
#else
    uint8_t sort_order[_LIMIT];
#endif
char sortshort[_LIMIT][_LENGTH];
char sortnames[_LIMIT][_LENGTH];
```

```
#if _DYNAMIC_RAM
    uint8_t *sort_order;
#else
    uint8_t sort_order[_LIMIT];
#endif
char **sortshort, **sortnames;
```

```
#if _DYNAMIC_RAM
    uint8_t *sort_order;
#else
    uint8_t sort_order[_LIMIT];
#endif
#if _USES_RAM
    #if _CACHE_NAMES
        #if _DYNAMIC_RAM
            char **sortshort, **sortnames;
        #else
            char sortshort[_LIMIT][_LENGTH];
            char sortnames[_LIMIT][_LENGTH];
        #endif
    #elif !_USES_STACK
        char sortnames[_LIMIT][_LENGTH];
    #endif
#endif
```

Figure 13: The (slightly simplified) running example of Sec. 2.4 performed by applying the VTS editing model. The internal representations (i.e., repository contents) are depicted with grey background. The external representations (i.e., workspace contents) have a white background. User actions are placed along arrows and represented with a bold border. Expressions are given as Boolean formulas over features. For brevity, the prefix SDSORT_ is replaced by a simple _.

4. The updated internal representation with IE _DYNAMIC_RAM. The new line (green) gets the condition $EE \wedge IE$ = TRUE $\wedge$ _DYNAMIC_RAM = _DYNAMIC_RAM. The removed line (red) gets the condition $EE \wedge \neg IE$ = TRUE $\wedge \neg$_DYNAMIC_RAM = $\neg$_DYNAMIC_RAM (which is achieved by putting it in the *#else* part).
5. The external representation with some variability removed (features _USES-_RAM and _CACHE_NAMES have been decided) and some remaining (feature _DYNAMIC_RAM remains variable). Again, we have to keep in mind that this EE will be used in the presence condition for following changes.
6. The external representation with two lines replaced by a new one.
7. The updated internal representation with IE _DYNAMIC_RAM. This gives the new line (green) the presence condition _USES_RAM $\wedge$ _CACHE_NAMES $\wedge$ _DYNAMIC_RAM and the removed lines (red) the presence condition _USES_RAM$\wedge$ _CACHE_NAMES $\wedge \neg$_DYNAMIC_RAM (again achieved by putting it in the *#else* part).

6.2.4. Analysis and Discussion

We now analyze and discuss the illustrated modification scenario in detail.

The scenario illustration for VTS was shortened for space reasons (the last view lines of code were ignored), which is why it appears to be easier to use. However, in fact, performing the full scenario (as was done for ECCO and SuperMod) would have required several more *get* (externalization) and *put* (internalization) cycles. This is due to the fact that VTS only allows internalization with an expression that is at least as strong as the externalization. This forces the user to change the projection (via the get operation) to another variant many times in order to achieve the desired conditions for the performed changes. VTS supports partial EEs and therefore variable external representations that expose preprocessor annotations directly to the user. This could be leveraged to reduce the number of necessary operations by instead directly editing the variation points (i.e., annotations). However, in the extreme case, VTS would then provide no benefit over simply working with preprocessor annotated code directly. In summary, VTS is more tedious to use because it only allows internalization that is at least as strong as the externalization. However, this has the advantage of avoiding inconsistencies and side effects.

In contrast, SuperMod only supports internalization that is at most as strong as the externalization. This drastically reduces the number of times the user needs to change variants (via the checkout operation) to achieve the desired conditions for the performed changes. Additionally, specifying the IE is easier than in VTS, as it is applied directly as a presence condition to the performed changes and not concatenated with the EE (as done by VTS to guarantee that the resulting presence condition is at least as strong as the one used for the externalization).

Both VTS and SuperMod require the user to specify an IE that ends up more or less directly as the presence condition of changes in the internal representation normally not shown to the user. In other words, while users do not see or directly modify the internal representation, they still need to provide an expression that is very close to it.

