A Conceptual Model for Unifying Variability in Space and Time

Sofia Ananieva,[1] Sandra Greiner,[2] Thomas Kühn,[3] Jacob Krüger,[4,5] Lukas Linsbauer,[6] Sten Grüner,[7] Timo Kehrer,[8] Heiko Klare,[3] Anne Koziolek,[3] Henrik Lönn,[9] Sebastian Krieter,[5,10] Christoph Seidl,[11] S. Ramesh,[12] Ralf Reussner,[3] Bernhard Westfechtel[2]
[1]FZI Research Center for Information Technology, [2]University of Bayreuth, [3]Karlsruhe Institute of Technology, [4]University of Toronto, [5]Otto-von-Guericke University, [6]Technical University of Braunschweig, [7]ABB Corporate Research Center Germany, [8]Humboldt University of Berlin, [9]Volvo Group Trucks Technology, [10]Harz University of Applied Sciences, [11]IT-University of Copenhagen, [12]General Motors Global R&D

ABSTRACT

Software engineering faces the challenge of developing and maintaining systems that are highly variable in space (concurrent variations of the system at a single point in time) and time (sequential variations of the system due to its evolution). Recent research aims to address this need by managing variability in space and time simultaneously. However, such research often relies on nonuniform terminologies and a varying understanding of concepts, as it originates from different communities: software product-line engineering and software configuration management. These issues complicate the communication and comprehension of the concepts involved, impeding the development of techniques to unify variability in space and time. To tackle this problem, we performed an iterative, expert-driven analysis of existing tools to derive the first conceptual model that integrates and unifies terminologies and concepts of both dimensions of variability. In this paper, we present the unification process of concepts for variability in space and time, and the resulting conceptual model itself. We show that the conceptual model achieves high coverage and that its concepts are of appropriate granularity with respect to the tools for managing variability in space, time, or both that we considered. The conceptual model provides a well-defined, uniform terminology that empowers researchers and developers to compare their work, clarifies communication, and prevents redundant developments.

CCS CONCEPTS

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

KEYWORDS

revision management, product lines, variability, version control

ACM Reference Format:
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, Bernhard Westfechtel. 2020. A Conceptual Model for Unifying Variability in Space and Time. In *24th ACM International Systems and Software Product Line Conference (SPLC '20), October 19–23, 2020, MONTREAL, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3382025.3414955

1 INTRODUCTION

Software-intensive systems have to exist in different variations to satisfy varying customer requirements [5, 19, 51, 64]. Such variations can be distinguished according to the two dimensions from which they originate [3, 15, 63]. First, *variability in space* refers to variations that are defined by feature options implemented in a system, which allow to mass-customize the system by enabling or disabling its features (user-visible functionalities of the system [5]). For example, the Linux kernel comprises over 15 000 feature options to customize a specific product; allowing it to run on small embedded systems, servers, or distributed computer clusters. The concepts of this dimension are of primary interest in the area of software product-line engineering (SPLE) [5, 51].

Second, *variability in time* refers to variations that occur due to the evolution of a system. Namely, a feature is only available after it has been implemented, and older revisions of the system can be deployed to exclude the feature. This results in variations in the system based on its evolution over time. The concepts related to this dimension have been investigated primarily in the area of software configuration management (SCM) [19] and in the context of version control systems (VCSs) [53].

SPLE and SCM are well-known research areas, and their combination has recently gained increasing attraction [11, 31, 36, 42, 63, 65]. However, one prerequisite for advancing their combination is an established and well-defined understanding of the concepts in both areas; especially considering the varying, but also synonymous, terminology. For example, the term "configuration" is used in SPLE to refer to a valid selection of features (we refer to as *feature configuration*), whereas the same term is used in SCM to refer to a specific revision (we refer to as *revision configuration*). In a previous paper [3], we presented an initial conceptual model attempting to capture the concepts of both research areas and their relations, but not their unification. Still, there are numerous problems caused by the two independent research areas of SPLE and SCM that require a conceptual model that unifies concepts and relations of both areas. For instance, recent literature reviews [8, 33, 46, 53] show a growth of research and tools originating from both areas that tackle the same problems with the same concepts. Such redundant development wastes resources and is caused by a missing understanding of the concepts, terminology, and advancements of the two research areas.

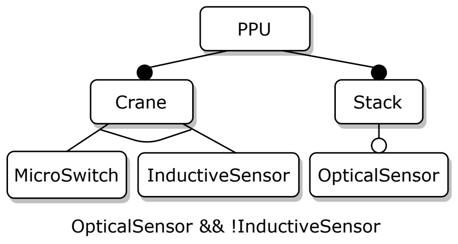

Figure 1: Simplified feature model of the PPU.

For that reason, we contribute a conceptual model for unifying variability in space and time, improving the current situation by establishing a common understanding and terminology of the concepts used in SPLE and SCM. We derive the conceptual model by systematically eliciting concepts used in 10 established tools from both research areas (e.g., Git [37], SVN [50], FeatureIDE [39], ECCO [21]). For this purpose, we interviewed developers and experts to identify the concepts and relations used in these tools. Based on this input, we conducted two workshops in which we constructed the conceptual model. We adopted four ontology-based metrics [24] to validate the granularity and coverage of the conceptual model compared to the tools. Our contributions are:

- We present a conceptual model for unifying the concepts of variability in space and time.
- We report an empirical validation of the conceptual model, showing how well it describes existing tools.
- We provide an open-access repository containing the anonymized interviews and evaluation data we used to construct the conceptual model.[1]

The conceptual model provides a foundation for the unified management of variability in space and time. It supports researchers and developers in scoping and communicating their research by unifying terminology and relating concepts of both research areas.

2 BACKGROUND

This section introduces a small, yet representative example to which we refer to throughout the paper. We provide background knowledge on variability in space and time (to which we refer to as *variability dimensions*), and their combination. Moreover, we present the initial conceptual model capturing these dimensions.

2.1 Example Scenario: Pick and Place Unit

As example scenario, we present excerpts of the software of a *pick and place unit (PPU)* with its mechanical parts being described by Vogel-Heuser et al. [66]. The PPU consists of a crane and a stack, both mechatronic components requiring operational control. The crane moves work pieces that have been put in the stack, and uses either a micro switch or an inductive sensor. Moreover, an optical sensor may be used optionally in the stack.

2.2 Variability in Space

Variability in space supports the systematic development of families of related products, building on the principles and concepts of SPLE [5, 14, 45, 51]. In a product line, the *platform* implements features, which are typically documented in variability models

```
1  class Crane {
2    // #IF MicroSwitch && !InductiveSensor
3      MicroSwitch switch;
4      Crane(MicroSwitch ms) {...}
5    // #END
6  }
```

Listing 1: Crane.java in first revision.

```
1  class Stack {
2    // #IF OpticalSensor
3      Stack(OpticalSensor s) {...}
4    // #END
5    // #IF !OpticalSensor
6      Stack() {...}
7      // #END
8  }
```

Listing 2: Stack.java in first revision.

```
1  class Stack {
2      ...
3      Stack() {...}
4      // #END
5      // #IF OpticalSensor
6      void exchangeOpticalSensor(OpticalSensor newSens) {...}
7      // #END
8  }
```

Listing 3: Stack.java in second revision.

(e.g., feature models) [9, 16, 27, 40]. By providing a selection of the features as a feature configuration, a customized product can automatically be derived from the platform. Depending on the variability mechanism (i.e., annotative [6], compositional [13], or transformational [54]), the derivation process varies [5]. In annotative mechanisms, the implementation fragments in the platform are annotated with presence conditions, namely Boolean expressions over features, for example, in the form of preprocessor directives. In compositional mechanisms, the implementation fragments are contained in modules. Modules group fragments that have the same presence condition. A composer generates a product by merging all modules whose presence conditions are satisfied by a given configuration. Transformational (also called *delta-oriented*) mechanisms build on a core implementation and delta modules. A delta module contains a sequence of delta operations that are applied when a presence condition is satisfied. A product is derived by applying all delta operations whose presence conditions are satisfied by a given configuration in a defined order.

