Empirical Software Engineering manuscript No.
(will be inserted by the editor)

A Conceptual Model for Unifying Variability in Space and Time: Rationale, Validation, and Illustrative Applications

Sofia Ananieva · Sandra Greiner · Timo Kehrer · Jacob Krüger · Thomas Kühn · Lukas Linsbauer · Sten Grüner · Anne Koziolek · Henrik Lönn · S. Ramesh · Ralf Reussner

Received: date / Accepted: date

Sofia Ananieva
FZI Research Center for Information Technology Berlin, Germany
E-mail: ananieva@fzi.de

Sandra Greiner
University of Bayreuth, Germany
E-mail: Sandra1.Greiner@uni-bayreuth.de

Timo Kehrer
Humboldt University of Berlin, Germany
E-mail: timo.kehrer@informatik.hu-berlin.de

Jacob Krüger
Ruhr-University Bochum & Otto-von-Guericke University Magdeburg, Germany
E-mail: jacob.krueger@rub.de

Thomas Kühn
Karlsruhe Institute of Technology, Germany
E-mail: thomas.kuehn@kit.edu

Lukas Linsbauer
Technical University of Braunschweig, Germany
E-mail: l.linsbauer@tu-braunschweig.de

Sten Grüner
ABB Corporate Research Center, Germany
E-mail: sten.gruener@de.abb.com

Anne Koziolek
Karlsruhe Institute of Technology, Germany
E-mail: koziolek@kit.edu

Henrik Lönn
Volvo Group Trucks Technology, Sweden
E-mail: Henrik.Lonn@volvo.com

S. Ramesh
General Motors Global R&D Bangalore, India
E-mail: ramesh.s@gm.com

Ralf Reussner
Karlsruhe Institute of Technology, Germany
E-mail: reussner@kit.edu

Abstract With the increasing demand for customized systems and rapidly evolving technology, software engineering faces many challenges. A particular challenge is the development and maintenance of systems that are highly variable both in space (concurrent variations of the system at one point in time) and time (sequential variations of the system, due to its evolution). Recent research aims to address this challenge by managing variability in space and time simultaneously. However, this research originates from two different areas, software product line engineering and software configuration management, resulting in non-uniform terminologies and a varying understanding of concepts. These problems hamper the communication and understanding of involved concepts, as well as the development of techniques that unify variability in space and time. To tackle these problems, we performed an iterative, expert-driven analysis of existing tools from both research areas to derive a conceptual model that integrates and unifies concepts of both dimensions of variability. In this article, we first explain the construction process and present the resulting conceptual model. We validate the model and discuss its coverage and granularity with respect to established concepts of variability in space and time. Furthermore, we perform a formal concept analysis to discuss the commonalities and differences among the tools we considered. Finally, we show illustrative applications to explain how the conceptual model can be used in practice to derive conforming tools. The conceptual model unifies concepts and relations used in software product line engineering and software configuration management, provides a unified terminology and common ground for researchers and developers for comparing their works, clarifies communication, and prevents redundant developments.

Keywords Product Lines · Variability · Version Control · Revision Management

1 Introduction

Modern software systems exist in many variations to fulfill, for instance, different customer requirements, hardware limitations, and regulations (Apel et al. 2013; Estublier 2000; Pohl et al. 2005; Stănciulescu et al. 2015). Each variation may be distinguished based on which dimension it stems from (Ananieva et al. 2019b; Conradi and Westfechtel 1998; Strüber et al. 2019). First, variations can be implemented as feature options in a system. This allows developers to mass-customize products of the system by enabling or disabling its features, which represent abstract concepts to describe user-visible functionalities (Apel et al. 2013). The concepts relating to such *variability in space* are extensively studied in the context of software product line engineering (SPLE) (Apel et al. 2013; Pohl et al. 2005). As a concrete example, the Linux kernel has more than 15,000 feature options that allow developers to use it in embedded systems, servers, operating systems, or distributed computer clusters. Second, variations may be the result of a system's evolution. More precisely, a specific feature is only available in a system after it has been developed, and previous revisions can be deployed without that feature. The concepts relating to such *variability in time* are studied in the context of software configuration management (SCM) (Estublier 2000) and version control systems (VCSs) (Ruparelia 2010).

The missing foundation and tooling for supporting the evolution of variability proactively during a system's development has led to numerous approaches in the software product line (SPL) community that each tackle a subset of the resulting problems (Dintzner et al. 2016; Gamez and Fuentes 2011; Kröher et al. 2018; Nunes et al. 2012; Passos et al. 2013; Schulze et al. 2016). For example, retroactively mining feature evolution information from VCSs is only necessary, because VCSs do not support or track feature evolution proactively (Dintzner

et al. 2016; Kröher et al. 2018). Not explicitly tracking variability evolution may also incur additional costs (Krüger and Berger 2020). Unfortunately, common VCSs (e.g., Git) do not have a feature concept at all. Instead, variability is managed by creating one branch per product, which requires a high manual effort to maintain the products via merging between branches (Conradi and Westfechtel 1998). Simply combining existing approaches for managing both variability dimensions does also not suffice, since developers need to deal with a heterogeneous tool landscape (e.g., a VCS; potentially multiple variability mechanisms, such as a preprocessor and a build system as in the Linux kernel; and variability mining tools)—which hampers cross-dimensional variability modeling and analyses, for instance, for capturing the volatility (i.e., frequency of change) of a feature. The combination of SPLE and SCM, and thus explicit proactive management of variability in space and time during a system's development, aims at solving these problems and has only recently received increasing attention (Berger et al. 2019; Kehrer et al. 2021; Krüger et al. 2020; Linsbauer et al. 2018; Nieke et al. 2019; Strüber et al. 2019; Thüm et al. 2019).