ECCO, on the other hand, is fully variant-centric and aims at freeing users from worrying about the internal representation and the presence conditions of changes. ECCO thus hides the internal representation (as done by other VarCS) but also the presence conditions from the user. The specified expressions are simple configurations (i.e., lists of features) and never presence conditions, and in almost all cases the IE will be the same as the EE. The user only needs to specify the desired variant (externalization), potentially even including new features that have not existed until that point. ECCO then issues a list of warnings that tell the user what work still might need to be done on the variant (if any) to complete its implementation (e.g., implementing yet unknown features). While technically not required, in almost all cases the user can use the same expression for internalization as was used for externalization after completing the implementation of the variant. ECCO then analyzes the observed variants and incrementally refines the internal representation automatically. This means that the internal presence conditions might initially contain uncertainty and be less specific than they could be. However, in practice, it might not be necessary to have such specific presence conditions attached. The assumption is that only a small fraction of all supported variants will ever be used in practice and that the presence conditions only need to be specific enough to support the actually needed variants. If ever a new variant is needed, the presence conditions will be refined accordingly as needed.

7. Challenges and Research Perspectives

A key motivation of our work is to raise awareness of the potential of VarCS and what would be needed to be able to fully leverage it. Based on the results of our study we discuss open problems, identify remaining challenges and suggest research perspectives related to VarCS.

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 as pointed out in the *Cognitive Dimensions of Notations* framework [94] where this is referred to as *hard mental operations*. For instance, creating EEs 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 may help. On the other hand, for developers that are used to the clone-and-own practice it might be easier to think in variants (i.e., full configurations). Likewise, generating the EE 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.

Useful abstractions seem essential to facilitate the use of VarCS. 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. There even exist feature modeling approaches that incorporate a concept of evolution such as Hyper Feature Models [95] or Temporal Feature Models [96] that could be used for representing revisions in the context of VarCS. Developers can be supported by creating EEs based on feature models, as the SuperMod system shows. Perhaps even more intuitive and expressive, DSL-based abstractions can be crafted. An example is the use of intentions [20] for integrating variants, which can replace IEs (internalization expressions) and which form a small DSL.

In summary, besides the technical challenges of creating more intentional and feature-based front-ends, user studies are needed to better understand how developers can cope with the cognitive complexity.

Higher number of operations needed to perform a task. No matter which VarCS is used, the number of operations needed to achieve the same goal that can be achieved with a single manual change is always higher (see running example performed manually in Sec. 2.4). This is well illustrated in Sec. 6.2. Every individual operation in itself might be fairly simple, but decomposing the overall goal into the right sequence of such operations is challenging (see above) and requires additional effort. However, one can argue that with current version control systems (such as Git) many tasks that *should* also be performed in multiple steps simply are not, due to lack of discipline and also lack of immediate negative consequences in many cases. For example, consider a developer working with Git on a certain feature in the respective feature branch, noticing a typo in another feature's code. The developer *should* stash (or commit) the current changes, switch branch, make the change, commit on that branch, and switch back to the original feature's branch. What actually happens in most cases is, that the developer simply fixes the typo immediately and the change gets mixed up with the wrong feature's changes. With traditional version control systems the negative effect of this undisciplined behavior is usually limited as the feature branch will be merged back into its parent branch at some point anyway.

Unclear criteria for using VarCS. It may not be obvious in what situations a VarCS would be the right tool to use and actually provide tangible benefits. Studies on what characteristics a variable system should have to benefit from the use of a VarCS would give necessary insights. Such characteristics could be the types of artifacts that need to be variable (textual, graphical, etc.), whether the implementation consists of only a single or multiple types of artifacts, or whether the complexity of artifacts is dominated by variability or the actual payload. If variations are fairly comprehensible (i.e., fairly simple expressions), but the artifacts themselves are very complex (e.g., because they solve a complex problem), then VarCS could be useful as they make it easier to focus on the complex problem by removing the unneeded variability from the (already complex) artifacts. Also, VarCS provide a benefit if variations (and, consequently, EE and IE) are simple, but the variation points are scattered [97] all over the place. In this case, the expressions the user needs to provide are fairly simple and the placement of all the conditions is done automatically. For instance, when considering the context of preprocessor annotations this essentially comes down to the relation of the number of lines of code responsible for variability (i.e., annotations) compared to the number of lines of code responsible for the actual implementation. A possible insight *could be* that VarCS are useful when more lines of code contain annotations than actual implementation. All this needs to be investigated in more detail, for example by performing user studies.

