Unified Operations for Variability in Space and Time

Sofia Ananieva,[1] Sandra Greiner,[2,3] Jacob Krüger,[4,5] Lukas Linsbauer,[6]
Sten Grüner,[7] Timo Kehrer,[8,9] Thomas Kühn,[10] Christoph Seidl,[3] Ralf Reussner[10]
[1]FZI Research Center for Information Technology, [2]University of Bayreuth,
[3]IT-University of Copenhagen, [4]Otto-von-Guericke University Magdeburg, [5]Ruhr-University Bochum,
[6]Technische Universität Braunschweig, [7]ABB Corporate Research Center Germany, [8]Humboldt University of Berlin,
[9]University of Bern, [10]Karlsruhe Institute of Technology

ABSTRACT

Software and systems engineering is challenged by variability in space (concurrent variations at a single point in time) and time (sequential variations due to evolution). Managing both dimensions of variability independently is cumbersome and error-prone. A common foundation for operations on these dimensions is still missing, hampering the comparison and integration of existing techniques coping with variability in space and time as well as the design of new ones. In this paper, we address this problem by systematically identifying, categorizing, and unifying operations from contemporary tools and extending them to cope with both variability dimensions. Based on our gained insights, we identify gaps and trade-offs in current tools for managing variability in space and time, and discuss open challenges. The unified operations establish a common foundation that helps researchers and practitioners to gain a deeper understanding of existing techniques and tools for managing variability in space and/or time, analyze and compare them, and design new ones.

CCS CONCEPTS

• **Software and its engineering → Software version control**; **Software product lines**; **Software configuration management and version control systems**.

KEYWORDS

variability, product lines, version control

ACM Reference Format:
Sofia Ananieva, Sandra Greiner, Jacob Krüger, Lukas Linsbauer, Sten Grüner, Timo Kehrer, Thomas Kühn, Christoph Seidl, Ralf Reussner. 2022. Unified Operations for Variability in Space and Time. In *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS '22), February 23–25, 2022, Florence, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3510466.3510483

1 INTRODUCTION

Industries producing variant-rich systems, such as the automotive industry, need to keep track of different variations and their evolution, for instance, to track faulty variations in past and future system generations. Yet, simply combining existing techniques for managing either variability dimension, namely variability in space (concurrent variations of a system) and variability in time (sequential generations of a system), is insufficient, since developers need to deal with a heterogeneous tool landscape–which limits cross-dimensional variability modeling and analyses. *Variability in space* allows to customize a system to different requirements based on features and is studied in the field of software product-line engineering (SPLE) [4, 11, 46]. *Variability in time* refers to evolutionary changes of a system, such as bug fixes or optimizations, and is the focus of software configuration management (SCM) [14] and version-control systems (VCSs) [50]. Dealing with both dimensions of variability is essential for maintaining long-living systems [8, 26, 32, 41, 55, 58], such as automotive systems. While some variation control systems (VarCSs) attempt to manage both variability dimensions, their behavior varies widely [33]. For instance, these tools employ different concepts or operations following different modalities and paradigms. Therefore, it is difficult to compare approaches or build tools that combine capabilities for both dimensions.

Although we recently proposed a conceptual model for unifying concepts of both dimensions [2], a common understanding for the operational management of these dimensions is still missing. For example, developers need to manage the evolution of the entire system as well as of individual features to track volatile features or revert faulty features to previous revisions (instead of the entire system). Understanding contemporary tools that manage variability in space, time, or both, as well as comparing their functionalities is essential to support industry, tool builders, and researchers, avoid redundant development, gain knowledge about different ways to operate on both dimensions, and define adequate new tool functionality.

In this paper, we contribute unified operations that serve as a conceptual foundation to address these needs. In detail, we offer the following contributions: (a) We *identify operations for managing variability in space, time, and both* by studying ten contemporary tools. (b) We *unify the identified operations*, using the conceptual model [2] as common data structure. We not only combine the behavior of existing tools, but also extend it to support both variability dimensions simultaneously. (c) We *analyze* the operations' feasibility, *identify gaps and trade-offs* in current tool support, and *discuss open challenges*. Moreover, we *publish an open-access repository* comprising our artifacts (e.g., anonymized questionnaire responses).[1] With these contributions, we aim to provide a common ground for researchers as well as practitioners in the areas of SPLE and SCM to analyze, compare, design, and implement techniques for managing variability in space and time.

VAMOS '22, February 23–25, 2022, Florence, Italy

ACM ISBN 978-1-4503-9604-2/22/02…$15.00
https://doi.org/10.1145/3510466.3510483

[1] https://doi.org/10.5281/zenodo.5825135

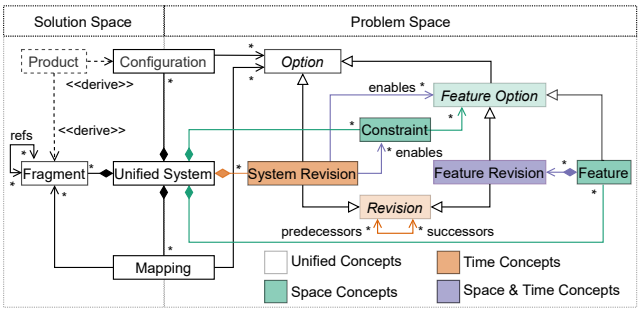

Figure 1: The unified conceptual model [2].

Concrete advantages of a tool that supports the unified operations are: i) a more homogeneous tool landscape, since a single tool suffices to manage both dimensions, which are highly intertwined; ii) tracking revisions (time) per feature (space) enables additional functionalities, such as rolling back individual (faulty) features instead of a system-wide rollback; and iii) analyses across both dimensions, such as the volatility (i.e., frequency of change) of each feature, no longer require expensive or approximate mining techniques. These advantages enable new possibilities for DevOps pipelines, as changes are precisely mapped onto features, which, for example, enables rejecting commits that modify features or products that were not supposed to be modified.

2 BACKGROUND

In this section, we explain the variability dimensions and provide an overview of the unified conceptual model.

Variability Dimensions. In SCM [35], a VCS [50] manages the evolution of a system via *revisions* that represent the state of the system at different points in time, thus capturing *variability in time*. Moreover, a system may allow for different *configurations*. This introduces *variability in space*, which is addressed in SPLE [4, 11, 46]. A variability model [6, 13, 22, 39], such as a feature model, documents the features of a product line. Features specify common or distinguishing functionality of a product. A configuration is a selection that assigns values (typically Boolean) to the configurable features. To implement a variable system and derive customized products based on a configuration, a variability mechanism is needed, such as an annotative mechanism (e.g., a preprocessor) [4]. Although established tools exist to target variability in space (SPLE) or time (VCS) in isolation, both dimensions are highly intertwined [8, 60], requiring sophisticated support for their joint maintenance.