A prerequisite for advancing this combination and avoiding redundant research is a well-defined and established understanding of the concepts and relations of both areas. Particularly, both areas rely on varying, but also synonymous, terms to refer to their concepts. For instance, "configuration" in SPLE refers to a valid selection of features (we call this a *feature configuration*), while in SCM it refers to a particular revision of the system (we call this a *revision configuration*). Such ambiguities can cause various problems that require a conceptual model to provide a unified understanding of both areas. For example, several literature reviews (Bashroush et al. 2017; Linsbauer et al. 2017a; Pereira et al. 2015; Ruparelia 2010) indicate a growth of research and tools from either research area that tackle the same problems using the same concepts. By providing a unified understanding of the terms, concepts, and relations established in both areas, a conceptual model supports researchers and developers in comparing their works, clarifying communication and reducing redundant research. As a consequence, it helps to proactively avoid many of the problems tackled in research on variability evolution and mining (e.g., the mining of feature evolution from source code repositories (Dintzner et al. 2016; Kröher et al. 2018)). In our previous work (Ananieva et al. 2019b, 2020), we described how we constructed a conceptual model relating the concepts of SPLE and SCM to provide a unified foundation of both areas, while also introducing hybrid concepts that emerge from the combination of variability in space and time. To derive the conceptual model, we systematically elicited concepts of 10 tools (cf. Section 4) from SPLE (e.g., FeatureIDE (Meinicke et al. 2017)), SCM (e.g., Git (Loeliger and McCullough 2012)), and both areas (e.g., ECCO (Fischer et al. 2015)). We interviewed developers and experts of these tools to identify the concepts and relations used. During a series of workshops, we constructed the model and adopted four ontology-based metrics (Guizzardi et al. 2005) to validate its coverage and granularity. In this article, we extend upon our previous work by providing additional insights regarding our construction and validation processes. Moreover, we add static semantics to the conceptual model, improve its description, provide detailed examples on the validation and practical use of the model, and conduct a formal concept analysis (FCA) to illustrate and discuss how the tools we analyzed relate to each other and to the conceptual model.

In more detail, our contributions in this article are (extensions are highlighted in *italics*):

- We report how we constructed the conceptual model for unifying concepts of variability in space and time as well as their relations (Section 2).
- We explain the conceptual model, its properties, and its concepts (Section 6).
- *We define static semantics, such as well-formedness rules, of the conceptual model using the object constraint language (OCL) (Section 6.3).*

- *We explain the design decisions that had major impact on the terminology and structure of the conceptual model (Section 6.1).*
- We provide an empirical validation of the conceptual model to show how well it covers existing tools (Section 7).
- *We extend our qualitative analysis by also including the well-formedness rules. Equivalently to concepts and relations, we provide and discuss a mapping between the rules and each tool (Section 7.1).*
- *We add details to our quantitative analysis. Specifically, we provide concrete examples to illustrate how each metric is applied to compare a tool to the conceptual model. Furthermore, we present additional results by providing each metric result per variability in space, time, and both (Section 7.2).*
- *We conduct an FCA of the conceptual model and the tools we analyzed, discussing how tools can be compared and on what dimensions of variability they focus. (Section 7.3).*
- *We add illustrative applications of the conceptual model to show how to use the model in practice when developing conforming tools (Section 8).*
- We publish an open-access repository with all data relating to the construction, validation, and analysis of the conceptual model.[1]

The conceptual model and our examples provide a foundation for guiding researchers and developers in obtaining a unified perspective on the concepts of variability in space and time. Consequently, it supports scoping, implementing, and communicating new research and tools that aim to combine both dimensions.

2 Background

In this section, we first describe an exemplary product line that we use as running example throughout this article. Using this example, we then explain variability in space and time (which we refer to as *variability dimensions*), as well as the combination of both. Finally, we introduce an initial version of the conceptual model that we developed during a Dagstuhl seminar (Berger et al. 2019) and extended systematically to obtain the conceptual model we describe in this article.

2.1 Running Example: Pick and Place Unit

The *pick and place unit (PPU)* is a demonstrator for the evolution in industrial plant automation[2] introduced by Vogel-Heuser et al. (2014). Its purpose is to take work pieces from a stack and move them around a shop floor via conveyor belts and a crane. As running example in this article, we use excerpts of the PPU control *software* product line. Particularly, in our example, the PPU comprises the mandatory component crane and a stack from which the work items are taken. The PPU software product line implements the operational control for these components. The crane moves work pieces that have been placed on the stack, and can utilize either a micro switch or an inductive sensor. Optionally, the stack may be extended with an optical sensor. In the following sections, we exemplify the implementation of the PPU, particularly highlighting the inherent variability of the system.

[1] `https://github.com/SofiaAnanieva/EMSE2021-ConceptualModel-Artifacts` (will be published on Zenodo)

[2] `https://www.mw.tum.de/ais/forschung/demonstratoren/ppu/`

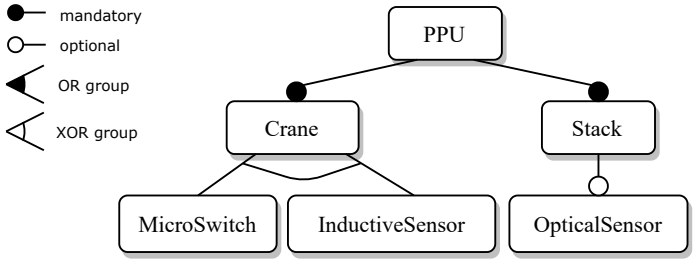
2.2 Variability in Space

Variability in space enables developers to systematically engineer a configurable system based on principles, methods, and concepts of SPLE (Apel et al. 2013; Clements and Northrop 2001; Parnas 1976; Pohl et al. 2005). In general, SPLE distinguishes between *problem space* and *solution space* (Apel and Kästner 2009). The problem space involves concepts to describe the domain, such as requirements and the variability of the product line (e.g., via feature models (Batory 2005; Czarnecki et al. 2012; Kang et al. 1990; Nešić et al. 2019)). The solution space involves concepts to implement the product line. Within a product line, a *platform* comprises all implementation artifacts, which are mapped to their corresponding features and can be configured (e.g., enabled or disabled) to automatically derive a customized system. For this purpose, the provided feature configuration is checked against the dependencies specified in the problem space (e.g., in a feature model) to ensure that it is valid (i.e., fulfills all dependencies). In this article, we use the term *feature* in its classical sense: A feature is "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems" (Kang et al. 1990) that must be "implemented, tested, delivered, and maintained" (Kang et al. 1998).

How a customized system is derived from the platform depends on the employed variability mechanism (Apel and Kästner 2009; Gacek and Anastasopoules 2001; Svahnberg et al. 2005). Annotative mechanisms (Apel et al. 2009a) rely on a single code base from which unwanted features are removed. Essentially, the developers implement a superimposition of all systems in the product line, and annotate the implementation artifacts with presence conditions. A presence condition is a Boolean expression over features, for example, in the form of preprocessor directives. To derive a concrete system, implementation artifacts whose presence conditions are not satisfied by the specified feature configuration are removed. Conversely, compositional variability mechanisms (Bosch 2010) extend a core system with features to derive a different, customized system. For this purpose, implementation artifacts are contained in feature modules (e.g., components) that summarize all artifacts relating to a specific presence condition (i.e., a feature or feature interaction). To derive a concrete system, a composer merges all modules specified in the feature configuration based on a defined feature order. Finally, transformational (a.k.a. delta-oriented) mechanisms (Schaefer et al. 2010) implement variability based on a core product and delta modules. In contrast to feature modules, a delta module comprises a sequence of delta operations and a presence condition. Delta operations can be used to add or remove implementation artifacts. To derive a customized system, the delta operations of all delta modules whose presence conditions are fulfilled by the feature configuration are applied in a specified order.