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 shows that while the investigated systems offer 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 [98]. Such techniques allow the development of tools visualizing the variants affected by a change to understand and assess its impact. In the context of VarCS this could be used to determine the effect of changes that are applied with a weaker IE than EE on the variants that are not visible in the current view. If the change affects these not visible variants but the change has no additional impact beyond the one it has in the visible variants then there is no risk in using a weaker IE.

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 import and export its repository contents from and to a universal exchange format. However, it is not trivial to agree on such a format. For instance, a format based on textual preprocessor annotations would work well for systems such as VTS and Leviathan, which already use such formats as their internal representation. On the other hand, this format can only be used on textual artifacts but not on models, diagrams or other non-textual artifacts which is an important limitation. A more advanced approach would be to use transformations to and from artifact-specific representations, possibly requiring 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 [99]. For every type of artifact a VarCS supports it would be required to have transformations from and to at least one artifact-specific format that can be used without the VarCS.

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 by first creating all variants from one VarCS (assuming they are not too many) and then importing them into another, thereby also reducing the locked-in syndrome. 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*,[10] 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 as also pointed out by Hinterreiter et al. [85, 86]. 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.

8. Threats to Validity

A threat to the external validity of our study is that we might have missed important VarCS. Our focus was to study selected existing VarCS in depth, which would not be possible with a pure literature review. Some of our subjects are older than 30 years and often the publications are no longer available in digital libraries. This would make a keyword-based search strategy highly unreliable.

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

However, our research method included several measures to ensure completeness: as explained in Section 3 (Step 1) our literature search already builds on existing survey papers. We used snowballing to identify further relevant papers and systems. We performed a retrospective search as part of the thematic synthesis. We presented the list of subject systems to the participants of Dagstuhl-Seminar 19191 (Software Evolution in Time and Space: Unifying Version and Variability Management) [100]. Participants included senior researchers (some involved in earlier survey papers) and junior researchers actively pursing research in this field. Overall, we are confident that our study covers the most relevant subject systems for the purpose of this article.

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 [73]. 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 concepts that are needed to realize new and better systems. However, since we have been involved in the development of some of the systems we are confident that we identified the relevant concepts and characteristics. 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. Conclusion

We 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 even though they were developed in different research communities, for different purposes, and in a time span covering several decades. The results also reveal particular strengths of individual VarCS.

Based on these findings our work aims to raise awareness about VarCS and remaining research challenges such as the cognitive complexity of handling logical variants for different revisions of features, the complex workflows needed to consistently write back changes made to variants to the shared artifact repository, the risk of becoming locked-in to a particular style of artifact repository, the lack of support for collaborative and distributed development, and the maturity of the current evaluations. Based on these challenges our article suggests research perspectives related to VarCS.

Acknowledgments

This work has been supported by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, KEBA AG Austria, the Swedish Research Council, Vinnova Sweden, and the Wallenberg Academy.

We thank the participants of Dagstuhl-Seminar 19191 (Software Evolution in Time and Space: Unifying Version and Variability Management) [100] for all the interesting and fruitful discussions on this topic.

References

[1] T. Berger, J.-P. Steghöfer, T. Ziadi, J. Robin, J. Martinez, The state of adoption and the challenges of systematic variability management in industry, Empirical Software Engineering 25 (2020) 1755–1797.

[2] T. Berger, S. She, R. Lotufo, A. Wasowski, K. Czarnecki, A study of variability models and languages in the systems software domain, IEEE Transactions on Software Engineering 39 (2013) 1611–1640.

[3] T. Berger, S. She, R. Lotufo, K. Czarnecki, A. Wasowski, Feature-to-code mapping in two large product lines, in: 14th International Software Product Line Conference (SPLC), 2010.

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

[5] J. Melo, C. Brabrand, A. Wasowski, How does the degree of variability affect bug finding?, in: Proceedings of the 38th International Conference on Software Engineering (ICSE), ACM, 2016, pp. 679–690.