Unified Conceptual Model. We [2] have proposed a *conceptual model for unifying concepts of variability in space and time*, shown in Figure 1. The right side of the model covers concepts of the problem space (i.e., domain abstraction), while the left side covers concepts of the solution space (i.e., the implementation) [46]. The Unified System is the core concept containing most of the other concepts. Options represent abstract configuration possibilities in terms of Feature Options (i.e., Features and Feature Revisions) or System Revisions. Constraints restrict which combinations of Feature Options are valid. Fragments describe the implementation of a Unified System on an arbitrary level of granularity (e.g., a line of text or a file). Mappings connect Fragments and Options, for instance, via some form of logical expression. Deriving a Product from the Unified System requires a Configuration. Concepts belong to either variability in space (i.e., Feature Option, Feature, and Constraint), variability in time (i.e., Revision and System Revision), to both dimensions (i.e., Feature Revision), or are unified concepts of either variability in space, time, or both (i.e., Unified System, Option, Mapping, Configuration, Fragment, and Product).

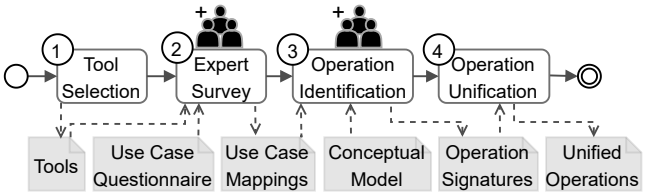

Figure 2: Unification process.

Table 1: Distinguishing concepts of the selected tools.

Tool \ Concept	Feature	Constraint	Feature Revision	System Revision
FeatureIDE [23, 36]	●	●	–	–
VTS [57]	●	–	–	–
SiPL [24, 43, 44]	●	●	–	–
SVN [45]	–	–	–	●
Git [34]	–	–	–	●
SuperMod [52, 53]	●	●	–	●
DarwinSPL [40]	●	●	–	●
DeltaEcore [54, 55]	●	●	●	–
ECCO [15, 16, 31]	●	–	●	–
VaVe [3]	●	●	●	–

3 SCOPE AND UNIFICATION PROCESS

In this section, we set the scope of this work and explain the employed unification process shown in Figure 2.

Scope. We are concerned with *tool-based variability management* [4]: We considered tools that i) *manage variability in space, time, or both*; and ii) *implement concepts of both problem and solution space*. We considered operations that i) operate on *the same level of abstraction as the unified conceptual model* [2]; and ii) *modify the system or create mutable output from it*.

Tool Selection (Step ①). Recent studies [2, 20, 33] investigated relevant tools, which influenced our tool selection. Table 1 shows our tool selection and the distinguishing concepts. FeatureIDE, VTS, and SiPL manage variability in space via features and (except for VTS) constraints. SVN and Git manage variability in time via system revisions. SuperMod and DarwinSPL support variability in space and time via features, constraints, and system revisions. DeltaEcore, ECCO, and VaVe manage variability in space and time via feature revisions. We consider combinations of tools (e.g., FeatureIDE and Git) as implicitly covered by inspecting each tool individually.

Expert Survey (Step ②). We conducted a survey with experts of the selected tools based on questionnaires. For each tool, we invited one expert that was involved in the conception or implementation of the tool. All of the tool experts were researchers from academia. By involving tool experts, we elicited current, detailed, and reliable information. We asked whether and how a given set of use cases can be covered by a tool, for inputs, outputs, pre and post-conditions of

Table 2: *Categorization:* Predicates.

Predicate \ Tool	FeatureIDE	VTS	SiPL	SVN	Git	SuperMod	DarwinSPL	DeltaEcore	ECCO	VaVe
Complete Configuration	●	–	●	●	●	●	●	●	–	●
Valid Configuration	●	–	●	●	●	●	●	●	–	●
Well-Formed Product	–	–	–	–	–	●	●	●	–	–
Valid Expression[1]	–	–	–	–	–	–	–	–	–	–

Predicate is either evaluated ● or not –. [1]Required for unified operations.

each use case, a description of its semantics in the tool, and for any further use cases a tool may address. Exemplary use cases are the addition of a feature or constraint, or the integration of a modified product into a unified system. For SVN and Git, we completed the questionnaires ourselves due to their detailed documentations. This step resulted in a *use case mapping* per tool.

Identification (Step ③). Based on the use case mappings and feedback of the tool experts, we collected operations for managing variability in space, time, or both from each tool. We mapped the inputs and outputs to the concepts of the unified conceptual model to categorize operations according to their signatures and semantics. To avoid redundancy and ambiguity, we defined categories with each covering a single concern only (e.g., one for the derivation of a product and one for the derivation of a partial product line), named each category, and specified the signature of its operation (i.e., name, input, and output). We used the same procedure to identify predicates for pre and post-conditions of tool operations.

Unification (Step ④). Based on the identified and categorized operations, we unified the behavior of operations (including the predicates used in pre- and post-conditions) to combine the capabilities of existing tools while avoiding redundancy and ambiguity. Furthermore, we extended them to both variability dimensions in cases where only one was supported.

4 IDENTIFIED AND UNIFIED OPERATIONS

While examining the behavior of the identified operations and discussing with the tool experts, we found that operations can be classified according to the *edit modality* and *edit paradigm*. The *edit modality* describes how a unified system can be edited: either directly (*direct editing*) or via well-defined views [5, 10] (*view-based editing*). The *edit paradigm* describes the development of a unified system. *Product-oriented development* is closer to clone & own [15, 28, 48] as a user focuses on a single product. *Platform-oriented development* is closer to traditional SPLE, since the entire platform (i.e., the unified system) must be considered. In this section, we present the identified predicates and operations (Step ③ in Figure 2), discuss their commonalities and differences, and describe their unification (Step ④ in Figure 2).

4.1 Predicates

Predicates evaluate to *true* or *false* and are used in pre and post-conditions of operations. Table 2 categorizes the identified predicates. Figure 3 shows the definitions of all unified predicates.

Predicate: Complete Configuration. Checks whether all options (i.e., features and revisions) are bound in a configuration. Intuitively,

complete

Predicate: **Complete Configuration**
Input: Unified System US, Configuration c

Configuration c is complete, if and only if it selects one system revision sr, either selects or deselects every feature f_{sr} enabled by sr, and selects at least one feature revision $fr_{f_{sr}}$ enabled by sr for each selected feature f_{sr}.

valid

Predicate: **Valid Configuration**
Input: Unified System US, Configuration c

Configuration c is valid, if and only if all features in c exist in US and are either selected $f \in c$ or deselected $\neg f \in c$ and never both, for any deselected feature $\neg f \in c$ no feature revision fr_f is selected in c, if c selects a system revision sr in US, then

- all selected features $f \in c$ and feature revisions $fr_f \in c$ are enabled by sr, and
- for no constraint ct_{sr} enabled by sr, formula $c \wedge ct_{sr}$ is unsatisfiable.

validExpr

Predicate: **Valid Expression**
Input: Unified System US, System Revision sr, Formula e

An arbitrary propositional formula e over feature options is valid for a given system revision sr, if and only if there is no constraint ct_{sr} enabled by sr where $e \wedge ct_{sr}$ is unsatisfiable.

wellformed

Predicate: **Well-Formed Product**
Input: Product p

Product p is well-formed, if and only if no fragment $ft \in FT_p$ references a fragment $ft' \notin FT_p$, where FT_p denotes the fragments from which p was constructed.