The feature model shown in Figure 1 represents the variability in space of the PPU example with mandatory features *Crane* and *Stack* and the optional feature *OpticalSensor*. The crane either possesses a *MicroSwitch* or an *InductiveSensor* (alternative relation). In addition, a cross-tree constraint prohibits the coexistence of the *OpticalSensor* and the *InductiveSensor*. Listing 1 and 2 capture the first implementation state of the crane and the stack, respectively. We show an annotative variability mechanism operating on preprocessor directives (`#IF`, `#END`) that encapsulate optional parts of the source code. Thus, Line 3 of Listing 2 will only exist in a product if the feature *OpticalSensor* is selected in the feature configuration, otherwise Line 6 is present; each enabled by the directive in Line 2 or 5, respectively.

2.3 Variability in Time

Variability in time considers the evolution of a system. In SCM, VCSs have been developed to manage evolution, offering storage

[1]https://doi.org/10.5281/zenodo.3975442

options and operations to handle collaborative development. While different (academic) VCSs allow to version almost arbitrary elements and relationships [15], mainstream VCSs, like SVN [50] and Git [37], organize files (as the only versioned elements) in directed, acyclic graphs. A download mechanism retrieves a local copy from the common storage (i.e., a repository) and an upload mechanism propagates local changes back to the repository. Each state of the local copy, uploaded to the repository, is referred to as *revision*, marked with a (numbered) label. In contrast to a feature configuration in SPLE, it is possible to configure a customized product by selecting a specific revision without knowing whether the revision incorporates, for example, patches or a new feature.

In our example, the files in Listing 1 and 2 are uploaded to the repository as first revision. Afterwards, another developer downloads the current state of the repository and modifies the local copy. For instance, this second revision adds an extension of the stack to exchange the optical sensor, thereby evolving the file as depicted in Listing 3: Lines 5-7 are added and uploaded to the repository as second revision. Likewise, in a third revision (not shown) the inductive sensor may be added to the crane.

2.4 Variability in Space and Time

Variability in space and in time has always been intertwined: Product lines evolve over time and VCSs can maintain different product functionalities in separated branches [30, 52, 64]. Still, the combination of both dimensions is less investigated and tends to involve inconvenient use cases. For instance, developing individual products in branches causes maintenance overhead for synchronizing branches [18, 52]. On the contrary, many existing solutions for SPLE consider variability in space only, thus requiring integration with a VCS to manage variability in time. This missing dedicated tool support for tracing feature evolution challenges developers.

Considering both dimensions systematically can help to solve this problem. Moving in this direction, Westfechtel et al. [67] have proposed the *unified version model* about 20 years ago to unify the concepts of SPLE and SCM. This model introduces a version as an abstract unification, which can either be a product variant (variability in space) or a revision (variability in time). However, the model prescribes a dedicated development process to ensure consistency, which contradicts the practical usage of almost all contemporary tools. Thus, it cannot serve as an up-to-date conceptual model.

In our example, the implementation corresponding to the feature *Stack* exists in two revisions: The first one, in which only the constructor exists (cf. Listing 2), and the second one incorporating an exchange mechanism in Listing 3. Consequently, the second revision is also a feature revision.

2.5 Conceptual Model

Recently, we introduced an initial conceptual model [2, 3] documenting elements of each variability dimension and their relations. As we build on this model, we briefly summarize its key ideas. The initial conceptual model, shown in Figure 2, distinguishes the `Revision Space` from the `Variant Space` and the `System Space`, and categorizes its elements with respect to the variability dimension they belong to. The `Revision Space` represents concepts of variability in time only, that is `Versioned Systems` composed of

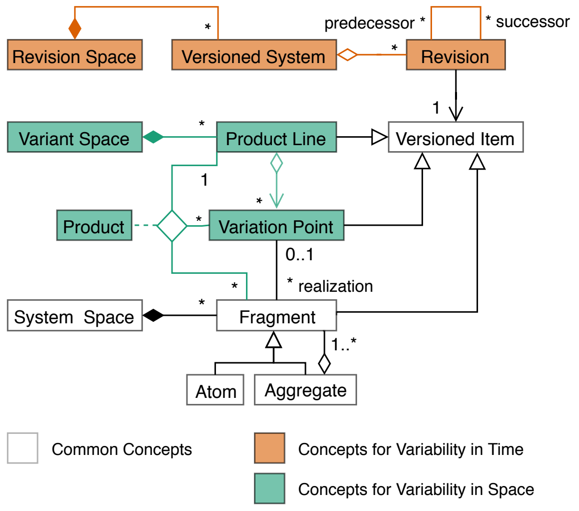

Figure 2: The initial conceptual model for unifying concepts of variability in space (green) and in time (orange) with common concepts (white) [3].

`Revision` graphs. In contrast, the `Variant Space`, which incorporates `Product Lines`, belongs to variability in space. `Products` can be derived from a `Product Line` by selecting `Variation Points` that are implemented by arbitrary `Fragments`, which are, for instance, lines of code or model elements. A `Fragment` represents one of the concepts common to both, variability in space and time. Moreover, a `Versioned Item` puts the concepts under version control.

While the conceptual model documents concepts and their relations present in each variability dimension, it misses a unification of these concepts and is not designed to represent concepts of contemporary variability tools. In contrast, we now contribute a systematically elicited conceptual model that not only unifies concepts, but also allows to classify contemporary tools.

3 STATE OF THE ART

Conceptual Models for Variability in Space. The SPLE community has designed multiple processes and conceptual models to define the terminology used to specify variability in space [5, 43, 51]. Despite these efforts, even within the SPLE community varying terminologies have evolved, for example, resulting in the synonymous use of `product` and `variant`. A particular technique of SPLE to unify terminology and provide a common conceptual model or ontology for a domain is variability modeling, and particularly the de-facto standard feature modeling [16, 17, 26, 40, 55]. However, while this technique exists, the terminology of variability in space has never been unified, and the conceptual model we describe tackles this problem with an even broader perspective. Particular limitations of existing processes and models are their missing capabilities to describe systems that allow for variability in space and time, and their limited independence of implementation specifics.

Conceptual Models for Variability in Time. Similarly to SPLE, conceptual models and taxonomies for SCM have been proposed [15, 38, 50, 53]. The most prominent concept to specify and capture variability in time is arguably the version model, describing how the versions in a SCM system are managed. However, as Conradi and

Westfechtel [15] show, each SCM system employs its own version model with varying terminology and conceptual differences. While a mapping between the concepts and terms of different SCM systems exists, we are not aware of an actual conceptual model providing a unified terminology to specify variability in space and time.

Related Surveys of Variability in Space and Time. The closest research to the conceptual model is the work of Conradi and Westfechtel [15] who extend the version models identified towards capturing the relation of variability in space and time. Building on this idea, Westfechtel et al. [67] introduce the uniform version model, which provides a common model for basic SCM and SPLE concepts. However, the model is intertwined with implementation details, relying on propositional logic and selective deltas to manage variability. Schwägerl [57] builds upon the uniform version model, replacing some of the concepts and partly resembling an own conceptual model. In contrast to our work, the goal was to develop a specific tool (i.e., SuperMod), which we analyzed to derive a general conceptual model for capturing variability in space and time; independent of concrete implementation details of a certain tool. Similarly to our work, Linsbauer et al. [33] survey variation control systems. Variation control systems aim to combine variability in space and time, and thus we analyzed a set of tools that have been published more recently, and for which we could interview experts (e.g., SuperMod, ECCO). Other researchers compared tools for SPLE or SCM [8, 22, 46, 49, 53]. In contrast to the conceptual model, these works focus on classifying and comparing the identified tools instead of unifying their concepts and relations. They do not perform a unification to derive a unified conceptual model for variability in space and time.

4 CONTEMPORARY VARIABILITY TOOLS

For constructing the conceptual model, we select a representative set of tools that i) handle either variability in space, time, or both, ii) implement concepts of problem *and* solution space [25] (e.g., in contrast to pure variability modeling languages and tools which cover only the problem space [7, 10, 23, 56]), and iii) were available to us. In this section, we briefly introduce each tool.

4.1 Tools for Variability in Space

As mentioned in Section 2, annotative, compositional, and transformational variability mechanisms exist in SPLE. We selected three tools, each covering at least one mechanism.