In Figure 1, we capture the variability in space of our PPU example using a feature model that incorporates the mandatory features *Crane* and *Stack*, the optional feature *OpticalSensor*, and an alternative group allowing the crane to possess either a *MicroSwitch* or an *InductiveSensor*. Finally, the cross-tree constraint $\neg \mathit{OpticalSensor} \vee \neg \mathit{InductiveSensor}$ below the actual model prohibits that the features *OpticalSensor* and *InductiveSensor* can coexist.

In Listings 1 and 2, we display an initial implementation of the PPU. For this example, we use an annotative variability mechanism based on preprocessor directives (i.e., `// #IF`, `// #END`) that encapsulate the optional lines of the source code. For example, the annotation in Line 2 in Listing 1 represents the presence condition MicroSwitch ∧ ¬InductiveSensor that guards Lines 3 and 4. Consequently, these lines are only included if the feature *MicroSwitch* is enabled and the feature *InductiveSensor* is *not* enabled. Further examples are the directives in Lines 2 and 4 in Listing 2, where Line 3 is only part of a system if the feature *OpticalSensor*



¬OpticalSensor ∨ ¬InductiveSensor

Fig. 1: Simplified feature model of the PPU.

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

Listing 1: Crane.java in its first revision.

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

Listing 2: Stack.java in its first revision.

```
1  public class Stack {
2    // #IF OpticalSensor
3      Stack(OpticalSensor s) {...}
4    // #END
5    // #IF !OpticalSensor
6      Stack() {...}
7    // #END
8    // #IF OpticalSensor
9      void exchangeOpticalSensor(OpticalSensor newSens) {...}
10   // #END
11 }
12 // #IF OpticalSensor
13 class OpticalSensor{}
14 // #END
```

Listing 3: Stack.java in its second revision.

is enabled in a feature configuration. Otherwise, Line 6 will be part of the system, due to the annotations in Lines 5 and 7.

2.3 Variability in Time

Variability in time involves concepts related to the evolution of a system. Concretely, SCM is concerned with VCSs that developers can use to manage software evolution and collaborative development. While some (academic) VCSs support versioning of almost arbitrary artifacts (Conradi and Westfechtel 1998), those established in practice (e.g., SVN (Pilato et al. 2008) or Git (Loeliger and McCullough 2012)) version files only. Developers can retrieve a local copy of the system from a common storage (i.e., a repository) and propagate local changes back to that storage. Each state (e.g., a commit) of the local copy that is propagated to the storage is referred to as a *revision*, which is marked with a (numbered) label to allow developers to restore a specific state. In contrast to feature configurations in SPLE, a system at a specific state (i.e., a revision configuration) can be restored without knowing whether that state is fully functional or incorporates specific patches and features.

For instance, Listings 1 and 2 in the running example represent the first revision of the PPU propagated to a VCS. Then, another developer retrieves that revision, modifies their local copy, and propagates these modifications back to the storage. In this scenario, the code we exemplify in Listing 3 represents (part of) the second revision. Concretely, this revision extends the implementation of the class *Stack* (i.e., in Lines 8–10).

2.4 Variability in Space and Time

In practice, concepts of variability in space and time are always connected: A product line evolves over time, and a VCS can manage various features or systems in separated branches or forks (Krüger 2019; Rubin and Chechik 2013; Stănciulescu et al. 2015). However, the combination of both dimensions has been examined less often and missing systematic tool support may cause inconvenient scenarios. For example, implementing and maintaining individual systems in branches introduces maintenance overheads if these systems must be synchronized (e.g., when propagating features or bug fixes between branches) (Dubinsky et al. 2013; Kehrer et al. 2014; Krüger and Berger 2020; Rubin and Chechik 2013). In contrast, many existing solutions for SPLE only support variability in space. Consequently, these solutions require developers to integrate an additional VCS to support variability in time, but that VCS is typically unaware of the variability in space. More advanced tools for managing both dimensions simultaneously and consistently are missing, and thus key functionalities, such as tracing the evolution of individual features, are hardly available.

Considering variability in space and time simultaneously may solve such problems. In this direction, Westfechtel et al. (2001) proposed the *uniform version model* that aims to unify concepts of SPLE and SCM. The uniform version model introduces the *version* as an abstract unification that can either be a customized system (variability in space) or a revision (variability in time). To ensure consistency, this model prescribes a specific development process, which however contradicts the practical use of most contemporary tools. For that reason, the uniform version model cannot serve as a conceptual model to cover and advance the state of the art.

In the running example, the implementation of the class `Stack` is modified, resulting in a second revision. Inside the class, the feature *OpticalSensor* is modified. In the first revision (cf. Listing 2), only the constructor exists while the second revision (cf. Listing 3) adds an exchange mechanism by introducing the method *exchangeOpticalSensor()* in Line 9. Consequently, the second revision revises not only the class *Stack*, but also the feature *OpticalSensor*; thus involving both variability dimensions: One specific feature (variability

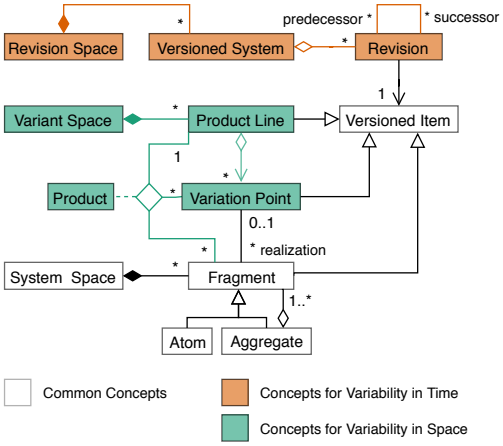


Fig. 2: The initial conceptual model for unifying concepts of variability in space (green) and in time (orange) with common concepts (white) (Ananieva et al. 2019b).

in space) is changed and thereby evolves from the first to the second revision (variability in time). Therefore, this modification is typically considered to represent a *feature revision*.

When using tools that do not manage variability in space and time simultaneously, such *feature revision* (i.e., a combination of variability in space and time) may involve multiple actions, such as adding an entirely new feature, refactorings, or bug fixes—making it hard for developers to understand and manage the variability of both dimensions.

2.5 Initial Conceptual Model

At a Dagstuhl Seminar on the topic of *Unifying Version and Variability Management* (Berger et al. 2019), a subgroup of its participants (Ananieva et al. 2019a) organized concepts of variability in space and time into a UML class model. We refer to the resulting model as *initial conceptual model* (Ananieva et al. 2019b). Based on interviews with tool developers and discussions during follow-up workshops, we refined the initial conceptual model by relying on the knowledge of experts in the area of managing both variability dimensions simultaneously (Ananieva et al. 2020). In the following, we briefly introduce the initial conceptual model that we display in Figure 2, which served as starting point for constructing the conceptual model we describe in this article.