Figure 3: Overview of the predicates.

a *Complete* (or *full*) *Configuration* is one for which no options can (or should) be (de)selected anymore. An incomplete configuration is referred to as *partial*. While the semantics are well-understood in space and used uniformly in SPLE, they are not obvious when also considering the time dimension. In FeatureIDE and SiPL, a configuration is *complete in space* if every feature in the unified system is either selected or deselected. In SVN and Git, a configuration is *complete in time* if exactly one system revision is selected. In DeltaEcore, ECCO, and VaVe, which combine variability in space and time via feature revisions, a configuration is complete if every feature is either selected or deselected and, for every selected feature, exactly one feature revision is selected. ECCO allows to select more than one feature revision per selected feature for merging feature revisions. In SuperMod and DarwinSPL, which combine variability in space and time via features and system revisions, a configuration is complete if exactly one system revision is selected and every enabled feature in that system revision is either selected or deselected.

Predicate: Valid Configuration. Checks whether a configuration violates any constraints. Intuitively, a *Valid Configuration* is one that does not violate any (explicit or implicit) constraints. Again, the semantics are well-understood in space where constraints are specified explicitly (e.g., via a variability model), but are not obvious in time. In tools supporting variability in space, a configuration (a set of selected and deselected features) is valid if it does not violate any of the explicitly specified constraints. In tools supporting variability in time, a configuration (a selected system revision) is valid if the selected system revision exists in the unified system. Tools supporting variability in space and time via features and feature

Table 3: *Categorization:* Direct editing operations.

Operations per Concept	Add / Update / Delete	FeatureIDE	VTS[2]	SiPL[1]	SVN	Git	SuperMod[1]	DarwinSPL	DeltaEcore	ECCO[2]	VaVe[2]
Mapping	A, U, D	●	●	●	○	○	○	●	●	○	○
Fragment	A, U, D	●	●	●	○	○	○	●	●	○	○
Feature	A	●	●	●	—	—	○	●	●	○	○
	U, D	●	●	●	—	—	○	○	●	○	○
Feature Revision	A, U, D	—	—	—	—	—	—	—	●	○	○
System Revision	A, U, D	—	—	—	○	○	○	○	—	—	—
Constraint	A	●	—	●	—	—	○	●	●	—	○
	U, D	●	—	●	—	—	○	○	●	—	○
Configuration	A, U, D	●	—	●	○	○	—	●	●	—	—

Direct editing supported ●, not supported ○, or concept does not exist —.
[1]Mapping is part of fragment. [2]Fragment is part of mapping.

revisions extend configurations to additionally consist of feature revisions. Tools supporting variability in space and time via features and system revisions require all selected features to be enabled by the selected system revision and only those constraints that are enabled by the selected system revision must not be violated.

Predicate: Well-Formed Product. Checks whether a product's implementation (i.e., a set of fragments) is well-formed with respect to a set of rules (e.g., a grammar or meta-model) specific to the type of fragment. For example, if fragments represent a UML model, conformance with the corresponding meta-model and OCL constraints could be verified. For Java code, its syntactic validity could be checked. The only tools evaluating well-formedness of products are SuperMod, DarwinSPL, and DeltaEcore. We consider well-formedness independently of the type of fragment and express it on the abstraction level of the conceptual model, where the only structural information is the references between fragments.

Predicate: Valid Expression. Checks whether an expression over feature options (i.e., features and feature revisions) violates any constraints. Interestingly, it is not evaluated by any tool, but needed for the unified operation *iC*.

4.2 Direct Editing Operations

Table 3 categorizes the identified direct editing operations: one add, update, and delete operation per concept of a unified system. Tools that support direct editing and variability in space usually allow to add, update, and delete instances of the respective concepts. Tools that support variability in time do not allow direct editing at all. Tools that support both variability dimensions either do not allow direct editing at all or only for the space dimension. For example, DarwinSPL permits direct additions for space concepts but no direct modifications or deletions to guarantee a reproducible history. An exception is DeltaEcore, which permits direct editing of both dimensions. Note that all direct editing operations are platform-oriented and do not exhibit complex behavior that would require unification.

4.3 View-Based Operations

Table 4 categorizes the identified view-based operations and classifies them according to the edit paradigms. *Internalize* operations modify the unified system based on a view. *Externalize* operations create output from the unified system. View-based operations are less flexible but offer a higher degree of automation than direct editing operations, since they essentially execute predefined sequences of direct-editing operations (e.g., adding a feature automatically creates a new revision and a corresponding mapping). Thus, they ensure certain properties (e.g., an acyclic revision graph), prevent modifications to past revisions (i.e., automatically creating new revisions to map changes to), and reduce *cognitive complexity*.

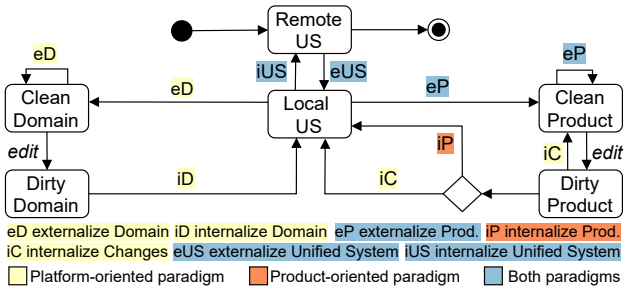

Figure 4: Execution sequences of unified operations.

While every operation is supported by at least one tool, there is no tool that supports all operations. Also, there is no tool that implements any of the operations for both system revisions and feature revisions. We found cases in which one operation of a tool covers two separate concerns. For instance, VTS uses the operation *get* to externalize a product (if a complete configuration is provided) or a subset product line (if a partial configuration is provided), and *put* to integrate the changes in both cases. SuperMod uses the operation *checkout* as a two-step process: First, to externalize a domain at a certain point in time (i.e., system revision), and, second, to externalize a product based on a feature selection via the externalized domain. Interestingly, Git's branch and merge operations do not fit within our scope criteria: The merge operation does not actually modify the unified system, since the actual merge point in the revision graph is created by the commit operation that follows the merge operation. Similarly, the branch operation only creates an alias for a commit, while the actual branch point in the revision graph is created by the following commit operation. Finally, we excluded niche functionalities, such as Git's cherry-picking, sub-trees, or sub-modules.