[6] J. Favre, Preprocessors from an abstract point of view, in: 3rd Working Conference on Reverse Engineering (WCRE), IEEE Computer Society, 1996, pp. 287–296.

[7] H. Spencer, C. Geoff, #ifdef Considered Harmful, or Portability Experience With C News, in: USENIX Summer Technical Conference, 1992, pp. 185–198.

[8] L. Linsbauer, T. Berger, P. Grünbacher, A classification of variation control systems, in: Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE), ACM, 2017, pp. 49–62.

[9] R. Conradi, B. Westfechtel, Version models for software configuration management, ACM Computing Surveys 30 (1998) 232–282.

[10] A. Mahler, Configuration management, John Wiley & Sons, Inc., New York, NY, USA, 1995, pp. 73–97.

[11] S. A. MacKay, 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, 1995.

[12] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, K. Czarnecki, 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, 2015, pp. 16–25.

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

[14] B. Westfechtel, B. P. Munch, R. Conradi, A layered architecture for uniform version management, IEEE Transactions on Software Engineering 27 (2001) 1111–1133.

[15] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, K. Czarnecki, An exploratory study of cloning in industrial software product lines, in: Proceedings 17th European Conference on Software Maintenance and Reengineering, 2013, pp. 25–34.

[16] J. Businge, M. Openja, S. Nadi, E. Bainomugisha, T. Berger, Clone-based variability management in the Android ecosystem, in: IEEE International Conference on Software Maintenance and Evolution, (ICSME), IEEE Computer Society, 2018, pp. 625–634.

[17] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stanciulescu, A. Wasowski, I. Schaefer, Flexible product line engineering with a virtual platform, in: P. Jalote, L. C. Briand, A. van der Hoek (Eds.), Proceedings 36th International Conference on Software Engineering, (ICSE), ACM, 2014, pp. 532–535.

[18] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, A. Egyed, Reengineering legacy applications into software product lines: a systematic mapping, Empirical Software Engineering 22 (2017) 2972–3016.

[19] J. Krueger, T. Berger, An empirical analysis of the costs of clone- and platform-oriented software reuse, in: 28th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), 2020.

[20] M. Lillack, S. Stanciulescu, W. Hedman, T. Berger, A. Wasowski, Intention-based integration of software variants, in: Proceedings of the 41st International Conference on Software Engineering (ICSE), IEEE/ACM, 2019, pp. 831–842.

[21] D. L. Parnas, On the design and development of program families, IEEE Transactions on Software Engineering 2 (1976) 1–9.

[22] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, Boston, MA, 2001.

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

[24] K. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report SEI-90-TR-21, CMU, 1990.

[25] N. Singh, C. Gibbs, 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.

[26] C. Kästner, S. Apel, M. Kuhlemann, Granularity in software product lines, in: Proceedings of the 30th International Conference on Software Engineering, ICSE '08, ACM, New York, NY, USA, 2008, pp. 311–320.

[27] B. Behringer, J. Palz, T. Berger, PEoPL: projectional editing of product lines, in: Proceedings of the 39th International Conference on Software Engineering (ICSE), IEEE/ACM, 2017, pp. 563–574.

[28] F. Angerer, H. Prähofer, D. Lettner, A. Grimmer, P. Grünbacher, 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, 2014, pp. 52–61.

[29] B. Kullbach, V. Riediger, Folding: An approach to enable program understanding of preprocessed languages, in: E. Burd, P. Aiken, R. Koschke (Eds.), Proceedings of the Eighth Working Conference on Reverse Engineering, (WCRE), IEEE Computer Society, 2001, pp. 3–12.

[30] J. Krueger, T. Berger, Activities and costs of re-engineering cloned variants into an integrated platform, in: 14th International Working Conference on Variability Modelling of Software-intensive Systems (VaMoS), 2020.

[31] M. T. Rahman, L. Querel, P. C. Rigby, B. Adams, Feature toggles: practitioner practices and a case study, in: Proceedings 13th International Conference on Mining Software Repositories (MSR), ACM, 2016, pp. 201–211.