FeatureIDE [32, 39] originates from academia and is a tool platform supporting the development of product lines based on the Eclipse platform. The tool includes, for instance, extensive feature modeling support, implementing, configuring, and testing. FeatureIDE provides annotative and compositional variability mechanisms, and thus serves to cover both in our analysis.

pure::variants [12] is an industrial SPLE tool with similar functionalities as FeatureIDE. pure::variants builds also on the Eclipse platform and covers different variability mechanisms, but focuses on the annotative mechanism in the form of preprocessor directives. We consider the *pure::variants evaluation edition*, which is why we may not have obtained all insights. However, the main advantage of pure::variants is its design to suite practitioners from industry, helping us to incorporate this perspective in our model.

SiPL [47, 48] implements product lines based on a transformational variability mechanism. SiPL uses feature models and delta modules to implement variability, differing from the previous tools. Compared to other delta-oriented SPLE tools, a unique characteristic of SiPL is that the notion of a delta is refined in an edit script [28] generated by comparing models through which a modeler expresses variability in space.

4.2 Tools for Variability in Time

As representatives of variability in time, we selected two well-established and widely used VCSs, covering a centralized and a decentralized system.

Subversion (SVN) [50] is a centralized VCS, meaning that one central repository is stored on a server. SVN allows users to check-out one state of this repository into a local workspace, implement changes, and *commit* them directly to the central repository. Each commit results in a new revision, which is numbered sequentially. Thus, developers may *check out* a specific revision into their local workspace. SVN supports branching of the central repository as well as merging of a branch with the main branch.

In contrast to SVN, **Git** [37] supports decentralized versioning, meaning that every user has its own copy of the entire repository. This leads to a distributed network of repositories. As such, Git supports local operations (such as a *commit* of changes to the local repository) as well as distributed operations (such as the *clone* operation that creates a local copy of the entire remote repository, or the *push* and *pull* operations that are used to synchronize between repository clones). Since Git and SVN apply different concepts and are widely used in the software engineering community, they are ideal representatives for managing variability in time.

4.3 Tools for Variability in Space and Time

This section presents five contemporary tools covering variability in space *and* time.

ECCO [20, 21, 34, 35] was initially designed for re-engineering cloned system variants into a product line, thereby computing mappings between features and fragments of implementation artifacts. The tool evolved to support feature revisions based on the common checkout/commit workflow for distributed software development. Upon commit, ECCO assigns presence conditions consisting of feature revisions to the corresponding artifact fragments, and thus combines concepts for variability in space and time.

SuperMod [58, 59] is based on the unified version model [67], consequently unifying temporal revisions and spatial variants as *versions*. The development of a product line takes place product-wise, meaning that the product space (workspace) is populated with the feature model and the model artifacts belonging to one revision and feature configuration. The version space comprises an internal repository holding the superimposition of all product-line elements annotated with logical expressions over features and revisions. Similar to Git, SuperMod builds on the checkout/commit workflow locally, and allows multi-user development by pushing/pulling the local state to one remote repository server.

DeltaEcore [61, 62] supports, like SiPL, a transformational variability mechanism. The tool automatically derives delta languages, which are used to express the delta operations to the common

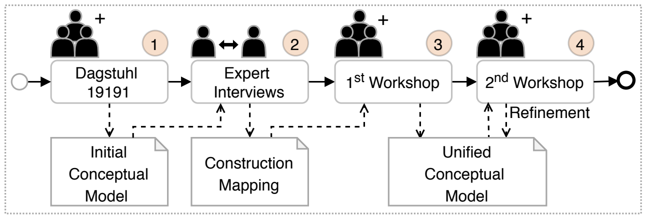

Figure 3: Construction process of the unified conceptual model.

core of the product line. Developers specify these delta operations to define how to transform a system from one state into another, building on the delta language that can parse the programming language of the system. Products are configured based on a *hyper feature model* [60], which provides the option to specify revisions of individual features in contrast to revisions of the whole system (which are not explicitly supported).

DarwinSPL [41] copes with variability in space and time based on DeltaEcore, adapting and extending its concepts. For instance, the hyper feature model in DeltaEcore allows for temporal variability of features, but the feature model itself cannot evolve. DarwinSPL allows to evolve the feature model, but in the process removes the ability to evolve individual features.

VaVe [4] integrates the management of *VA*riants (space) and *VE*rsions (time). It builds on Vitruvius [29], a view-based framework that preserves consistency by propagating changes made in a projection of a system to all affected parts of the system. In contrast to the aforementioned tools, each change in one particular product potentially initiates internal consistency preserving mechanisms to maintain consistency of the whole product line.

Note that some of the selected tools can be used in combination. For example, an SPLE tool may integrate a VCS for supporting the evolution of the product line. We did not consider these combinations in the construction process of the reference model, as they are covered implicitly by considering each individual tool and do not contribute new concepts for variability in space and time—which the tools supporting both dimensions do.

5 CONSTRUCTION PROCESS

Figure 3 shows the construction process of the conceptual model. It was inspired by the work of Ahlemann and Riempp [1] who propose iterative steps, such as expert interviews and refining the model until domain experts reach consensus. As mentioned in Section 2.5, the initial conceptual model (originating from Dagstuhl Seminar 19191 [2]) aimed to document concepts for variability in space and in time, and their relations, but not to unify them (cf. step ①). The initial conceptual model served as input to expert interviews, which we describe in the following.

5.1 Expert Interviews

We (specifically, the first author of this paper) conducted semi-structured interviews with one tool expert per tool (cf. step ②). The goal was to understand to what extent the conceptual model captures concepts of contemporary tools and which adaptations are needed. Note that we did not conduct interviews on Git and SVN because they are widely used and extensive documentation is available. One week before each interview, we provided the blank interview guide to each tool expert and completed the guide jointly during the interview. Subsequently, we conducted a follow-up inspection of the documented answers to ensure they were complete and consistent. The eight interviews took 83 minutes on average.

The interview guide involves four parts. The first part introduces the initial conceptual model and definitions of the involved concepts. The second part asks for a mapping of concepts of the conceptual model onto constructs of the expert's tool (to create a *construction mapping*) based on the following questions:

- What are the main constructs of the tool?
- For every concept in the conceptual model, what are the semantically equivalent tool constructs?
- Is there a tool construct not represented by any concept of the conceptual model?

The third part comprises main use cases of each tool and its scope in order to distinguish it from other tools. Finally, the fourth part encompasses tool operations (e.g., code analysis) to obtain a holistic understanding of each tool.

From the interviews, we identified tool constructs that did not map to any concepts in the conceptual model. These constructs were `Feature` and `Constraint`. Moreover, we observed that some tools (i.e., DeltaEcore, DarwinSPL, ECCO, SuperMod, VaVe) do not distinguish the concepts of `Versioned System` and `Product Line` and, instead, represent both by a single construct (i.e., `Product Line` in DeltaEcore and DarwinSPL, `Repository` in ECCO and SuperMod, `System` in VaVe). Finally, we found that many tools involve a construct for the `Mapping` between `Fragments` and `Features` as well as for the `Configuration`. In the initial conceptual model, however, these constructs are addressed only implicitly: the `realization` relation between `Variation Point` and `Fragment` represents the `Mapping`, whereas the ternary association between `Product`, `Product Line`, and `Variation Point` aligns with the `Configuration`.

5.2 Workshops

We conducted two workshops (cf. steps ③ and ④) including up to 15 participants, which were tool experts of the interviews, authors of this paper, and more researchers of the SPLE and SCM communities. The members of the workshops discussed the results of the interviews (specifically, the construction mapping) and adapted the initial conceptual model to gradually evolve it into the model we present in this paper. Major changes involve the unification of concepts that we found to be represented by a single construct in tools (e.g., `Product Line` and `Versioned System`), adding concepts or making them explicit (e.g., `Constraint`, `Mapping`), unifying concepts that handle both variability in space and time (e.g., `Configuration`), and introducing new hybrid concepts that do not exist in tools focusing either on variability in space or in time.

6 THE CONCEPTUAL MODEL

In this section, we present the conceptual model shown in Figure 4. We highlight concepts for variability in space green, concepts for variability in time orange, concepts for variability in both dimensions purple, and unified concepts white. We use lighter colors for abstract concepts. Moreover, the model is split in two parts. The right side shows the problem space in SPLE, namely the abstraction of the domain, which is equivalent to the version space in SCM. The left side shows the solution space in SPLE, namely the actual implementation, which is equivalent to the product space in SCM [15].