Figure 4 shows an overview of the unified view-based operations and possible execution sequences. We highlight product-oriented operations in orange, platform-oriented operations in yellow, and operations used in both paradigms in blue. A local instance of a unified system can be obtained from a remote unified system via *eUS*. To edit feature options and constraints of the local unified system, an *iD* operation must follow an *eD* operation. The operation *eD* creates a *clean* view on the domain and can be performed repeatedly to switch between different system revisions of the domain. *Editing* the domain view marks it as *dirty*. The modified view is then internalized via *iD*. The operation *eP* creates a *clean* view on the product and can be performed repeatedly to switch between different configurations. Performing *edits* marks the product as *dirty*. The modified product can be internalized into the local unified system via *iP* when following product-oriented development or via *iC* when following platform-oriented development. In the latter case, changes are integrated in a fine-grained manner based on an expression provided by the user. The *edit* and *iC* cycle can be performed repeatedly before transitioning back to the local unified system. The operation *iUS* integrates the contents of the local unified system into the remote unified system. Figure 5 shows the definitions of all unified view-based operations, which we discuss in the following.

Table 4: *Categorization:* View-based operations.

ID	Unified Operation Name	Paradigm	Feature-IDE	VTS	SiPL	SVN	Git	Super-Mod	Darwin-SPL	Delta-Ecore	ECCO	VaVe
eD	**Externalize Domain**	platform	–	–	–	–	–	checkout	getCopyOf-ValidModel	–	–	–
iD	**Internalize Domain**	platform	–	–	–	–	–	commit	-	–	–	–
eP	**Externalize Product**	both	compose	get	generate-Product	checkout	checkout	checkout	derive-Product	derive-Product	checkout	derive-Product
iP	**Internalize Product**	product	–	–	–	commit	commit	–	–	–	commit	–
iC	**Internalize Changes**	platform	–	put	–	–	–	commit	–	–	–	commit
eUS	**Externalize Unified System**	both	deriveSubset-ProductLine	get	–	–	clone	–	–	–	clone	–
iUS	**Internalize Unified System**	both	–	put	–	–	pull / push	–	–	–	pull / push	–

Operation: **Internalize Domain** iD
Input: Unified System US, System Revisions SR, Feature Options FO, Constraints CT

Integrates the sets of feature options FO and constraints CT into the unified system US. Creates a new system revision sr' that is added as successor of each system revision $sr \in SR$, creating a merge point at sr' if $|SR| > 1$, and a branch point if sr has a successor. All feature options FO and constraints CT are enabled by sr'. Creates new mappings $m_{sr'} = m_{sr}$ for all mappings m_{sr} in US with $sr \in SR$.

Operation: **Externalize Domain** eD
Input: Unified System US, System Revisions SR
Output: Feature Options FO, Constraints CT

Returns the sets of feature options $FO = \bigcup_{sr \in SR} FO_{sr}$ and constraints $CT = \bigcup_{sr \in SR} CT_{sr}$, where FO_{sr} and CT_{sr} are the feature options and constraints enabled by the system revision $sr \in SR$ in the unified system US.

Operation: **Externalize Product** eP
Input: Unified System US, Configuration c
Pre-condition: valid(US, c) ∧ complete(US,c)
Output: Product p
Post-condition: wellformed(p)

Creates a well-formed product p from a complete and valid configuration c. Selects all mappings $M' = \{m' \in M \mid c \Rightarrow m'\}$ from the mappings M in the unified system US implied by c and collects their fragments $FT_{m'}$ into $FT_p = \bigcup_{m' \in M'} FT_{m'}$ to create the product p.

Operation: **Internalize Product** iP
Input: Unified System US, Product p
Pre-condition: wellformed(p) ∧ valid(US, c_p)

Updates the unified system US to additionally cover product p. Creates a new system revision sr' and adds it as successor of the system revision in c_p. Creates a new feature revision fr'_f for every feature f in configuration c_p that is either new (and added to US) or was changed in product p. Adds fr'_f as successor to every feature revision fr_f of f selected in c_p, creating a merge point if multiple were selected, and a branch point if fr_f has a successor. Enables all new and all unchanged features and feature revisions appearing in the configuration c_p. Adds all fragments FT_p of product p to US and adds new mappings from sr' to each fragment $ft \in FT_p$.

Operation: **Externalize Unified System** eUS
Input: Unified System US, Configuration c
Pre-condition: valid(US, c)
Output: Unified System US'

Creates a new unified system US' from the existing unified system US and the (partial) valid configuration c by selecting only those features, mappings, fragments, and revisions (including their predecessors) that are not contradicted by c.

Operation: **Internalize Unified System** iUS
Input: Unified System US, Unified System US'

Integrates another unified system US' into an existing unified system US by merging their fragments, mappings, features, constraints, and revisions (including their relations) creating their union.

Operation: **Internalize Changes** iC
Input: Unified System US, Product p, Expression e over Feature Options FO
Pre-condition: validExpr(US, sr_{c_p}, e) ∧ $(c_p \Rightarrow e)$ ∧ wellformed(p)

Integrates changes made to a product p (with a complete and valid configuration c_p) into the unified system US. Determines the set of fragments that were added FT^+, remained unchanged FT^o and were removed FT^- from product p. Creates a new system revision sr'. Creates new feature revision fr'_f enabled by sr' for each positive feature f appearing in expression e. Adds sr' as successor to the system revision sr in c_p, such that it enables the same features and feature revisions as sr (except those succeeded by any of the new feature revisions). Adds each new feature revision fr'_f as successor to every feature revision fr_f of f selected in c_p, creating a merge point if multiple were selected, and a branch point if fr_f has a successor. Creates new mappings $m'_{sr',ft}$ for every fragment ft based on its mapping $m_{sr,ft}$ in the previous system revision sr, such that $m'_{sr',ft^+} = m_{sr,ft^+} \vee e$, for each $ft^+ \in FT^+$; $m'_{sr',ft^o} = m_{sr,ft^o}$, for each $ft^o \in FT^o$; and $m'_{sr',ft^-} = m_{sr,ft^-} \wedge \neg e$, for each $ft^- \in FT^-$.

Platform-oriented paradigm Product-oriented paradigm Both paradigms

Figure 5: Overview of the unified operations.

Operation: Externalize Domain (*eD*). Produces a view of the domain (i.e., feature options and constraints) at one or multiple points in time (i.e., system revisions) that can be edited, merged, or used to create configurations. It is supported by SuperMod (as part of the *checkout* operation) and DarwinSPL. The behavior of the two tools coincides and considers both variability dimensions via features and system revisions. Note that we allow multiple system revisions as input to support merging of revisions.

Operation: Internalize Domain (*iD*). Integrates changes performed on a view of the domain (i.e., feature options and constraints produced by the operation *eD*) into the unified system. It is supported only by SuperMod. While DarwinSPL supports the creation of views on the domain via *eD*, it does not provide the view-based operation *iD* for modifying the domain. For the unification, we do not allow the user to add new feature revisions, so that only feature revisions of the previously externalized view may be used to formulate

constraints. If a system revision *sr* specified during *eD* already had a successor, it becomes a branch point (i.e., has multiple successors). If multiple system revisions were specified during *eD*, the new system revision *sr′* becomes a merge point (i.e., has multiple predecessors).