[32] J. Meinicke, C.-P. Wong, B. Vasilescu, C. Kästner, Exploring differences and commonalities between feature flags and configuration options, in: Proc. Int'l Conf. Software Engineering–Software Engineering in Practice (ICSE-SEIP). ACM, 2020.

[33] D. S. Batory, Feature-oriented programming and the AHEAD tool suite, in: A. Finkelstein, J. Estublier, D. S. Rosenblum (Eds.), Proceedings 26th International Conference on Software Engineering (ICSE), IEEE Computer Society, 2004, pp. 702–703.

[34] D. S. Batory, J. N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, IEEE Trans. Software Eng. 30 (2004) 355–371.

[35] S. Apel, C. Kästner, C. Lengauer, FEATUREHOUSE: language-independent, automated software composition, in: Proceedings 31st International Conference on Software Engineering, ICSE, May 16-24, 2009, Vancouver, Canada, IEEE, 2009, pp. 221–231.

[36] S. Apel, C. Kästner, C. Lengauer, Language-independent and automated software composition: The FeatureHouse experience, IEEE Transactions on Software Engineering 39 (2013) 63–79.

[37] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings 11th European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science 1241, Springer, 1997, pp. 220–242.

[38] I. Schaefer, L. Bettini, V. Bono, F. Damiani, N. Tanzarella, Delta-oriented programming of software product lines, in: J. Bosch, J. Lee (Eds.), Proceedings 14th International Conference on Software Product Lines: Going Beyond (SPLC), Lecture Notes in Computer Science 6287, Springer, 2010, pp. 77–91.

[39] R. Hirschfeld, P. Costanza, O. Nierstrasz, Context-oriented programming, Journal of Object Technology 7 (2008) 125–151.

[40] B. Kitchenham, Guidelines for performing systematic literature reviews in software engineering, Technical Report Version 2.3, EBSE Technical Report, 2007.

[41] W. Tichy, Software Configuration Management Overview, Technical Report, 1988.

[42] C. W. Fraser, E. W. Myers, An editor for revision control, ACM Transactions on Programming Languages and Systems 9 (1987) 277–295.

[43] D. Budgen, M. Turner, P. Brereton, B. Kitchenham, Using mapping studies in software engineering, in: Proc. of PPIG, volume 8, Lancaster University, 2008, pp. 195–204.

[44] D. S. Cruzes, T. Dyba, Recommended steps for thematic synthesis in software engineering, in: Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE Computer Society, Washington, DC, USA, 2011, pp. 275–284.

[45] S. Stanciulescu, T. Berger, E. Walkingshaw, A. Wasowski, Concepts, operations, and feasibility of a projection-based variation control system, in: Proceedings IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE Computer Society, 2016, pp. 323–333.

[46] B. Ganter, G. Stumme, R. Wille (Eds.), Formal Concept Analysis, Foundations and Applications, Lecture Notes in Computer Science 3626, Springer, 2005.

[47] B. Ganter, R. Wille, Formal Concept Analysis – Mathematical Foundations, Springer, 1999.

[48] B. Ganter, R. Wille, D. Borchmann, J. Prochaska, Implications and dependencies between attributes, in: Proceedings of the 14th International Conference on Formal Concept Analysis (ICFCA), 2017, pp. 23–35.

[49] V. J. Kruskal, Managing multi-version programs with an editor, IBM Journal of Research and Development 28 (1984) 74–81.

[50] 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.

[51] M. J. Rochkind, The source code control system, IEEE Transactions on Software Engineering 1 (1975) 364–370.

[52] B. P. Munch, J.-O. Larsen, B. Gulla, R. Conradi, E.-A. Karlsson, Uniform versioning: The change-oriented model, in: Proceedings Fourth International Workshop on Software Configuration Management (SCM), 1993, p. 188–196.

[53] B. P. Munch, Versioning in a Software Engineering Database – The Change Oriented Way, Ph.D. thesis, The Norwegian Institute of Technology, 1993.

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

[55] A. Lie, Versioning in Software Engineering Databases, Ph.D. thesis, The Norwegian Institute of Technology, 1990.

[56] A. Lie, R. Conradi, T. Didriksen, E. Karlsson, Change oriented versioning in a software engineering database, in: Proceedings 2nd International Workshop on Software Configuration Management (SCM), 1989, pp. 56–65.