The initial conceptual model distinguishes the `Revision Space` from the `Variant Space` as well as from the `System Space`, and categorizes all elements according to the variability dimension they belong to. The `Revision Space` involves concepts relating only to variability in time, namely a `Versioned System` that is composed of `Revisions`. In contrast, the `Variant Space` covers variability in space, incorporating a `Product Line` from which `Products` can be derived by selecting `Variation Points` that are implemented by arbitrary `Fragments` (e.g., lines of code, model elements). The `Fragment` is the main concept of those common to both, variability in space and time. Finally, a `Versioned Item` connects both variability dimensions and the common concepts, essentially enabling version control on all concepts. The initial model documents the concepts and relationships existing

in each variability dimension. However, it provides no unification of these concepts, does not represent concepts used in contemporary tools, and was not systematically constructed. The conceptual model we present in this article considerably advances on this initial one, building on a systematic empirical process for the construction and validation. As a result, the conceptual model improves the unification of concepts, incorporates new concepts, and allows to understand as well as compare contemporary tools. In this article, we further extend our contributions on the conceptual model we presented in the previous conference paper (Ananieva et al. 2020) by defining static semantics, reasoning on specific design decisions, and explaining how to use the model in practice.

3 State of the Art

In this section, we introduce and discuss the state of the art of conceptual models in the areas of SPLE and SCM, as well as related surveys of variability in space and time.

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 (Apel et al. 2013; Northrop 2002; Pohl et al. 2005). 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 (Czarnecki et al. 2006, 2012; Johansen et al. 2010; Nešić et al. 2019; Schaefer et al. 2012). 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 (Conradi and Westfechtel 1998; MacKay 1995; Pilato et al. 2008; Ruparelia 2010). The most prominent concept to specify and capture variability in time is arguably the version model, which describes how the versions in a SCM system are managed. However, as Conradi and Westfechtel (1998) 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 (1998) who extend the version models identified towards capturing the relation of variability in space and time. Building on this idea, Westfechtel et al. (2001) introduce the uniform version model, which provides a common model for basic SCM and SPLE concepts. In some regards, this model is highly flexible and, as a consequence, overly generic. However, some aspects are intertwined with implementation details, such as propositional logic and deltas to manage variability. In contrast, we aim to devise a unified conceptual model that is as specific as possible, while still covering all relevant concepts dealing with variability in space and time without focusing on implementation options. Schwägerl (2018) builds upon the uniform version model, replacing some of the concepts and partly describing 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. (2017a) survey variation control systems, some of which support variability in space and time. So, we included this type of tools in our analysis, too. Other researchers compared tools for SPLE or SCM (Bashroush et al. 2017; Galster et al. 2014; Pereira et al. 2015; Pietsch et al. 2020; Ruparelia 2010). 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

In this section, we introduce the tools we analyzed to construct the conceptual model. First, we describe the key criteria for selecting the tools. Then, we present the tools according to the supported variability dimension.

4.1 Tool Selection

For constructing the conceptual model, we examined a representative set of available and commonly used tools. These tools cover i) solely the dimension of variability in space, ii) solely the dimension of variability in time, or iii) both dimensions jointly. Moreover, the tools must iv) allow to specify the problem space as well as implement the solution space, and v) be available as well as usable. Thus, we did not consider tools that support only the solution space (e.g., pure variability mechanisms, such as FeatureHouse (Apel et al. 2009b)) or only the problem space (e.g., variability modeling or analysis (Asikainen et al. 2006; Beek et al. 2019; Gheyi et al. 2008; Schobbens et al. 2007)).

A recent study (Horcas et al. 2019) of available and usable tools for SPLE shows that only 19 % out of the 97 examined tools are usable, and only a small subset of the 97 tools offers support for problem *and* solution space. The study also demonstrates that many of the discontinued tools, such as FeatureHouse (Apel et al. 2009b), have been integrated as variability mechanisms into FeatureIDE Kästner et al. (2009); Meinicke et al. (2017). Since we examined FeatureIDE, we considered many concepts of such tools.

Regarding the tools that support only variability in space, we covered each category of the main variability mechanisms (i.e., annotative, transformational, and compositional) through at least one tool. In addition to FeatureIDE, we also included an industrial tool for which we could interview a tool expert and access openly available documentation. For tools that support only variability in time, we analyzed the two most pervasive VCSs, Git and SVN. To reflect on tools that aim to manage variability in space *and* time simultaneously, we selected variation control systems (Linsbauer et al. 2021) that are (still) available and are grounded in a profound conceptual basis. For instance, SuperMod and VaVe both allow for versioning of models, instead of only text files, and apply different paradigms to represent and compute changes. While SuperMod employs a state-based comparison to create symmetric deltas, VaVe monitors changes and computes directed deltas. Overall, we incorporated diverse perspectives while addressing the unification of both variability dimensions for designing the conceptual model.

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 conceptual model, since they are covered implicitly by considering each tool individually. Particularly, in contrast to the tools

supporting both dimensions explicitly, these (artificial) combinations do not contribute new concepts of variability in space and time.

4.2 Tools for Variability in Space

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

FeatureIDE (Kästner et al. 2009; Meinicke et al. 2017) originates from academia and is a tool platform supporting the development of product lines based on the Eclipse platform. The tool includes not only extensive feature modeling, but also implementation, configuration, and testing support. FeatureIDE implements annotative and compositional variability mechanisms, covering these two mechanisms in our analysis.

pure::variants (Beuche 2013) is an industrial SPLE tool. While pure::variants builds on the Eclipse platform and covers different variability mechanisms as well, the tool 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, a main advantage of including pure::variants is its design for practitioners from industry, which allowed us to incorporate the practical and industrial perspective in our model. There are other proprietary tools similar to pure::variants, such as Gears from BigLever (Krueger and Clements 2012), which we did not consider in this work due to availability reasons.

SiPL (Pietsch et al. 2015, 2017, 2019) supports the implementation of model-based product lines based on a transformational variability mechanism. SiPL uses delta modules to capture variability in space, 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 (Kehrer et al. 2013) generated by comparing models. Moreover, edit scripts are an essential prerequisite for several quality-assurance techniques, which aim to detect and mitigate design flaws in the delta-oriented implementation of a product line.

4.3 Tools for Variability in Time

As representative tools for variability in time, we considered SVN and Git as well-established and widely used VCSs, covering a centralized and a decentralized system, respectively.

Subversion (SVN) (Pilato et al. 2008) is a centralized VCS (i.e., one central repository is stored on a server). SVN allows users to checkout 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 branches.