Operation: Externalize Product (*eP*). Produces a product (i.e., view) from the fragments contained in the unified system based on a complete and valid configuration. In space, this is usually referred to as *product derivation* while in time it is referred to as *checkout*. It is supported by *all* tools, and in essentially the same manner. First, mappings whose expressions are satisfied by the configuration are selected. The differences between tools are whether configurations and mappings contain system revisions, features, feature revisions, or any combination thereof. Afterwards, the product is constructed using the fragments in the selected mappings. In SVN or Git, only products that have been explicitly internalized before can be externalized via *iP* (referred to as extensional versioning [12]), since the only available option concept is that of a system revision (of which only one can appear in a valid configuration). Tools that support platform-oriented development (e.g., via direct editing or the *iC* operation) can externalize products that have not been internalized explicitly before (referred to as intensional versioning [12]), since multiple options can be combined in valid configurations. An exception is ECCO, which does not support platform-oriented development, but still supports intensional versioning. For the selection of mappings (and consequently fragments), ECCO performs feature location [37] to identify required fragments in previously internalized products and reuses them in new products during externalization.

Operation: Internalize Product (*iP*). Integrates a product into the unified system. It is supported by the tools SVN, Git, and ECCO. All three tools create a new system revision *sr′* and then map the fragments of the internalized product to *sr′*. For SVN and Git, which deal only with variability in time, this is all that is needed. ECCO supports variability in space and time (via features and feature revisions) and requires that modified features are marked in a product's configuration to indicate that a new feature revision must be created for them. The new system revision enables exactly the feature options that appear in the configuration of the product. Thereby, the feature revisions are tracked explicitly via the unified operation, in contrast to SVN and Git, where this information would have to be documented manually in the commit message. If more than one revision of the same feature is selected in the externalized product, the new feature revision becomes a successor to all of them, and thus a merge point in the respective feature revision graph. If the system revision or any of the feature revisions in the configuration of the externalized product already had a successor, they become branch points in their respective revision graphs.

Operation: Internalize Changes (*iC*). Integrates changes performed on a product into the unified system based on a manually provided expression. It is supported by SuperMod, VTS, and VaVe. The user provides as input an expression over features, which SuperMod and VTS refer to as *ambition*. In SuperMod, the expression is essentially a partial configuration (i.e., conjunction of positive or negative features); in VTS, it is an arbitrary expression over features; and in VaVe, it consists of a single feature. SuperMod ensures that the expression does not violate any constraints via the *Valid Configuration* predicate. VTS does not ensure the validity of the expression, but if it did, this predicate would not suffice, since the expression can be arbitrary. Therefore, the predicate *Valid Expression* is required to allow the combination of the behavior of all tools in our unification. Interestingly, SuperMod and VTS enforce opposing pre-conditions. In VTS, the expression e for *iC* must imply the configuration c_p of product p ($e \Rightarrow c_p$) to prevent the user from affecting configurations that are not visible in the current view. The exact opposite is the case in SuperMod ($c_p \Rightarrow e$), where changes made on a product must affect *at least* that product. The suitability of the VTS pre-condition for the unified *iC* operation is arguable. Since, per definition of *eP*, a view represents a product based on a complete configuration, the pre-condition of VTS boils down to $e \Leftrightarrow c$. This would limit the effect an edit can have to the exact product on which it was performed. Therefore, we exclude this pre-condition from the unification. Despite the conflicting pre-conditions, the intention and behavior of this operation across the tools is essentially the same. Each tool computes the added, unchanged, and removed fragments and adds or updates the respective mappings. If more than one feature revision of the same feature was externalized in the product, the new feature revision becomes a merge point in the feature revision graph. If the system revision or any of the feature revisions of the externalized product already had a successor, they become branch points in their respective revision graphs.

Operation: Externalize Unified System (*eUS*). Derives a new instance of a unified system US' that is a full or partial copy of the original unified system US, depending on the given (complete or partial) configuration c. It is supported by FeatureIDE, Git, VTS, and ECCO. The tools dealing with variability in space behave identically. Feature options without assigned values (i.e., neither selected nor deselected in the provided partial configuration) remain variable. Feature options with a positive value (i.e., selected) are retained and set to *true*. Feature options with a negative value (i.e., deselected) are removed and substituted by *false* in mapping expressions and constraints. Mappings whose expression cannot be satisfied anymore (i.e., contradicts configuration c) and the corresponding fragments are removed. In Git, only system revisions that are selected in the configuration and their ancestors are retained. If no system revision is selected, then all revisions are retained. In our unification, we transfer the behavior from Git to also feature revisions, since there is no conflicting behavior in how ECCO deals with feature revisions.

Operation: Internalize Unified System (*iUS*). Integrates another instance of a unified system US' into the current unified system US, essentially creating their union. It is supported by the tools Git, VTS, and ECCO. In Git, it maps to the *push* and *pull* operations. It merges all system revisions, including their predecessors and successors. In ECCO, all features and feature revisions are merged. In ECCO and Git, all fragments and mappings are merged. In VTS, this operation corresponds to any execution of the *put* operation that follows a *get* operation with a partial configuration as parameter. In this case, the *put* operation simply overwrites fragments and mappings in the unified system US with the ones in the unified system US' (which is not a desirable behavior for the unification).

```
1 #ifdef USE_RAM
2   #ifdef CACHE_NAMES
3     char sort_short[SORT_LIMIT][FILE_NAME_LIMIT];
4   #endif
5 #endif
6 uint8_t sort_order[SORT_LIMIT];
```

Listing 1: Initial revision.

```
1 #ifdef USE_RAM
2   #ifdef CACHE_NAMES
3     #ifdef DYNAMIC_RAM
4       char **sort_names, **sort_short;
5     #else
6       char sort_names[SORT_LIMIT][FILE_NAME_LIMIT];
7       char sort_short[SORT_LIMIT][FILE_NAME_LIMIT];
8     #endif
9   #endif
10 #endif
11 #ifdef DYNAMIC_RAM
12   uint8_t* sort_order;
13 #else
14   uint8_t sort_order[SORT_LIMIT];
15 #endif
```

Listing 2: Final revision.

5 ILLUSTRATING EXAMPLE

We now discuss an illustrating example based on an adapted excerpt of the highly variable 3D-printer firmware Marlin [33]. The initial revision of the system is shown in Listing 1. In the solution space, each line of code is a Fragment. In the problem space, macros used in the preprocessor annotations are Options, namely the Features USE_RAM (UR) and CACHE_NAMES (CN). The first system revision enables the features CN and UR as well as the constraint that name caching requires the use of RAM ($CN_1 \Rightarrow UR_1$). The preprocessor annotations represent Mappings that connect solution and problem space. For instance, the array sort_order in Line 6 of Listing 1 is present in the first system revision in each configured product. The array sort_short in Line 3 is introduced in the same system revision, but only present if the features CN and UR are selected in their first revision. The final revision in Listing 2 allows for the use of dynamic RAM as an additional option where memory is allocated dynamically by using pointers (Lines 4 and 12). In addition, an array for sorting names is added in Line 6, which modifies the implementation of the features UR and CN, and, thus, can be considered a revision of these features. Next, we demonstrate how the unified operations can be used to gradually transition from the initial state US_{init} to the final state US_{final}.