[57] B. P. Munch, R. Conradi, J.-O. Larsen, M. N. Nguyen, P. H. Westby, Integrated product and process management in EPOS, Integrated Computer-Aided Engineering 3 (1996) 5–19.

[58] A. I. Wang, J.-O. Larsen, R. Conradi, B. P. Munch, Improving cooperation support in the EPOS CM system, in: Proceedings 6th European Workshop on Software Process Technology (EWSPT), Lecture Notes in Computer Science 1487, Springer, 1998, pp. 75–91.

[59] W. Hofer, C. Elsner, F. Blendinger, W. Schröder-Preikschat, D. Lohmann, Toolchain-independent variant management with the Leviathan filesystem, in: Proceedings of the 2nd International Workshop on Feature-Oriented Software Development (FOSD), ACM, 2010, pp. 18–24.

[60] E. Walkingshaw, K. Ostermann, Projectional editing of variational software, in: Generative Programming: Concepts and Experiences (GPCE), 2014, pp. 29–38.

[61] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, J. Siegmund, Efficiency of projectional editing: A controlled experiment, in: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2016, pp. 763–774.

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

[63] L. Linsbauer, E. R. Lopez-Herrejon, A. Egyed, Recovering traceability between features and code in product variants, in: Proceedings of the 17th International Software Product Line Conference (SPLC), ACM, 2013, pp. 131–140.

[64] L. Linsbauer, R. E. Lopez-Herrejon, A. Egyed, Variability extraction and modeling for product variants, Software and Systems Modeling 16 (2017) 1179–1199.

[65] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, A. Egyed, Enhancing clone-and-own with systematic reuse for developing software variants, in: Proceedings 30th IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014, pp. 391–400.

[66] L. Linsbauer, S. Fischer, R. E. Lopez-Herrejon, A. Egyed, 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, 2015, pp. 57–60.

[67] L. Linsbauer, A. Egyed, R. E. Lopez-Herrejon, A variability aware configuration management and revision control platform, in: Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016, pp. 803–806.

[68] L. Linsbauer, Managing and Engineering Variability Intensive Systems, Ph.D. thesis, Johannes Kepler University Linz, 2016.

[69] F. Schwägerl, B. Westfechtel, Supermod: tool support for collaborative filtered model-driven software product line engineering, in: Proceedings 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016, pp. 822–827.

[70] F. Schwägerl, B. Westfechtel, 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, 2016, pp. 83–94.

[71] B. Westfechtel, R. Conradi, Multi-variant modeling - concepts, issues and challenges, in: Proceedings of the 1st International Workshop on Model-Driven Product Line Engineering (MDPLE 2009), CTIT Proceedings, 2009, pp. 57–67.

[72] N. Sarnak, R. Bernstein, V. Kruskal, Creation and maintenance of multiple versions, in: Workshop on Software Version and Configuration Control, 1988.

[73] I. P. Goldstein, D. G. Bobrow, A layered approach to software design, Technical Report CSL-80-5, Xerox. Palo Alto Research Center, 1980.

[74] W. M. Gentleman, A. MacKay, D. A. Stewart, Commercial realtime software needs different configuation management, in: Proceedings 2nd International Workshop on Software Configuration Management (SCM), ACM, 1989, pp. 152–161.

[75] R. D. Cronk, Tributaries and deltas, BYTE 17 (1992) 177–186.

[76] Software Maintenance & Development Systems, Inc, Aide de camp product overview, Concord, Massachusetts, 1990.

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

[78] D. L. Atkins, 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, 1998, pp. 146–157.

[79] A. A. Pal, M. B. Thompson, 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, 1989.

[80] J. O. Coplien, D. L. DeBruler, M. B. Thompson, The Delta system: A nontraditional approach to software version management, in: AT&T Technical Papers, International Switching Symposium, 1987.

[81] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, A. Wasowski, Three cases of feature-based variability modeling in industry, in: Proceedings 17th International Conference Model-Driven Engineering Languages and Systems (MODELS), Springer International Publishing, 2014, pp. 302–319.