Git (Loeliger and McCullough 2012), in contrast to SVN, supports decentralized versioning (i.e., every user has their own copy of the entire repository evoking a distributed network of repositories). As such, Git supports local operations (e.g., a *commit* of changes to the local repository) as well as distributed operations (e.g., the *clone* operation that creates a local copy of the entire remote repository, the *push* and *pull* operations that are used to synchronize between clones of the repository).

4.4 Tools for Variability in Space and Time

In the following, we introduce the five contemporary tools that manage variability in space *and* time simultaneously that we analyzed.

ECCO (Fischer et al. 2014, 2015; Linsbauer et al. 2016, 2017b) was initially designed for re-engineering cloned systems 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/modify/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 (Schwägerl and Westfechtel 2016, 2019) is based on the uniform version model (Westfechtel et al. 2001), consequently unifying temporal revisions and spatial variants as *versions*. A product line is developed 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/modify/commit workflow locally, and allows multi-user development by pushing/pulling the local state to one remote repository server.

DeltaEcore (Seidl et al. 2014b,c) is a tool-suite for model-based SPLE based on a transformational variability mechanism. The tool automatically derives delta languages, which are used to express the delta operations to the common 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. DeltaEcore can be used in conjunction with a *hyper feature model* (Seidl et al. 2014a), which extends the notion of individual features with revisions (in contrast to revisions of the whole system, which are not explicitly supported).

DarwinSPL (Nieke et al. 2017) copes with variability in space and time, while integrating contextual information that restricts the configuration space of the product line. For product derivation, it integrates with DeltaEcore. In contrast to DeltaEcore, DarwinSPL captures the evolution of the whole system with a *temporal feature model*, and thus supports the planning for the future evolution of a product line.

VaVe (Ananieva et al. 2018) integrates the management of *VA*riants (space) and *VE*rsions (time). It builds on VITRUVIUS (Klare et al. 2021; Kramer et al. 2013), a view-based framework that supports consistent system development by providing multiple languages to preserve consistency between views. Specifically, VaVe aims to extend VITRUVIUS with capabilities for variability management by introducing the problem space and extending the original consistency preserving mechanisms with variability-related consistency preservation regarding problem space and solution space.

5 Construction Process

In this section, we describe the construction process of the conceptual model for unifying variability in space and time, which we show in Figure 3. We followed an informed design methodology inspired by the work of Ahlemann and Riempp (2008) who propose iterative steps, such as expert interviews and refinements of the model until consensus is reached. Therefore, we made the deliberate choice to include all available tools fitting our key criteria

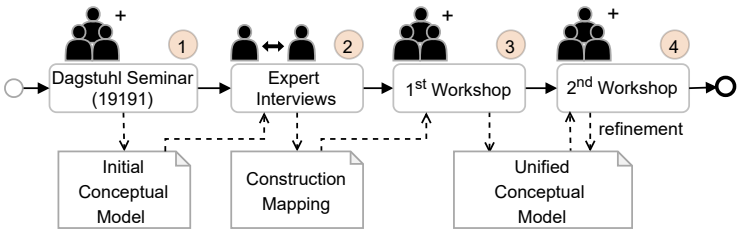


Fig. 3: Construction process of the unified conceptual model.

in the construction process of the unified model. In the following, we describe each step of the construction process.

5.1 Dagstuhl Seminar (19191)

During a Dagstuhl seminar on *Software Evolution in Time and Space: Unifying Version and Variability Management* (Berger et al. 2019), we developed the initial conceptual model as introduced in Section 2.5 and shown in Figure 2 (cf. step ①). The initial model documents concepts for variability in space (e.g., `Feature`) and in time (e.g., `Revision`) as well as their relations. However, this initial model does not address the unification of these concepts.

5.2 Expert Interviews

Following an empirical construction process for unifying concepts of variability in space and time, the initial model served as input to expert interviews (cf. step ②). In particular, we (specifically, the first author of this article) conducted semi-structured interviews with one tool expert per tool. The goal was to understand to what extent the initial model captures concepts of contemporary tools and what adaptations were needed to derive a unified model.

We invited tool experts that are closely involved in the conceptual design or implementation of the respective tool, and thus are among the most knowledgeable experts for each tool. Most of the tool experts are researchers from academia, while we also involved one expert from industry. Note that we did not conduct interviews on Git and SVN, because these 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 completeness and consistency. The eight interviews took 83 minutes on average.

The interview guide involved four parts. In the first part, we introduced the initial conceptual model and definitions of the involved concepts. The second part asked for a mapping of concepts of the initial 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 model, what are the semantically equivalent tool constructs?
- Is there a tool construct not represented by any concept of the model?

During the third part, we elicited the main use cases of each tool and its scope to distinguish the tools from each other. Finally, the fourth part encompassed tool operations (e.g., code analysis) to obtain a holistic understanding of each tool.

5.3 Construction Mapping

The expert interviews resulted in a construction mapping for each tool, where tool constructs were mapped to the concepts of the *initial* conceptual model. Based on the construction mappings, we performed informed improvements to the initial conceptual model. For example, we decided whether new model concepts needed to be introduced and existing ones removed, merged, or split up, by discussing how these concepts mapped to constructs of the studied tools. In the following, we describe the insights gained from these mappings.

Overall, we could map the majority of tool constructs to at least one concept in the initial conceptual model. However, we also identified tool constructs that did not map to any concepts of the 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 as 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`. However, in the initial conceptual model, these constructs are only implicit: 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`.

5.4 Workshops

The construction mappings served as input to a series of closed, dedicated workshops organized for building the unified conceptual model. Participants involved tool experts we interviewed before, authors of this article, and further researchers of the SPLE and SCM communities that became aware of this effort during the presentation of the initial conceptual model (Ananieva et al. 2019b) and voiced their interest to participate. During these workshops, the initial model was gradually refined into the unified conceptual model we present in this article. Specifically, we conducted two workshops (cf. steps ③ and ④). The first workshop was a one-day open discussion with loose moderation involving 15 participants. It was based on the prepared interview results and impulse questions. The second workshop involved 12 participants and lasted 1.5 hours. It included a presentation of the preliminary conceptual model based on the results of the first workshop, followed by a discussion of open issues and the opportunity for each participant to voice suggestions for improvement.