6.1 Concepts for Variability in Space

The conceptual model represents variability in space using three concepts: `Feature Option`, `Feature`, and `Constraint`. A `Feature Option` represents the abstract possibility to customize a system in terms of its concrete `Features`, which can be enabled (and potentially assigned a value) or disabled. Considering our example, the *Crane* and the *Stack* of the PPU represent concrete `Features`, and thus a `Feature Option`. Another established concept for variability in space is the `Constraint`. A `Constraint` can be any arbitrary expression (e.g., a propositional formula) defined over `Feature Options` to constrain which combinations of `Feature Options` are valid (e.g., exclude, alternative). In the PPU, the cross-tree constraint *OpticalSensor && !InductiveSensor* of the feature model exemplifies one kind of `Constraint`.

6.2 Concepts for Variability in Time

The conceptual model covers variability in time using two concepts: `Revision` and `System Revision`. A `Revision` describes evolution over time and relates to its predecessor and successor `Revisions`. The structure of multiple directly succeeding and preceding `Revisions` represents branches and merges. A `Revision` is an abstract concept and can be a concrete `System Revision`, representing the state of the whole system at a particular point in time. For the PPU, a new `System Revision` involves modifications in the feature *Stack* (cf. Listing 3).

6.3 Concepts for Variability in Space and Time

Concepts for variability in space and time are hybrid concepts not present in tools focusing solely on variability in space or time. In the conceptual model, the `Feature Revision` represents variability in space and time as a concrete combination of `Feature Option` and `Revision`. It represents the state of one particular `Feature` at one point in time. In the PPU, the modification of the *Stack* implementation (cf. Listing 3) can also be considered a `Feature Revision` of the *Stack* instead of a `System Revision` of the entire PPU. Whether a modification is considered a `System Revision` or a `Feature Revision` is a matter of the granularity at which modifications in the solution space are mapped to the problem space. In contrast to a `Feature Revision`, which is local to a `Feature`, a `System Revision` also determines which `Feature Options` and `Constraints` are enabled. For example, the `System Revision` of the PPU determines which `Features`, `Feature Revisions`, and `Constraints` exist in this `System Revision`.

6.4 Unified Concepts

While concepts for variability in space and time involve both dimensions jointly, unified concepts represent either variability in space, in time, or both. The main concept in the conceptual model is the `Unified System`, containing most other concepts and essentially describing the developed system. An `Option` is a high-level abstraction of any variation in space, time, or both of a system in the problem space. Therefore, we use a generic name not associated with SPLE or SCM terminology (as is `Variation Point`, which is associated with SPLE). An `Option` manifests as `Feature` (variability in space), `System Revision` (variability in time), or `Feature Revision` (both). Similarly, a `Fragment` is the core concept to describe the implementation of a system. Depending on the granularity and system, a `Fragment` may be an entire file, a single element, or a line of text (e.g., in source code, documentation, models, delta modules, or meta-information). We specify neither the level of granularity nor the purpose of `Fragments` to keep the conceptual model as generic as possible. In the PPU, every line in a Java file (e.g., in Listing 2) or the file itself may represent `Fragments` depending on the implementation of the respective tool. A `Mapping` connects an expression (e.g., a propositional formula) over `Options` with `Fragments`. Thus, it establishes the connection between the solution space (`Fragments`) and the problem space (`Options`). It is possible that expressions do not contain `Options`, but only Boolean constants to govern the presence or absence of `Fragments` (e.g., mandatory or deprecated code). In our example, Line 3 in Listing 2 implies a `Mapping` of a `Fragment` (i.e., the line of code) to `Options`, namely *Stack && OpticalSensor* (the stack is not explicitly mentioned in the directive as it must be present as mandatory feature).

The concept `Configuration` exists in different forms in both communities, SCM and SPLE (cf. Section 1). To align both communities, we unify its meaning: a `Configuration` is a selection of `Options` (in space and/or time) used to derive a specific `Product`. In the PPU, a `Configuration` may define to use *Crane* and *OpticalSensor* in the first revision, and *Stack* in the second revision. In contrast to the aforementioned concepts, the `Product` does not directly impact the `Unified System`, but is included for completeness. Based on a configuration, a `Product` is derived by tool-specific mechanisms (e.g., delta modules) that are part of the tool's behavior and specify which and how `Fragments` are composed.

6.5 Application in Practice

Extension and Refinement of Concepts. When applying the model in practice, each of the presented concepts can be refined by means of subclassing. For example, the `Constraint` can be refined into subclasses, such as mandatory child or optional child, for tools that use feature models to model feature constraints. Another example is the `Fragment`, which could be refined into the subclasses `Core Model` and `Delta Module` for tools that use deltas.

Well-Formedness. In addition to the graphical representation, we include constraints for the well-formedness of the conceptual model to avoid undesirable effects. For example, `Feature Revisions` in one revision graph (that is induced by predecessor and successor revisions) may belong to different `Features`, or a revision graph may be a mix of `System Revisions` and `Feature Revisions`—which is not desired. The following invariants (which may be implemented as OCL constraints [44]) avoid this effect: 1) a revision graph can only contain revisions of the same type (i.e., either `System Revisions` or `Feature Revisions`), 2) all revisions of the same system must be contained in the same container (i.e., the `Unified System` for `System Revisions` or a `Feature` for `Feature Revisions`), and 3) a revision graph must be a directed, acyclic

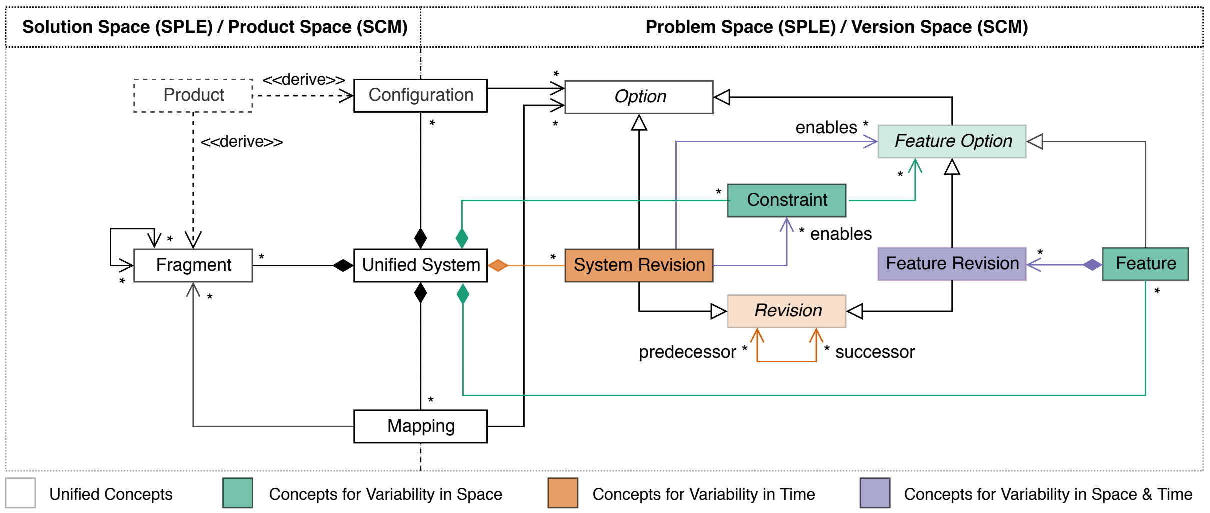
Solution Space (SPLE) / Product Space (SCM)
Problem Space (SPLE) / Version Space (SCM)
Product
<<derive>>
Configuration
Option
enables
Feature Option
Constraint
enables
Fragment
Unified System
System Revision
Feature Revision
Feature
Revision
predecessor
successor
Mapping
Unified Concepts
Concepts for Variability in Space
Concepts for Variability in Time
Concepts for Variability in Space & Time

Figure 4: UML class diagram of the conceptual model for unifying concepts for variability in space and time.

graph (DAG). We refrain from adding further invariants, such as the conformance of a `Configuration` with `Constraints`, since some tools may allow invalid configurations.