[82] D. Nesic, J. Krueger, S. Stanciulescu, T. Berger, Principles of feature modeling, in: 27th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), 2019.

[83] K. Schmid, R. Rabiser, P. Grünbacher, A comparison of decision modeling approaches in product lines, in: Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS), ACM, 2011, pp. 119–126.

[84] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, A. Wasowski, 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), ACM, 2012, pp. 173–182.

[85] D. Hinterreiter, H. Prähofer, L. Linsbauer, P. Grünbacher, F. Reisinger, A. Egyed, Feature-oriented evolution of automation software systems in industrial software ecosystems, in: 23rd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2018, pp. 107–114.

[86] D. Hinterreiter, K. Feichtinger, L. Linsbauer, H. Prähofer, P. Grünbacher, Supporting feature model evolution by lifting code-level dependencies: A research preview, in: E. Knauss, M. Goedicke (Eds.), Proceedings 25th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ), Lecture Notes in Computer Science 11412, Springer, 2019, pp. 169–175.

[87] C. Seidl, I. Schaefer, U. Aßmann, Capturing variability in space and time with hyper feature models, in: 8th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS), ACM, 2014, pp. 6:1–6:8.

[88] S. Apel, C. Kästner, An Overview of Feature-Oriented Software Development, Journal of Object Technology 8 (2009) 49–84.

[89] F. Schwägerl, B. Westfechtel, Maintaining workspace consistency in filtered editing of dynamically evolving model-driven software product lines, in: Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2017, pp. 15–28.

[90] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. S. Batory, G. Saake, On the impact of the optional feature problem: analysis and case studies, in: Proceedings 13th International Conference on Software Product Lines (SPLC), ACM, 2009, pp. 181–190.

[91] L. Linsbauer, F. Angerer, P. Grünbacher, D. Lettner, H. Prähofer, R. Lopez-Herrejon, A. Egyed, Recovering feature-to-code mappings in mixed-variability software systems, in: Proceedings 30th International Conference on Software Maintenance and Evolution (ICSME), 2014, pp. 426–430.

[92] D. Hinterreiter, L. Linsbauer, K. Feichtinger, H. Prähofer, P. Grünbacher, Supporting feature-oriented evolution in industrial automation product lines, Concurrent Engineering: Research and Applications (2020).

[93] F. Schwägerl, B. Westfechtel, Integrated revision and variation control for evolving model-driven software product lines, Software and Systems Modeling 18 (2019) 3373–3420.

[94] A. Blackwell, T. Green, Notational Systems – the Cognitive Dimensions of Notations framework, in: J. Carroll (Ed.), HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science, Morgan Kaufmann, San Francisco, 2003, pp. 103–134.

[95] C. Seidl, I. Schaefer, U. Aßmann, Capturing Variability in Space and Time with Hyper Feature Models, in: Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '14, ACM, New York, NY, USA, 2013, pp. 6:1–6:8.

[96] D. Hinterreiter, M. Nieke, L. Linsbauer, C. Seidl, H. Prähofer, P. Grünbacher, Harmonized temporal feature modeling to uniformly perform, track, analyze, and replay software product line evolution, in: Proceedings of the 18th International Conference on Generative Programming: Concepts & Experiences (GPCE), Athens, Greece, 2019, pp. 115–128.

[97] L. Passos, R. Queiroz, M. Mukelabai, T. Berger, S. Apel, K. Czarnecki, J. Padilla, A study of feature scattering in the linux kernel, IEEE Transactions on Software Engineering (2018). Preprint.

[98] F. Angerer, A. Grimmer, H. Prähofer, P. Grünbacher, Configuration-aware change impact analysis, in: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE '15, 2015, pp. 385–395.

[99] A. von Rhein, T. Thüm, I. Schaefer, J. Liebig, S. Apel, Variability encoding: From compile-time to load-time variability, Journal of Logical and Algebraic Methods in Programming 85 (2016) 125–145. Formal Methods for Software Product Line Engineering.

[100] T. Berger, M. Chechik, T. Kehrer, M. Wimmer, Software evolution in time and space: Unifying version and variability management (dagstuhl seminar 19191), in: Dagstuhl Reports, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2019.