Change domain. First, we extend the domain with the new feature DR and a corresponding constraint. We create an editable view on the domain at system revision sr_1 by executing the operation $eD(US_{init}, \{sr_1\})$. This results in the set of feature options $\{UR_1, CN_1\}$ and the set of constraints $\{CN_1 \Rightarrow UR_1\}$. We add the feature DR and the constraint $DR \Rightarrow UR_1$, and internalize the updated domain view via $iD(US_{init}, \{sr_1\}, \{UR_1, CN_1, DR\}, \{CN_1 \Rightarrow UR_1, DR \Rightarrow UR_1\})$. This operation adds a new system revision sr_2 to the unified system as successor of sr_1 while enabling DR, the respective constraint, as well as all unmodified features and constraints. Additionally, all mappings containing sr_1 are copied and sr_1 is replaced by sr_2 in their expressions.

Change implementation via product-oriented editing. We then use $eP(US_2, \{sr_2, UR_1, CN_1, DR\})$ to obtain a product p comprising Lines 3 and 6 in Listing 1 to which we can add the implementation of DR. We delete Lines 3 and 6 in Listing 1 from product p and add Lines 4 and 12 in Listing 2 to obtain the modified product p', and update its configuration (to mark the modified feature DR) from $\{UR_1, CN_1, DR\}$ to $\{UR_1, CN_1, DR_*\}$. We then execute $iP(US_2, p')$ to internalize the modified product p', which leads to the new system revision sr_3 that enables the new feature revision DR_1 of feature DR. All fragments of the internalized product are mapped to the new system revision. To add Line 6 in Listing 2, we externalize another product p_2 via $eP(US_3, \{sr_3, UR_1, CN_1, \neg DR\})$, which contains Lines 7 and 14. We then add Line 6 to product p_2 to obtain the modified product p_2'. We modify its configuration from $\{UR_1, CN_1, \neg DR\}$ to $\{UR_1, CN_*, \neg DR\}$ and use the operation $iP(US_3, p_2')$ for internalization. This leads to the new system revision sr_4 that enables the new feature revision CN_2 of feature CN.

Change implementation via platform-oriented editing. Alternatively to iP, we can use iC to internalize more fine-granular changes to a product. We start with the same product p, delete both Lines 3 and 6 in Listing 1 and add Line 12, but not yet Line 4 in Listing 2, to obtain the modified product p''. We internalize p'' with expression DR via $iC(US_2, p'', DR)$. This adds a new system revision sr_3 and the first feature revision DR_1 to feature DR. The new system revision sr_3 enables the new feature revision DR_1 and the latest revisions UR_1 and CN_1 of the unmodified features UR and CN. The new line is added to the unified system and mapped to DR_1. The mapping of the two deleted lines is appended with $\wedge\neg DR$, resulting in mappings $((sr_1 \vee sr_2) \wedge UR_1 \wedge CN_1) \vee (sr_3 \wedge UR_1 \wedge CN_1 \wedge \neg DR)$ and $sr_1 \vee sr_2 \vee (sr_3 \wedge \neg DR)$, respectively. We then edit product p'' further by adding Line 4 to obtain product p''' and internalize it via $iC(US_3, p''', UR_1 \wedge CN_1 \wedge DR_1)$. This creates another system revision sr_4, a second feature revision for each feature, and maps the new line to $UR_2 \wedge CN_2 \wedge DR_2$. Finally, we externalize another product p_2 via $eP(US_3, \{sr_3, UR_1, CN_1, \neg DR\})$, which contains Lines 7 and 14, and add Line 6 to obtain the modified product p_2'', which we internalize via $iC(US_4, p_2'', UR_2 \wedge CN_2 \wedge \neg DR)$. This leads to yet another new system revision sr_5 and the new feature revisions UR_3 and CN_3, but no new feature revision for DR (as it appears negated). The newly added line is then mapped to the expression $sr_5 \wedge UR_3 \wedge CN_3 \wedge \neg DR$.

Distribution of features. Starting from the final unified system US_{final} in Listing 2, we derive another unified system containing the features UR and CN. By executing $eUS(US_{final}, \{\neg DR\})$, we obtain a unified system US_{ext} containing all system revisions, features UR and CN, and all of their feature revisions. Feature DR is contradicted by the provided partial configuration and not retained. Similarly, the mappings for Lines 4 and 12 are contradicted, and therefore also excluded together with the Lines (i.e., fragments) themselves. Lines 6, 7, and 14 are included, since their mappings are not contradicted by the partial configuration. Additionally, mapping $sr_5 \wedge UR_3 \wedge CN_3 \wedge \neg DR$ (Line 6) is simplified to $sr_5 \wedge UR_3 \wedge CN_3$, mapping $sr_1 \vee sr_2 \vee (sr_3 \wedge \neg DR)$ (Line 14) is simplified to $sr_1 \vee sr_2 \vee sr_3$, and mapping $((sr_1 \vee sr_2) \wedge UR_1 \wedge CN_1) \vee (sr_3 \wedge UR_1 \wedge CN_1 \wedge \neg DR)$ is simplified to $((sr_1 \vee sr_2 \vee sr_3) \wedge UR_1 \wedge CN_1)$. We internalize the previously externalized unified system US_{ext} into the initial unified

system US_{init} to let it benefit from the new revisions of features UR and CN. We execute $iUS(US_{init}, US_{ext})$, which extends the initial unified system with the new system revisions up to sr_5, feature revisions UR_2, UR_3, CN_2, and CN_3, and the corresponding Line 6 in Listing 2 that maps to $sr_5 \wedge UR_3 \wedge CN_3$.

6 DISCUSSION

The illustrating example shows how to apply the unified operations. They support the same functionality as the analyzed tools and advance the state of the art by providing capabilities for the unified management of variability in space and time. Based on our gained insights during the unification and while applying the operations to the example, we identified trade-offs and open challenges in current tool support that also impact the operations. In the following, we discuss these trade-offs and provide guiding ideas.

Combination of Edit Paradigms. In none of the studied tools, the developer is able to alternate between platform and product-oriented editing. While projectional editing [7, 38, 51, 59], as implemented in VTS, allows developers to switch between editable product (*eP*) and (partial) platform (*eUS*) views, the integration of the changes is always performed in a platform-oriented manner (*iC*). We suggest to allow to arbitrarily alternate between both paradigms, since each has its distinct advantages. While platform-oriented development supports a user in modifying the entire platform, this task is cognitively demanding [33]. Product-oriented development alleviates the developer from this burden, but has limited support for intensional versioning [12]). Allowing the developer to switch paradigms and choose the more suitable paradigm for a specific development scenario would provide substantial benefits. The challenge is that, during product-oriented development, no fine-grained mapping expressions are provided by the user. Therefore, additional techniques are required, such as feature location [30, 37, 47] or feature trace recording [1, 9, 25].