7 VALIDATION

This section describes the validation of the unified conceptual model. The validation comprises a qualitative analysis based on a questionnaire, and a quantitative analysis based on metrics. Our analysis methods allow us to answer research questions that we derive from the following research goals.

Goals. The goal of the conceptual model is to cover and unify concepts and their relations that cope with variability in space, time, and both, based on the selected tools. Therefore, we consider the following two properties of the conceptual model:

- *Granularity*: The granularity of the concepts in the conceptual model should be appropriate, that is not unnecessarily fine-grained, but also not too coarse-grained.
- *Coverage*: The conceptual model should cover all concepts needed to describe the selected tools coping with variability in space, time, and both, yet no more than that.

Questions. We answer the following research questions:

RQ_1 Is the conceptual model of appropriate granularity? That is, are its concepts too fine-grained or too coarse grained?

RQ_2 Is the conceptual model of appropriate coverage? That is, are there any unused or missing concepts?

Answering these two research questions allows us to reason about the granularity and coverage of the conceptual model.

Steps. Figure 5 shows the concrete steps of the validation process (following the construction process) that we discuss in more detail in the following.

7.1 Qualitative Analysis

We performed a qualitative analysis based on questionnaires completed by tool experts (cf. step ⑤) to map constructs and relations of their tools to the concepts and relations of the conceptual model. Moreover, we asked for missing concepts and general remarks.

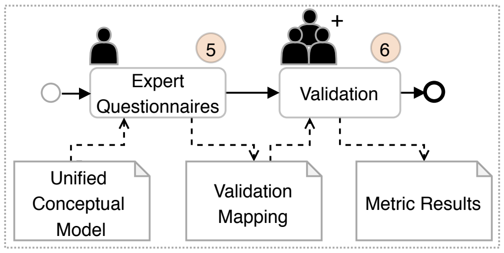

Figure 5: Validation process of the unified conceptual model.

Expert Questionnaire. Since after the interviews all tool experts were familiar with the mapping procedure, we refrained from explicit interviews. Instead, we provided questionnaires for mapping tools to the conceptual model. The questionnaire comprises two parts and is structured similarly to the interview guide. The first part introduces the unified conceptual model and definitions of the involved concepts and relations. The second part asks whether each concept and relation of the conceptual model maps to constructs and relations of the respective tool, also taking into account unmapped constructs and names of each tool construct.

Validation Mapping. In the following, we present the mapping of concepts and relations of the conceptual model to constructs and relations of each tool. Note that these mappings are performed on the conceptual level of the tools (considering their semantics and expressiveness), and not on the implementation level. Moreover, we do not consider the abstract concepts (`Option` and `Revision`), as they cannot be instantiated.

In Table 1, we show the mapping of concepts of the conceptual model to respective tool constructs. We can see that all tools incorporate constructs for five concepts: `Fragment`, `Product`, `Unified System`, `Mapping`, and `Configuration`. However, these constructs differ considerably between the tools. For example, a `Fragment` in Git is a `Blob` (file content) or a `Tree Object` (directory). In SVN, these constructs are called `File Node` and `Directory Node`, respectively. FeatureIDE and pure::variants manage `Fragments` as so-called `Assets` that are processed by an external composer. ECCO and SuperMod use the similarly generic terms `Artifact`

Table 1: *Validation Mapping:* Results of mapping the constructs of each tool to the concepts in the conceptual model.

Concept \ Tool	FeatureIDE	pure::variants	SiPL	SVN	Git	ECCO	SuperMod	DeltaEcore	DarwinSPL	VaVe
Fragment	Asset	Asset	Core Model, Delta Module	File Node, Directory Node	Blob, Tree Object	Artifact	Product Element	Core Model, Delta Module	Core Model, Delta Module	Core Model, Delta Module
Product	Product	Variant	Product	Working Copy	Working Copy	Variant	Product	Product	Product	Product
Unified System	Product Line	Product Line	Product Line	Repository	Repository	Repository	Repository	Product Line	Product Line	System
Mapping	Mapping[1]	Restriction	Application Condition	Tree Object	Tree Node	Association	Mapping[1]	Mapping Model	Mapping Model	Mapping[1]
Feature	Feature	Feature	Feature	–	–	Feature	Feature	Feature	Feature	Variant
System Revision	–	–	–	Commit	Revision	–	Revision	–	Temporal Validity	–
Feature Revision	–	–	–	–	–	Revision	–	Version	–	Version
Configuration	Variant, Configuration	Configuration	Configuration	Commit	Revision	Configuration	Choice	Configuration	Configuration	Configuration
Constraint	Constraint	Constraint, Relation	Constraint	–	–	–	Dependency	Constraint	Constraint	Constraint

[1]The `Mapping` is part of the conceptual level without an explicit construct on implementation level.

and `Product Element`, respectively. All delta-oriented tools use the same types of `Fragments`: `Core Model` and `Delta Module`.

Moreover, the mapping shows that concepts we introduced particularly for variability in space or time align with the corresponding tools: Git and SVN manage only variability in time using `System Revisions`. Similarly, FeatureIDE, pure::variants, and SiPL manage only variability in space using the concepts of `Features` and `Constraints`. The remaining tools involve the concepts `Features` and `Constraints` in addition to `System Revision` or `Feature Revision` to incorporate variability in time. Interestingly, none of the tools covering both variability dimensions considers `System Revision` and `Feature Revision` at the same time.

In every tool, a `Mapping` connects `Fragments` and `Options` (i.e., `Features`, `Feature Revisions`, and `System Revisions`). In tools that cover only variability in time (i.e., Git and SVN), the mapping is rather trivial as it only maps a `System Revision` to a number of `Fragments` in a tree structure. Tools that (additionally) consider variability in space require more complex `Mappings` since they need to deal with `Features`.

In Table 2, we show the mapping of relations of the conceptual model to respective relations in the tools. While all tool constructs map to a concept in the conceptual model, there are relations in some tools that are not represented by the conceptual model. More specifically, Git and ECCO include a relation called `Remote`, allowing a repository (i.e., `Unified System`) to refer to other repositories. This relation exists due to the distributed nature of these tools. It is not connected to the dimensions of space and time, which is why we did not incorporate it in the conceptual model for now.

7.2 Quantitative Analysis

We performed a quantitative analysis using metrics (cf. step ⑥) to quantify the fit of the conceptual model to the selected tools based on the validation mapping.

Metrics. We adapted the *framework for language evaluation* proposed by Guizzardi et al. [24] that introduces the properties *laconic*, *lucid*, *complete*, and *sound*. We extend these properties to metrics measuring the degree to which these properties hold for a given model and tool. The metrics *laconicity* and *lucidity* assess the granularity of concepts of the conceptual model (RQ$_1$), whereas *completeness* and *soundness* assess its coverage of concepts (RQ$_2$).

We define each metric for a conceptual model M and a tool T contained in the set of selected tools $\mathcal{T}$. The model M is a set of *model concepts* $m \in M$. A tool $T \in \mathcal{T}$ is a set of *tool constructs* $t \in T$. For simplicity, we consider relations as concepts respectively constructs, too. $\mathbb{R}_T^M \subseteq M \times T$ denotes the set of *mappings* of concepts in M onto constructs in T, displayed in the Tables 1 and 2.