During both workshops, we gradually modified the initial conceptual model to obtain the unified model we present in Section 6. Major changes involved the unification of concepts that we found to be represented by a single construct in tools. For example, a tool that deals with variability either in space or in time involves the `Product Line` construct or the `Versioned System` construct, respectively. Tools that deal with both variability in space and time do not represent the two concepts with two constructs, but instead represent both as one unified construct. In other words, no tool involves an individual construct for both of the two concepts. Furthermore, we added concepts or made them explicit. For example, many tools

involve constructs for constraining valid configurations. However, this was not reflected in any concept of the initial model. Another example addresses the mapping between `Fragment` and `Variation Point`, which was only represented implicitly in the initial model as an association. Considering the significance this concept carries in most of the tools, we made the `Mapping` concept explicit. Additionally, we generalized some concepts that were previously assigned to one dimension only to also apply to the other dimension. For example, the concept `Configuration` was only connected to variability in space, which we extended to also refer to variability in time. Finally, we introduced new hybrid concepts and relations that do not exist in tools that focus on only one variability dimension.

6 The Conceptual Model

In this section, we first explain design decisions that mainly impacted the terminology and structure of the conceptual model. Then, we present the unified conceptual model and define additional static semantics of the model in the form of well-formedness rules.

6.1 Design Decisions

During the workshops, we discussed and agreed on the terminology and several design decisions, which we present in the following.

Terminology. Regarding terminology, we aimed for generic and unambiguous names that are not associated with either SPLE or SCM terminology. This especially affected the naming of concepts representing variability in space, time, or both. Since the term `Variation Point` is associated with SPLE and generally used in the implementation context, and the term `Variant` is ambiguous as it represents either a `Product` or in case of the Orthogonal Variability Model (Pohl et al. 2005) an `Option` of a `Variation Point`, we chose the generic term `Option` to refer to any kind of variation in space, time, or both.

Our second decision on the terminology affected the use of concepts that serve as containers for other concepts. In the initial conceptual model, these concepts were `Product Line` and `Versioned System` (associated with SPLE and SCM, respectively). As described in Section 5, the tools we analyzed do not distinguish between the two and represent both through a single construct. The term `Repository` is often associated with persistence, which is not relevant on a conceptual level. Therefore, we agreed on the term `Unified System` (as it represents a container for the concepts of space, time, or both).

Modeling Pragmatics. The following design decisions relate to the modeling itself. First, we decided on the structure of the concept `Fragment`. Most tools structure `Fragments` as trees (e.g., ECCO, GIT), some as graphs (e.g., DeltaEcore). Since a graph structure is a generalization of a tree structure, we decided to model `Fragments` as a graph (i.e., `Fragments` may reference an arbitrary number of further `Fragments`).

Second, we were concerned with the different types of revisions. `System Revisions` and `Feature Revisions` are not the same, since they represent `Revisions` of different concepts (i.e., `Unified System` and `Feature`, respectively). Therefore, we introduced the concept of `System Revision` as counterpart to `Feature Revision` to clearly differentiate between both types of revision.

Third, we focused on `Constraints`. A preliminary version of the conceptual model allowed to define `Constraints` not only on `Features` and `Feature Revisions`, but additionally on `System Revisions`. However, in none of the selected tools `Constraints`

operate on `System Revisions`. We thus introduced the concept `Feature Option` as superclass of `Feature` and `Feature Revision` to define `Constraints` only over these concepts.

Fourth, we discussed the dependency between `Feature Revision` and `Feature`. Since a `Feature Revision` cannot exist without the respective `Feature`, we decided to use a composition relation. This way, we aimed to explicitly highlight the strong dependence of a `Feature Revision` on its respective `Feature`.

Finally, versioning could additionally be applied to concepts that depend on versioned concepts, for instance, to `Configuration` or `Mapping`, which depend on `Option`. However, this would introduce cycles (i.e., `Mappings` and `Configurations` are both changed by `Options`, but they also refer to `Options`). This is also reflected by the fact that no tool versions these two concepts. We decided to align the conceptual model with the selected tools. However, this decision is a candidate for future adaptations, depending on how new tools that integrate variability in space and time may be designed.

Additional minor decisions involved that we avoided interfaces that are specific to the respective implementation and which we therefore did not consider relevant on the conceptual level. Nonetheless, we used abstract classes to ensure that `Feature Option` and `Revision` can only be instantiated with their respective sub-classes, namely `System Revision`, `Feature Revision`, and `Feature`.

6.2 Concepts and Relations

In Figure 4, we show the conceptual model comprising 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 and italic font for abstract concepts. Relations are colored analogously. The model comprises two parts: The left side shows the problem space in SPLE, namely the abstraction of the domain, which is equivalent to the version space in SCM. The right side shows the solution space in SPLE, namely the actual implementation, which is equivalent to the product space in SCM (Conradi and Westfechtel 1998). Interestingly, all concepts for variability in space, time, or both are located in the problem space (left side of the model). All concepts in the solution space and on the border of both spaces are unified concepts, which are independent of the involved variability dimensions. In the following, we explain the model gradually from left to right, starting with concepts for variability in space followed by concepts for variability in time. Then, we introduce concepts for both dimensions. Finally, we conclude with the unified concepts.

Concepts and Relations for Variability in Space. The conceptual model represents variability in space using three concepts: `Feature Option` (abstract), `Feature`, and `Constraint`.

A `Feature Option` is an abstract concept with two concrete specializations, one being the `Feature`. A `Feature` represents a configuration option in space that can be selected or deselected. *Example:* The PPU involves six `Features` in total, for example, *Crane* or *Stack*.

Another concept for variability in space is the `Constraint`. `Constraints` express which `Feature Options` can, must, or must not be selected together. `Constraints` can be expressed in various ways, for example, as an arbitrary expression (e.g., a propositional formula) defined over `Feature Options` to constrain which combinations of `Feature Options` are valid. *Example:* In the PPU, the cross-tree constraint $\neg OpticalSensor \vee \neg InductiveSensor$ of the feature model exemplifies one `Constraint`.

Concepts and Relations for Variability in Time. The conceptual model covers variability in time using two concepts: `Revision` (abstract) and `System Revision`.

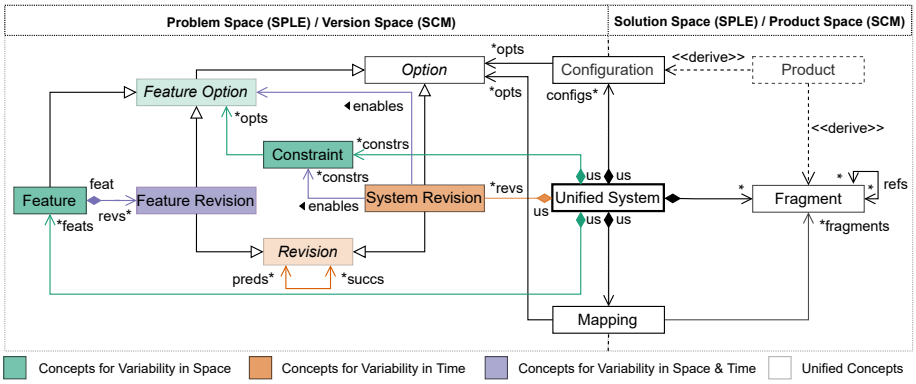


Fig. 4: UML class diagram of the conceptual model for unifying concepts for variability in space and time.