Combination of Edit Modalities. VTS is the only tool we studied that supports both view-based and direct editing. While mappings can be edited arbitrarily via direct editing, editing mappings is not well-supported via any view-based operation. Therefore, we suggest to combine both edit modalities, but discourage direct editing of concepts for variability in time (i.e., feature revisions and system revisions) to prevent a (accidental) corruption of the history and guarantee its preservation. The challenge is that direct editing is only conveniently possible if an adequate and editable view, including mappings (as opposed to a partial view without variability, such as a product), of the contents of the unified system can be provided. In VTS, this is the case, since it is limited to lines of text (as fragments) and annotations (as mappings), which can easily be edited directly. Some tools provide IDE support for dynamically switching between editable product and different (partial) platform views but are also limited to textual fragments [7, 38]. However, in cases where different types of fragments are allowed, a universal view that also includes editable mappings cannot be provided easily.

Threats to Validity. A threat to the construct and external validity are the analyzed tools. While they follow different ideas, concepts, modalities, and paradigms, there may still be other tools and operations, and thus ours may not be fully generalizable. A threat to the internal validity is that we performed the unification on the abstraction level of the unified conceptual model, and thus may have missed details regarding the behavior of tools. We mitigated this threat by closely involving the tool experts and ensuring that the unified operations can still cover the same edits as the individual tools.

7 RELATED WORK

Several researchers elicited or defined processes, patterns, or operations for the evolution of variability in space and in time [12, 17, 19, 21, 27–29, 42, 49, 56, 57]. Often, these operations have been derived from observations in practice or from an individual tool. For instance, Rubin et al. [49] derived a set of operations for re-engineering cloned variants into a product-line. Hinterreiter et al. [19] introduced local and distributed operations and scenarios for feature-oriented development and evolution specifically in the industrial automation domain. As their work is based on ECCO, these operations are covered by our unification. Projectional editing [59] is the foundation of VTS and introduces (partial) views on variable systems. Closely related to our work, Linsbauer et al. [33] compare variation control systems, and identified the two general types of operations *internalization* and *externalization*. We aligned our work with these general types, but extended them considerably, for instance, to distinguish between *Internalize Changes* or *Internalize Product*. Arguably, the closest work to ours is the uniform version management proposed by Westfechtel et al. [60], combining variability in space and time into one model. While it represents research from two decades ago, the rather recent tool SuperMod included in our analysis uses it as a foundation and builds on its concepts. Hinterreiter et al. [18] compared and harmonized approaches dealing with temporal feature modeling and also considers some of our studied tools. However, they focus on feature modeling to represent domain constraints, and thus are less generic.

8 CONCLUSION

In this paper, we presented unified operations for managing variability in space and time, which we systematically devised based on a diverse set of tools. We provide a foundation for researchers and practitioners to classify and compare their work, and guide the design of novel techniques dealing with variability in space and time. We identified gaps and trade-offs in current tools and discussed open challenges. While direct editing offers the most flexibility and allows for arbitrary changes, view-based operations offer a higher degree of automation and are less prone to human errors. The alteration between different edit paradigms and modalities is potentially useful, but not yet supported by any tool. As future work, we plan to tackle the identified gaps and challenges, and to provide a reference implementation of the unified operations.

ACKNOWLEDGMENTS

This work has been supported by the German Research Foundation within the projects VariantSync (KE 2267/1-1) and EXPLANT (SA 465/49-3), the Federal Ministry of Economic Affairs and Energy (BMWi) following a decision of the German Bundestag in the context of the SofDCar project (grant agreement 19S21002I and 19S21002K), and a fellowship within the IFI programme of the German Academic Exchange Service (DAAD).

REFERENCES

[1] Hadil Abukwaik, Andreas Burger, Berima Kweku Andam, and Thorsten Berger. 2018. Semi-Automated Feature Traceability with Embedded Annotations. In *International Conference on Software Maintenance and Evolution*. IEEE, 529–533.

[2] Sofia Ananieva, Sandra Greiner, Thomas Kühn, Jacob Krüger, Lukas Linsbauer, Sten Grüner, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, Sebastian Krieter, Christoph Seidl, S. Ramesh, Ralf Reussner, and Bernhard Westfechtel. 2020. A Conceptual Model for Unifying Variability in Space and Time. In *International Systems and Software Product Line Conference*. ACM, 1–12.

[3] Sofia Ananieva, Heiko Klare, Erik Burger, and Ralf Reussner. 2018. Variants and Versions Management for Models with Integrated Consistency Preservation. In *International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 3–10.

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer.

[5] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. 2010. Orthographic Software Modeling: A Practical Approach to View-Based Development. In *Evaluation of Novel Approaches to Software Engineering*. Springer, 206–219.

[6] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *International Conference on Software Product Lines*. Springer, 7–20.

[7] Benjamin Behringer, Jochen Palz, and Thorsten Berger. 2017. PEoPL: Projectional Editing of Product Lines. In *International Conference on Software Engineering*. IEEE, 563–574.

[8] Thorsten Berger, Marsha Chechik, Timo Kehrer, and Manuel Wimmer. 2019. *Software Evolution in Time and Space: Unifying Version and Variability Management (Dagstuhl Seminar 19191)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik.

[9] Paul Maximilian Bittner, Alexander Schultheiß, Thomas Thüm, Timo Kehrer, Jeffrey M. Young, and Lukas Linsbauer. 2021. Feature Trace Recording. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1007–1020.

[10] Erik Johannes Burger. 2013. Flexible Views for View-Based Model-Driven Development. In *International Doctoral Symposium on Components and Architecture*. ACM, 25–30.

[11] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

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

[13] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *International Workshop on Variability Modeling of Software-Intensive Systems*. ACM, 173–182.

[14] Jacky Estublier. 2000. Software Configuration Management: A Roadmap. In *Conference on The Future of Software Engineering*. ACM, 279–289.

[15] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *International Conference on Software Maintenance and Evolution*. IEEE, 391–400.

[16] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *International Conference on Software Engineering*. IEEE, 665–668.

[17] Sten Grüner, Andreas Burger, Tuomas Kantonen, and Julius Rückert. 2020. Incremental Migration to Software Product Line Engineering. In *Conference on Systems and Software Product Line*. ACM, 1–11.

[18] Daniel Hinterreiter, Michael Nieke, Lukas Linsbauer, Christoph Seidl, Herbert Prähofer, and Paul Grünbacher. 2019. Harmonized Temporal Feature Modeling to Uniformly Perform, Track, Analyze, and Replay Software Product Line Evolution. In *International Conference on Generative Programming: Concepts and Experiences*. ACM, 115–128.