A tool's construct t is *laconic*, iff it implements at most one concept m of the conceptual model M. *Laconicity* is then the fraction of *laconic* tool constructs. Low laconicity indicates that concepts of the conceptual model may be too fine-grained.

$$\text{laconic}(M,T,t) = \begin{cases} 1 & \textbf{if } |\{m \mid (m,t) \in \mathbb{R}_T^M\}| \leq 1 \\ 0 & \textbf{otherwise} \end{cases}$$
$$\text{laconicity}(M,T) = \frac{\sum_{t \in T} \text{laconic}(M,T,t)}{|T|}$$

A model's concept m is *lucid*, iff it is implemented by at most one construct t of a tool T. *Lucidity* is then the fraction of *lucid* model concepts. Low lucidity indicates that concepts of the conceptual model may be too coarse-grained.

$$\text{lucid}(M,T,m) = \begin{cases} 1 & \textbf{if } |\{t \mid (m,t) \in \mathbb{R}_T^M\}| \leq 1 \\ 0 & \textbf{otherwise} \end{cases}$$
$$\text{lucidity}(M,T) = \frac{\sum_{m \in M} \text{lucid}(M,T,m)}{|M|}$$

A tool's construct t is *complete*, iff it is represented by at least one concept m in the conceptual model M. *Completeness* is then the fraction of *complete* tool constructs. Low completeness indicates that the conceptual model might be missing concepts.

$$\text{complete}(M,T,t) = \begin{cases} 1 & \textbf{if } |\{m \mid (m,t) \in \mathbb{R}_T^M\}| \geq 1 \\ 0 & \textbf{otherwise} \end{cases}$$
$$\text{completeness}(M,T) = \frac{\sum_{t \in T} \text{complete}(M,T,t)}{|T|}$$

A model's concept m is *sound*, iff it is implemented by at least one construct t in the tool T. *Soundness* is then the fraction of *sound*

Table 2: ***Validation Mapping:*** **Results of mapping the relations in each tool to the relations in the conceptual model.**

Relation \ Tool	Feature-IDE	pure::variants	SiPL	SVN	Git	ECCO	SuperMod	DeltaEcore	Darwin-SPL	VaVe
Fragment has * Fragment	●	●	●	●	●	●	●	●	●	●
Mapping has * Fragment	●	●	●	●	●	●	●	●	●	●
Configuration has * Option	●	●	◒	◒	◒	●	◒	●	●	●
Unified System has * Fragment	●	●	●	●	●	●	●	◒	●	●
Unified System has * Mapping	●	●	●	●	●	●	●	●	●	●
Unified System has * Constraint	●	●	●	–	–	–	●	●	●	●
Unified System has * Feature	◒	●	●	–	–	●	●	●	●	●
Unified System has * System Revision	–	–	–	●	●	–	●	–	●	–
Unified System has * Configuration	●	●	–	●	●	–	–	●	●	–
Mapping has * Option	●	●	●	◒	◒	●	●	●	●	●
Feature has * Feature Revision	–	–	–	–	–	●	–	●	–	●
Constraint has * Feature Option	◒	●	●	–	–	–	●	●	●	●
System Revision enables * Feature Option	–	–	–	–	–	–	●	–	●	–
System Revision enables * Constraint	–	–	–	–	–	–	●	–	●	–
Revision has * Successor and * Predecessor	–	–	●	●	●	–	●	●	●	●
Unmapped	–	–	–	–	Repository refers to * Repository	Repository refers to * Repository	–	–	–	–

● The relations are identical. ◒ The cardinality of the relation in the conceptual model is less restrictive than the cardinality of the relation in the tool.

model concepts. Low soundness indicates that the conceptual model may include unused concepts.

$$\text{sound}(M,T,m) = \begin{cases} 1 & \textbf{if } |\{t \mid (m,t) \in \mathbb{R}_T^M\}| \geq 1 \\ 0 & \textbf{otherwise} \end{cases}$$

$$\text{soundness}(M,T) = \frac{\sum_{m \in M} \text{sound}(M,T,m)}{|M|}$$

Finally, we generalize each metric from a single tool T to a finite set of tools $\mathcal{T}$ to get a holistic measure over all tools, reflecting the goal of unification. For laconicity and completeness, this generalization is straightforward since these metrics are based on the assessment of the properties laconic and complete with respect to the conceptual model. For lucidity and soundness, we define how to assess the properties lucid and sound with respect to a set of tools as follows.[2] A model concept m is *lucid*, if it is *lucid* in all tools $T \in \mathcal{T}$. A model concept m is *sound*, if it is *sound* in at least one tool $T \in \mathcal{T}$.

$$\overline{\text{laconicity}}(M,\mathcal{T}) = \text{laconicity}\Big(M, \bigcup\nolimits_{T \in \mathcal{T}} T\Big)$$

$$\overline{\text{lucidity}}(M,\mathcal{T}) = \frac{\sum_{m \in M} \big(\min_{T \in \mathcal{T}} \text{lucid}(M,T,m)\big)}{|M|}$$

$$\overline{\text{completeness}}(M,\mathcal{T}) = \text{completeness}\Big(M, \bigcup\nolimits_{T \in \mathcal{T}} T\Big)$$

$$\overline{\text{soundness}}(M,\mathcal{T}) = \frac{\sum_{m \in M} \big(\max_{T \in \mathcal{T}} \text{sound}(M,T,m)\big)}{|M|}$$

Results. In Table 3, we display the values of the four metrics (columns) per tool (rows). Each row contains the percentage as well as the absolute number of conceptual model concepts and relations (in case of lucidity and soundness) or tool constructs and relations (in case of laconicity and completeness) that satisfy the condition for each metric. For all investigated tools, most metric results for laconicity, lucidity and completeness are close to 100 %. For instance, the conceptual model is 96 % lucid with respect to Git, because the concept `Fragment` does not satisfy the condition for

[2]We do not treat constructs that map to the same concept as equivalent (i.e., constructs mapping to the same concept do not form an equivalence class). Tool constructs are therefore unique (i.e., for all $S, T \in \mathcal{T}$ with $S \neq T$ it holds that $S \cap T = \emptyset$).

Table 3: ***Metric Results*** **for each tool individually.**

	laconicity	lucidity	completeness	soundness
FeatureIDE	100% (17/17)	100% (24/24)	100% (17/17)	71% (17/24)
pure::variants	100% (17/17)	100% (24/24)	100% (17/17)	71% (17/24)
SiPL	100% (16/16)	96% (23/24)	100% (16/16)	71% (17/24)
SVN	93% (14/15)	96% (23/24)	100% (15/15)	63% (15/24)
Git	94% (15/16)	96% (23/24)	94% (15/16)	63% (15/24)
ECCO	100% (16/16)	100% (24/24)	94% (15/16)	67% (16/24)
SuperMod	100% (21/21)	100% (24/24)	100% (21/21)	92% (22/24)
DeltaEcore	100% (20/20)	96% (23/24)	100% (20/20)	79% (19/24)
DarwinSPL	100% (22/22)	96% (23/24)	100% (22/22)	92% (22/24)
VaVe	100% (19/19)	96% (23/24)	100% (19/19)	79% (19/24)

lucidity since it is represented by the two constructs `Blob` and `Tree Object`. The soundness values are generally lower because no tool implements *all* concepts and relations.

In Table 4, we aggregate the results for all tools. We show the four metrics (columns) for concepts/constructs and relations as well as the different dimensions (rows). The conceptual model is not laconic, due to two constructs: `Commit` in Git and `Revision` in SVN each represent both, the `System Revision` and the `Configuration`. While the mapping of `System Revision` to `Commit`/`Revision` is straightforward, the mapping to `Configuration` is debatable since a configuration in Git and SVN does not explicitly exist (as these tools have no constructs for variability in space). Therefore, a configuration would trivially only consist of a single `Commit`/`Revision`. Considering completeness, the two aforementioned relations (self-relating repositories in Git and ECCO) are not mapped. In contrast to the soundness values of individual tools, the conceptual model is entirely sound due to the aggregation, because every concept of the model is implemented by at least one construct in at least one tool.

7.3 Discussion

In the following, we discuss the validation results to answer our research questions.

RQ$_1$: Is the conceptual model of appropriate granularity? We answer this question based on laconicity and lucidity. The laconicity values indicate that the concepts `System Revision` and `Configuration` are unnecessarily fine-grained with respect to the tools Git and SVN (both deal only with variability in time), because a

Table 4: *Metric Results* over all tools.