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 specialized into a `System Revision`, which represents the state of the whole system at a particular point in time. Note that the conceptual model does not enforce a certain notion of time. Instead, it uses the concept `Revision` as an abstract representation of time. A concrete implementation of the concept `Revision` (i.e., when building a concrete tool as we illustrate in Section 8) could employ sequential revision numbers (as SVN does), hashes (as Git does), real time, wall-clock time (as DarwinSPL does), or any other representation of time. *Example:* The PPU example involves revisions at two different points in time. `System Revisions` are used to refer to these points in time. Specifically, in the example, we refer to the earlier state as `System Revision` 1 and to the later state as `System Revision` 2, with the latter being a successor of the former.

Concepts and Relations for Variability in Space and Time. Concepts for variability in space and time are hybrid concepts not present in tools focusing solely on one dimension.

A `Feature Revision` represents variability in space and time as a combination of `Feature Option` and `Revision`. It is another specialization of `Feature Option` (next to `Feature`) and of `Revision` (next to `System Revision`). It represents the state of one particular `Feature` at one point in time. *Example:* In the PPU, the modification of the `Feature` *OpticalSensor* in class *Stack* (cf. Listing 3) can be considered a `Feature Revision` of the `Feature` *OpticalSensor* in addition to a `System Revision` of the entire PPU.

In contrast to a `Feature Revision`, which is local to a `Feature`, a `System Revision` also determines which `Feature Options` (i.e., `Features` and `Feature Revisions`) and `Constraints` are enabled. *Example:* The `System Revision` 1 of the PPU enables the `Feature Revision` 1 of the `Feature` *OpticalSensor*, while `System Revision` 2 enables `Feature Revision` 2 of the `Feature` *OpticalSensor*.

Unified Concepts and Relations. While concepts for variability in space and time are only relevant if both dimensions are involved simultaneously, unified concepts are relevant for either dimension at all times.

The central concept in the conceptual model is the `Unified System`. It contains most other concepts and essentially represents the developed system. *Example:* In the PPU example, the `Unified System` would simply represent the PPU in its entirety.

An `Option` is a high-level abstraction of any variation in space, time, or both of a `Unified System` in the problem space. It manifests either as `Feature` (variability in space), `System Revision` (variability in time), or `Feature Revision` (both).

A `Fragment` is the core concept to describe the implementation of a `Unified 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, or delta modules). We specify neither the level of granularity nor the purpose of `Fragments` to keep the conceptual model as generic as possible. *Example:* Every line in a Java file (e.g., in Listing 2), the file itself, or the containing folder may represent `Fragments`, depending on the implementation of the respective tool.

A `Mapping` connects `Options` with `Fragments`, and thus connects the solution space (`Fragments`) and the problem space (`Options`). A concrete representation of a mapping can, for example, be an expression (e.g., a propositional formula) over `Options`. It is possible that such `Mapping` expressions only consist of Boolean constants to govern the presence or absence of `Fragments` (e.g., core or dead `Fragments`). *Example:* Line 2 in Listing 1 represents a `Mapping` of a `Fragment` (i.e., the line of code) to `Options`, namely *MicroSwitch* && !*InductiveSensor*.

The `Configuration` exists in different forms in both areas, SCM and SPLE (cf. Section 1). To align both perspectives, we unify its meaning: a `Configuration` is a selection of `Options` used to derive a specific `Product`. *Example:* In the PPU, a `Configuration` may select the first `Feature Revision` of the `Features` *Crane* and *Stack*, and the second `Feature Revision` of the `Feature` *OpticalSensor*.

In contrast to the previous concepts, `Products` are not contained in the `Unified System`. Based on a `Configuration`, a `Product` is derived by tool-specific mechanisms (e.g., delta modules) that are part of the tool's behavior. Such mechanisms specify which and how `Fragments` are composed.

6.3 Static Semantics

The expressiveness of UML class diagrams is limited to their static structure and not sufficient to express more complex well-formedness rules. The following additional rules are needed to also include static semantics of the studied tools in the unified model, which we identified and collected during interviews and discussions with the tool experts. For example, the revision graph in Git must be acyclic. Next, we first introduce auxiliary definitions that we then use to specify well-formedness rules using OCL (Object Management Group 2014).

Auxiliary Definitions. In Listing 4, we specify three auxiliary definitions to simplify some of the well-formedness rules. The first definition specifies an operation that collects all `Options` contained in a `Unified System`. These `Options` can be `System Revisions`, `Features`, and `Feature Revisions`. The second definition specifies an operation that collects all `System Revisions` in a `Configuration`. The third definition collects all `Feature Options` in a `Configuration`, which can be `Features` and `Feature Revisions`.

Well-Formedness. The following ten well-formedness rules specify static semantics of the conceptual model. In particular, these rules are concerned with the revision graph, the `Unified System`, and the relationship between `Feature Options`, `System Revisions`, and `Constraints`. Finally, we specify a well-formedness rule on a `Configuration`.

```
context UnifiedSystem
def:
  getAllOptions : Set(Option) =
    self.feats -> union(self.revs) -> union(self.feats -> collect(f:
        Feature | f.revs) -> flatten())

context Configuration
def:
  getAllSystemRevisions : Set(SystemRevision) =
    self.opts -> select(o:Option | o.oclIsTypeOf(SystemRevision))
def:
  getAllFeatureOptions : Set(FeatureOption) =
    self.opts -> select(o:Option | o.oclIsKindOf(FeatureOption))
```

Listing 4: Auxiliary definitions for well-formedness rules.

```
context Revision
-- Rule 1: Every predecessor of a Revision must have the Revision as
     successor.
inv:
  self.preds -> forAll(r : Revision | r.succs -> includes(self))
inv:
  self.succs -> forAll(r : Revision | r.preds -> includes(self))

-- Rule 2: The revision graph must be a directed acylic graph.
inv:
  self.succs -> closure(r : Revision | r.succs) -> excludes(self)

-- Rule 3: All Revisions of a revision graph must be of the same type and
     have the same container.
inv:
 self.preds -> forAll(r : Revision | self.oclType() = r.oclType() and
 self.container = r.container)
inv:
 self.succs -> forAll(r : Revision | self.oclType() = r.oclType() and
 self.container = r.container)
```

Listing 5: Well-formedness of the revision graph.

We display the first three rules in Listing 5, which specify the well-formedness of the revision graph:

Rule 1 ensures that a bidirectional relationship exists between every direct predecessor and successor of a `Revision`. Consequently, each direct predecessor of a `Revision` *r* references the `Revision` *r* as successor. Equivalently, each direct successor of a `Revision` *r* references *r* as a predecessor.