[19] Daniel Hinterreiter, Herbert Prähofer, Lukas Linsbauer, Paul Grünbacher, Florian Reisinger, and Alexander Egyed. 2018. Feature-Oriented Evolution of Automation Software Systems in Industrial Software Ecosystems. In *International Conference on Emerging Technologies and Factory Automation*. IEEE, 107–114.

[20] Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2019. Software Product Line Engineering: A Practical Experience. In *International Systems and Software Product Line Conference*. ACM, 164–176.

[21] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability with Embedded Annotations. In *International Conference on Software Product Line*. ACM, 61–70.

[22] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Carnegie-Mellon University.

[23] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A Tool Framework for Feature-oriented Software Development. In *International Conference on Software Engineering*. IEEE, 611–614.

[24] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 2013. Consistency-Preserving Edit Scripts in Model Versioning. In *International Conference on Automated Software Engineering*. IEEE, 191–201.

[25] Timo Kehrer, Thomas Thüm, Alexander Schultheiß, and Paul Maximilian Bittner. 2021. Bridging the Gap Between Clone-and-Own and Software Product Lines. In *International Conference on Software Engineering: New Ideas and Emerging Results*. 21–25.

[26] Jacob Krüger, Sofia Ananieva, Lea Gerling, and Eric Walkingshaw. 2020. Third International Workshop on Variability and Evolution of Software-Intensive Systems. In *International Systems and Software Product Line Conference*. ACM, 34:1–1.

[27] Jacob Krüger and Thorsten Berger. 2020. An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 432–444.

[28] Jacob Krüger, Wardah Mahmood, and Thorsten Berger. 2020. Promote-pl: A Round-Trip Engineering Process Model for Adopting and Evolving Product Lines. In *International Systems and Software Product Line Conference*. ACM, 2:1–12.

[29] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is My Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software* 152 (2019), 239–253.

[30] Jacob Krüger, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2017. Finding Lost Features in Cloned Systems. In *International Systems and Software Product Line Conference*. ACM, 65–72.

[31] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability Extraction and Modeling for Product Variants. *Software and Systems Modeling* 16, 4 (2017), 1179–1199.

[32] Lukas Linsbauer, Somayeh Malakuti, Andrey Sadovykh, and Felix Schwägerl. 2018. 1st International Workshop on Variability and Evolution of Software-Intensive Systems. In *International Systems and Software Product Line Conference*. ACM, 294–294.

[33] Lukas Linsbauer, Felix Schwägerl, Thorsten Berger, and Paul Grünbacher. 2021. Concepts of Variation Control Systems. *Journal of Systems and Software* 171 (2021), 110796.

[34] Jon Loeliger and Matthew McCullough. 2012. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly.

[35] Stephen A. MacKay. 1995. The State of the Art in Concurrent, Distributed Configuration Management. In *International Workshop on Software Configuration Management*. Springer, 180–193.

[36] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.

[37] Gabriela K. Michelon, David Obermann, Lukas Linsbauer, Wesley K. G. Assunção, Paul Grünbacher, and Alexander Egyed. 2020. Locating Feature Revisions in Software Systems Evolving in Space and Time. In *International Systems and Software Product Line Conference*. ACM, 14:1–14:11.

[38] Mukelabai Mukelabai, Benjamin Behringer, Moritz Fey, Jochen Palz, Jacob Krüger, and Thorsten Berger. 2018. Multi-View Editing of Software Product Lines with PEoPL. In *International Conference on Software Engineering*. ACM, 81–84.

[39] Damir Nešić, Jacob Krüger, Stefan Stănciulescu, and Thorsten Berger. 2019. Principles of Feature Modeling. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 62–73.

[40] Michael Nieke, Gil Engel, and Christoph Seidl. 2017. DarwinSPL: An Integrated Tool Suite for Modeling Evolving Context-Aware Software Product Lines. In *International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 92–99.

[41] Michael Nieke, Lukas Linsbauer, Jacob Krüger, and Thomas Leich. 2019. International Workshop on Variability and Evolution of Software-Intensive Systems. In *International Systems and Software Product Line Conference*. ACM, 320–320.

[42] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of Variability Models and Related Software Artifacts: A Fresh Look at Evolution Patterns in the Linux Kernel. *Empirical Software Engineering* 21, 4 (2016), 1744–1793.

[43] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. 2015. SiPL–A Delta-Based Modeling Framework for Software Product Line Engineering. In *International Conference on Automated Software Engineering*. IEEE, 852–857.

[44] Christopher Pietsch, Udo Kelter, Timo Kehrer, and Christoph Seidl. 2019. Formal Foundations for Analyzing and Refactoring Delta-Oriented Model-Based Software Product Lines. In *International Systems and Software Product Line Conference*. ACM, 207–217.

[45] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. 2008. *Version Control with Subversion: Next Generation Open Source Version Control*. O'Reilly.

[46] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[47] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering, Product Lines, Languages, and Conceptual Models*. Springer,

29–58.

[48] Julia Rubin and Marsha Chechik. 2013. A Framework for Managing Cloned Product Variants. In *International Conference on Software Engineering*. IEEE, 1233–1236.

[49] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2015. Cloned Product Variants: From Ad-Hoc to Managed Software Product Lines. *International Journal on Software Tools for Technology Transfer* (2015), 627–646.

[50] Nayan B. Ruparelia. 2010. The History of Version Control. *SIGSOFT Software Engineering Notes* 35, 1 (2010), 5–9.

[51] Johannes Schröpfer, Thomas Buchmann, and Bernhard Westfechtel. 2021. A Framework for Projectional Multi-variant Model Editors. In *International Conference on Model-Driven Engineering and Software Development*. SCITEPRESS, 294–305.

[52] Felix Schwägerl and Bernhard Westfechtel. 2016. SuperMod: Tool Support for Collaborative Filtered Model-Driven Software Product Line Engineering. In *International Conference on Automated Software Engineering*. ACM, 822–827.

[53] Felix Schwägerl and Bernhard Westfechtel. 2019. Integrated Revision and Variation Control for Evolving Model-Driven Software Product Lines. *Software and Systems Modeling* 18, 6 (2019), 3373–3420.

[54] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. DeltaEcore - A Model-Based Delta Language Generation Framework. In *Modellierung*. GI, 81–96.

[55] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. Integrated Management of Variability in Space and Time in Software Families. In *International Software Product Line Conference*. ACM, 22–31.

[56] Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. 2019. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *International Systems and Software Product Line Conference*. ACM, 177–188.

[57] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. 2016. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *International Conference on Software Maintenance and Evolution*. IEEE, 323–333.

[58] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Towards Efficient Analysis of Variation in Time and Space. In *International Workshop on Variability Modelling of Software-Intensive Systems*. ACM.

[59] Eric Walkingshaw and Klaus Ostermann. 2014. Projectional Editing of Variational Software. In *International Conference on Generative Programming: Concepts and Experiences*. ACM, 29–38.

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