Kind	for	$\overline{\text{laconicity}}$	$\overline{\text{lucidity}}$	$\overline{\text{completeness}}$	$\overline{\text{soundness}}$
Concepts/ Constructs	*Space*	100% (15/15)	100% (2/2)	100% (15/15)	100% (2/2)
	Time	100% (4/4)	100% (1/1)	100% (4/4)	100% (1/1)
	Both	100% (3/3)	100% (1/1)	100% (3/3)	100% (1/1)
	Unified	96% (48/50)	80% (4/5)	100% (50/50)	100% (5/5)
	Total	97% (70/72)	89% (8/9)	100% (72/72)	100% (9/9)
Relations	*Space*	100% (22/22)	100% (3/3)	100% (22/22)	100% (3/3)
	Time	100% (10/10)	100% (2/2)	100% (10/10)	100% (2/2)
	Both	100% (7/7)	100% (3/3)	100% (7/7)	100% (3/3)
	Unified	100% (68/68)	100% (7/7)	97% (66/68)	100% (7/7)
	Total	100% (107/107)	100% (15/15)	98% (105/107)	100% (15/15)
All	*Space*	100% (37/37)	100% (5/5)	100% (37/37)	100% (5/5)
	Time	100% (14/14)	100% (3/3)	100% (14/14)	100% (3/3)
	Both	100% (10/10)	100% (4/4)	100% (10/10)	100% (4/4)
	Unified	98% (116/118)	92% (11/12)	98% (116/118)	100% (12/12)
	Total	99% (177/179)	96% (23/24)	99% (177/179)	100% (24/24)

`System Revision` is synonymous to a `Configuration`. Still, merging both concepts is not desirable for any tool that deals with variability in space, since a `Configuration` is no longer a single `System Revision` but rather a set of `Features`. The lucidity values indicate that the concept `Fragment` is too coarse-grained with respect to six tools and could be split up. Taking a closer look, the low value results from different levels of abstraction used in the tools. For example, ECCO and SuperMod align well with their abstract representation of `Fragments`. In contrast, other tools interpret `Fragments` more specifically, such as delta-oriented tools (e.g., DeltaEcore, SiPL), where the tool experts consider a `Fragment` to be represented by a `Core Model` and `Delta Modules`. These cases result in lower lucidity. However, the reduction in lucidity is desired as we aimed to avoid that the conceptual model becomes too tool-specific, and thus limited to specific techniques, which would cause lower laconicity.

In summary, the results show that the conceptual model is of appropriate granularity. No concepts should be merged (i.e., generalized). Also, no concepts could be split up (i.e., made more specific) without becoming too tool-specific.

RQ$_2$: Is the conceptual model of appropriate coverage?
We answer this question based on completeness and soundness. The completeness values indicate that the `Remote` relation of two of the tools is missing in the model and may be added (i.e., `Repository refers to * Repository`). The soundness values *per* tool are rather low. This is due to the fact that the conceptual model aims to cover concepts and relations of *all* tools coping with variability in space, time, and both. Consequently, the conceptual model shows lower soundness with respect to tools implementing only one of these dimensions. This fact propagates to the aggregated values in Table 4, confirming that every concept in the conceptual model is needed in at least one tool, and thus there are actually no unused concepts/relations in the conceptual model.

Altogether, the results show that the conceptual model achieves high coverage. There is no unused concept or relation. Moreover, no concepts are missing. Only relations related to distributed development are not (yet) represented in the conceptual model.

8 THREATS TO VALIDITY

One threat to the construct validity is the level of abstraction at which tool constructs are mapped to model concepts. We performed this mapping on the conceptual level, not on the implementation level. However, it is not always obvious which tool constructs constitute the conceptual level. For example, FeatureIDE implements the concept `Constraint` with multiple constructs that are quite specific to feature models, such as mandatory child or alternative group. In such cases, we chose the overarching parent constructs (in this example, the `Constraint`) as a representative and did not consider the more specific constructs. Interestingly, this was also the level of abstraction on which tool experts tended to answer the questionnaires. Generally, we took the answers in the questionnaires as literally as possible with a minimum amount of interpretation and adjustment of the level of abstraction.

A threat to construct and external validity is whether the selected tools are representative for both, the SPLE and SCM domain. We argue that our tool selection covers a representative body of existing tools of both domains. Furthermore, the tools are diverse: Every variability dimension (and combinations) is represented by at least two tools (i.e., tools only for variability in space, variability in time, both with `System Revisions`, and both with `Feature Revisions`). This way, we mitigated bias and local optimizations towards particular tools.

One potential threat to the internal validity is that some tool experts are authors of this paper, which could introduce bias towards their tools. However, involving experts is a recommendation for building conceptual models [1]. We aimed to mitigate this threat by involving further external researchers into the discussions on the model construction.

Finally, the answers of tool experts in the questionnaire were occasionally vague, incomplete, or posing questions. This threatens the conclusion validity. We carefully analyzed the answers and conferred with tool experts to improve the conclusion validity.

9 CONCLUSION

In this paper, we presented a conceptual model for unifying concepts of variability in space and time. We constructed the conceptual model based on an iterative, expert-driven analysis of representative tools. We showed that the conceptual model achieves high coverage and comprises concepts of appropriate granularity, since all concepts and most relations required in tools coping with variability in space, time, or both are represented. Consequently, the conceptual model fills the gap between SPLE and SCM by unifying concepts and aligning the terminology of both domains. It supports the understanding of existing tools, and thus can prevent duplicated tool development. Moreover, it provides a means for researchers and practitioners to compare their work and clearly communicate it based on a unified terminology. As future work, we plan to formally specify the semantics of the concepts and relations, and to analyze the operations of tools to also unify their behavior in the conceptual model.

ACKNOWLEDGMENTS

We thank everyone supporting the construction of the conceptual model, especially Ina Schaefer, Michael Nieke, and Uwe Ryssel. This work has been partially supported by the German Research Foundation within the projects VariantSync (KE 2267/1-1) and EXPLANT (LE 3382/2-3, SA 465/49-3).

REFERENCES

[1] Frederik Ahlemann and Gerold Riempp. 2008. RefMod^{PM}: A Conceptual Reference Model for Project Management Information Systems. *Wirtschaftsinformatik* 50, 2 (2008). https://doi.org/10.1365/s11576-008-0028-y

[2] Sofia Ananieva, Thorsten Berger, Andreas Burger, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, S. Ramesh, Gabriele Taentzer, and Bernhard Westfechtel. 2019. Conceptual Modeling Group. In *Software Evolution in Time and Space: Unifying Version and Variability Management (Dagstuhl Seminar 19191)*, Thorsten Berger, Marsha Chechik, Timo Kehrer, and Manuel Wimmer (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik. https://doi.org/10.4230/DagRep.9.5.1

[3] Sofia Ananieva, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, S. Ramesh, Andreas Burger, Gabriele Taentzer, and Bernhard Westfechtel. 2019. Towards a Conceptual Model for Unifying Variability in Space and Time. In *International Systems and Software Product Line Conference (SPLC)*. ACM. https://doi.org/10.1145/3307630.3342412

[4] 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 (VaMoS)*. ACM. https://doi.org/10.1145/3168365.3168377

[5] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer. https://doi.org/10.1007/978-3-642-37521-7

[6] Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. 2009. Model Superimposition in Software Product Lines. In *International Conference on Theory and Practice of Model Transformations (ICMT)*. Springer. https://doi.org/10.1007/978-3-642-02408-5_2

[7] Timo Asikainen, Tomi Männistö, and Timo Soininen. 2006. A Unified Conceptual Foundation for Feature Modelling. In *International Software Product Line Conference (SPLC)*. IEEE. https://doi.org/10.1109/SPLINE.2006.1691575

[8] Rabih Bashroush, Muhammad Garba, Rick Rabiser, Iris Groher, and Goetz Botterweck. 2017. CASE Tool Support for Variability Management in Software Product Lines. *ACM Computing Surveys* 50, 1 (2017). https://doi.org/10.1145/3034827

[9] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *International Conference on Software Product Lines (SPLC)*. Springer. https://doi.org/10.1007/11554844_3

[10] Maurice H. ter Beek, Klaus Schmid, and Holger Eichelberger. 2019. Textual Variability Modeling Languages: An Overview and Considerations. In *International Systems and Software Product Line Conference (SPLC)*. ACM. https://doi.org/10.1145/3307630.3342398

[11] 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. https://doi.org/10.4230/DagRep.9.5.1

[12] Danilo Beuche. 2013. pure::variants. In *Systems and Software Variability Management - Concepts, Tools and Experiences*, Rafael Capilla, Jan Bosch, and Kyo Chul Kang (Eds.). Springer. https://doi.org/10.1007/978-3-642-36583-6_12

[13] Jan Bosch. 2010. Toward Compositional Software Product Lines. *IEEE Software* 27, 3 (2010). https://doi.org/10.1109/MS.2010.32

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

[15] Reidar Conradi and Bernhard Westfechtel. 1998. Version Models for Software Configuration Management. *ACM Computing Surveys* 30, 2 (1998). https://doi.org/10.1145/280277.280280

[16] 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 Modelling of Software-Intensive Systems (VaMoS)*. ACM. https://doi.org/10.1145/2110147.2110167

[17] Krzysztof Czarnecki, Chang Hwan, Peter Kim, and KT Kalleberg. 2006. Feature Models are Views on Ontologies. In *International Software Product Line Conference (SPLC)*. IEEE. https://doi.org/10.1109/SPLINE.2006.1691576

[18] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE. https://doi.org/10.1109/CSMR.2013.13

[19] Jacky Estublier. 2000. Software Configuration Management: A Roadmap. In *Conference on the Future of Software Engineering (FOSE)*. ACM. https://doi.org/10.1145/336512.336576

[20] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. https://doi.org/10.1109/icsme.2014.61

[21] 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 (ICSE)*. IEEE. https://doi.org/10.1109/ICSE.2015.218

[22] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. 2014. Variability in Software Systems—A Systematic Literature Review. *IEEE Transactions on Software Engineering* 40, 3 (2014). https://doi.org/10.1109/TSE.2013.56

[23] Rohit Gheyi, Tiago Massoni, and Paulo Borba. 2008. Algebraic Laws for Feature Models. *Journal of Universal Computer Science* 14, 21 (2008).

[24] Giancarlo Guizzardi, Luís Ferreira Pires, and Marten van Sinderen. 2005. An Ontology-Based Approach for Evaluating the Domain Appropriateness and Comprehensibility Appropriateness of Modeling Languages. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer. https://doi.org/10.1007/11557432_51

[25] Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2019. Software Product Line Engineering: A Practical Experience. In *International Systems and Software Product Line Conference (SPLC)*. ACM. https://doi.org/10.1145/3336294.3336304

[26] Martin Fagereng Johansen, Franck Fleurey, Mathieu Acher, Philippe Collet, and Philippe Lahire. 2010. Exploring the Synergies Between Feature Models and Ontologies. In *International Conference on Software Product Lines (SPLC)*.

[27] 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.

[28] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 2013. Consistency-Preserving Edit Scripts in Model Versioning. In *International Conference on Automated Software Engineering (ASE)*. IEEE. https://doi.org/10.1109/ASE.2013.6693079

[29] Max E. Kramer, Erik Burger, and Michael Langhammer. 2013. View-Centric Engineering with Synchronized Heterogeneous Models. In *International Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO)*. ACM. https://doi.org/10.1145/2489861.2489864

[30] Jacob Krüger. 2019. Are You Talking about Software Product Lines? An Analysis of Developer Communities. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM. https://doi.org/10.1145/3302333.3302348

[31] Jacob Krüger, Sofia Ananieva, Lea Gerling, and Eric Walkingshaw. 2020. Third International Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution 2020). In *International Systems and Software Product Line Conference (SPLC)*. ACM. https://doi.org/10.1145/3382025.3414944

[32] 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 (ICSE)*. IEEE. https://doi.org/10.1109/ICSE.2009.5070568

[33] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *International Conference on Generative Programming: Concepts & Experience (GPCE)*. ACM. https://doi.org/10.1145/3136040.3136054

[34] Lukas Linsbauer, Alexander Egyed, and Roberto Erick Lopez-Herrejon. 2016. A Variability Aware Configuration Management and Revision Control Platform. In *International Conference on Software Engineering (ICSE)*. ACM. https://doi.org/10.1145/2889160.2889262

[35] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability Extraction and Modeling for Product Variants. *Software and Systems Modeling* 16, 4 (2017). https://doi.org/10.1007/s10270-015-0512-y

[36] Lukas Linsbauer, Somayeh Malakuti, Andrey Sadovykh, and Felix Schwägerl. 2018. 1st Intl. Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution). In *International Systems and Software Product Line Conference (SPLC)*. https://doi.org/10.1145/3233027.3241372

[37] Jon Loeliger and Matthew McCullough. 2012. *Version Control with Git*. O'Reilly.

[38] Stephen A. MacKay. 1995. The State of the Art in Concurrent, Distributed Configuration Management. In *International Workshop on Software Configuration Management (SCM)*. Springer. https://doi.org/10.1007/3-540-60578-9_17

[39] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer. https://doi.org/10.1007/978-3-319-61443-4

[40] Damir Nešić, Jacob Krüger, Ștefan 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 (ESEC/FSE)*. ACM. https://doi.org/10.1145/3338906.3338974

[41] 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 (VaMoS)*. ACM. https://doi.org/10.1145/3023956.3023962

[42] Michael Nieke, Lukas Linsbauer, Jacob Krüger, and Thomas Leich. 2019. Second International Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution 2019). In *International Systems and Software Product Line Conference (SPLC)*. ACM. https://doi.org/10.1145/3336294.3342367

[43] Linda M. Northrop. 2002. SEI's Software Product Line Tenets. *IEEE Software* 19, 4 (2002). https://doi.org/10.1109/ms.2002.1020285

[44] Object Management Group (OMG). 2014. *Object Constraint Language*.

[45] David L. Parnas. 1976. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering* SE-2, 1 (1976). https://doi.org/10.1109/

TSE.1976.233797

[46] Juliana Alves Pereira, Kattiana Constantino, and Eduardo Figueiredo. 2015. A Systematic Literature Review of Software Product Line Management Tools. In *International Conference on Software Reuse (ICSR)*. Springer. https://doi.org/10.1007/978-3-319-14130-5_6

[47] 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 (ASE)*. IEEE. https://doi.org/10.1109/ASE.2015.106

[48] 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 (SPLC)*. ACM. https://doi.org/10.1145/3336294.3336299

[49] Christopher Pietsch, Christoph Seidl, Michael Nieke, and Timo Kehrer. 2020. Delta-Oriented Development of Model-Based Software Product Lines with DeltaEcore and SiPL: A Comparison. In *Model Management and Analytics for Large Scale Systems*, Bedir Tekinerdogan, Önder Babur, Loek Cleophas, Mark van den Brand, and Mehmet Akşit (Eds.). Elsevier. https://doi.org/10.1016/B978-0-12-816649-9.00017-X

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

[51] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering*. Springer. https://doi.org/10.1007/3-540-28901-1

[52] Julia Rubin and Marsha Chechik. 2013. A Framework for Managing Cloned Product Variants. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2013.6606686

[53] Nayan B. Ruparelia. 2010. The History of Version Control. *ACM SIGSOFT Software Engineering Notes* 35, 1 (2010), 5–9. https://doi.org/10.1145/1668862.1668876

[54] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *International Conference on Software Product Lines (SPLC)*. Springer. https://doi.org/10.1007/978-3-642-15579-6_6

[55] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software Diversity: State of the Art and Perspectives. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012). https://doi.org/10.1007/s10009-012-0253-y

[56] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic Semantics of Feature Diagrams. *Computer Networks* 51, 2 (2007). https://doi.org/10.1016/j.comnet.2006.08.008

[57] Felix Schwägerl. 2018. *Version Control and Product Lines in Model-Driven Software Engineering*. Ph.D. Dissertation. University of Bayreuth.

[58] 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 (ASE)*. ACM. https://doi.org/10.1145/2970276.2970288

[59] 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). https://doi.org/10.1007/s10270-019-00722-3

[60] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. Capturing Variability in Space and Time with Hyper Feature Models. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM. https://doi.org/10.1145/2556624.2556625

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

[62] 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 (SPLC)*. ACM. https://doi.org/10.1145/2648511.2648514

[63] 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 (SPLC)*. ACM. https://doi.org/10.1145/3336294.3336302

[64] Ștefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. https://doi.org/10.1109/icsm.2015.7332461

[65] 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 Software Product Line Conference (SPLC)*. ACM. https://doi.org/10.1145/3307630.3342414

[66] Birgit Vogel-Heuser, Christoph Legat, Jens Folmer, and Stefan Feldmann. 2014. *Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit*. Technical Report TUM-AIS-TR-01-14-02. Technical University of Munich.

[67] Bernhard Westfechtel, Bjørn P. Munch, and Reidar Conradi. 2001. A Layered Architecture for Uniform Version Management. *IEEE Transactions on Software Engineering* 27, 12 (2001). https://doi.org/10.1109/32.988710