Rule 2 ensures that the revision graph must be a directed acyclic graph (DAG). Accordingly, the transitive closure over the successor revisions may not include the `Revision` itself.

Rule 3 ensures that a revision graph can only contain `Revisions` of the same type (i.e., either `System Revisions` or `Feature Revisions`). In other words, a revision graph is not allowed to intermingle `Feature Revisions` with `System Revisions`. Additionally, all `Revisions` of a revision graph must be contained in the same container (i.e., the `Unified System` for `System Revisions` or `Feature` for `Feature Revisions`).

In Listing 6, we introduce well-formedness rules that ensure the use of concepts belonging to the same `Unified System`:

```
1 -- Rule 4: All Options in a Configuration must be contained in the Unified
      System.
2 context Configuration
3 inv:
4   self.opts -> forAll(o:Option | self.us.getAllOptions -> includes(o))
5
6 -- Rule 5: All Fragments and Options in a Mapping must be contained in the
      Unified System.
7 context Mapping
8 inv:
9   self.opts -> forAll(o:Option | self.us.getAllOptions -> includes(o))
10 inv:
11   self.fragments -> forAll(f:Fragment | self.us.fragments -> includes(f))
12
13 -- Rule 6: All Feature Options in a Constraint must be contained in the
      Unified System.
14 context Constraint
15 inv:
16   self.opts -> forAll(
17     o:FeatureOption | self.us.getAllOptions -> includes(o)
18   )
```

Listing 6: Well-formedness of containments in a `Unified System`.

Rule 4 ensures that all `Options` in a `Configuration` must be contained in the enclosing `Unified System`. Consequently, there can be no `Configuration` that refers to `Options` that belong to other instances of a `Unified System` than the `Configuration`.

Rule 5 ensures that all `Fragments` and `Options` in a `Mapping` must be contained in the enclosing `Unified System`. Consequently, this rule forbids the use of `Fragments` and `Options` belonging to other instances of a `Unified System` than the `Mapping`.

Rule 6 ensures that all `Feature Options` in a `Constraint` must be contained in the enclosing `Unified System`. Therefore, there can be no `Constraints` with `Feature Options` belonging to other instances of a `Unified System` than the `Constraint`.

In Listing 7, we introduce well-formedness rules that specify the static semantics of the *enables* association between `System Revision` and `Feature Option` as well as between `System Revision` and `Constraint`:

Rule 7 ensures that all `Feature Options` as well as `Constraints` enabled by a `System Revision` must be contained in the enclosing `Unified System`. Therefore, there can be no `Feature Options` and `Constraints` enabled by a `System Revision` belonging to other instances of a `Unified System` than the `System Revision`.

Rule 8 ensures that all `Constraints` enabled by a `System Revision` can only refer to `Feature Options` enabled by the same `System Revision`. Consequently, there can be no enabled `Constraints` that refer to `Feature Options` enabled by other `System Revisions` than the `Constraint`. This rule, in particular, refers to tools dealing with variability in space and time with multiple `System Revisions`.

Rule 9 ensures that if a `System Revision` enables a `Feature`, then it must also enable a `Feature Revision` of the same `Feature` (unless the `Feature` has no `Feature Revisions`). Consequently, if a `Feature` is enabled, then at least one of its `Feature Revisions` must also be enabled. This rule, in particular, refers to tools dealing with variability in space and time using `System Revisions` and `Feature Revisions`.

Finally, we specify the static semantics for the well-formedness of a `Configuration` in Listing 8:

```
-- Rule 7: All Feature Options and Constraints enabled by a System
     Revision must be contained in the Unified System.
context SystemRevision
inv:
  self.opts -> forAll(c:Constraint | self.us.constrs -> includes(c))
inv:
  self.opts -> forAll(f:FeatureOption| self.us.getAllOptions ->
     includes(f))

-- Rule 8: All Constraints enabled by a given System Revision can only
     include Feature Options enabled by the same System Revision.
inv:
  self.constrs -> forAll(c:Constraint | self.opts -> includesAll(c.opts))

-- Rule 9: For every Feature that is enabled by a System Revision, the
     same System Revision must also enable at least one Feature Revision
     of the same Feature.
inv:
  self.opts -> select(o:Option | o.oclIsTypeOf(Feature)) -> forAll(
    f:Feature |  f.revs -> isEmpty() or f.revs -> exists(fr:
         FeatureRevision | self.opts.contains(fr))
  )
```

Listing 7: Well-formedness of the *enables* relations of `System Revision`.

```
-- Rule 10: A Configuration may only refer to Feature Options that are
     enabled by at least one System Revision in the same Configuration.
context Configuration
inv:
  self.opts -> exists(o:Option | o.oclIsTypeOf(SystemRevision)) implies (
    self.getAllFeatureOptions -> forAll(
      f:FeatureOption | self.getAllSystemRevisions -> exists(s:
          SystemRevision | s.opts.includes(f))
    )
  )
```

Listing 8: Well-formedness of `Configuration`.

Rule 10 ensures that, if a `Configuration` refers to at least one `System Revision`, then it may only refer to `Feature Options` that are enabled by at least one of these `System Revisions`. Note, that if `Configuration` does not refer to any `System Revisions`, then all `Feature Options` are enabled.

7 Validation

In this section, we describe our validation of the unified conceptual model. The validation comprises a qualitative analysis based on a questionnaire, and a quantitative analysis based on metrics. In addition, we performed a formal concept analysis (FCA) that provides a comprehensive visualization of relations between the tools on the one hand and the concepts and relations of the conceptual model on the other hand. Our analysis methods allow us to answer research questions that we derived 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:

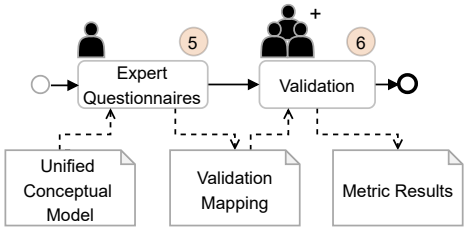


Fig. 5: Validation process of the unified conceptual model.

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.

Research Questions. Based on the two properties, we defined two 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.

Process. In Figure 5, we display our process for validating the unified conceptual model, which was consecutive to the construction process we presented in Figure 3. Step ⑤ represents a qualitative analysis based on expert questionnaires, which we explain in Section 7.1. Step ⑥ involves a quantitative analysis based on metrics, as we describe in Section 7.2.

7.1 Qualitative Analysis

We performed a qualitative analysis based on questionnaires completed by tool experts (cf. Step ⑤ in Figure 5) to map constructs, relations, and well-formedness rules of their tools to the concepts, relations, and well-formedness rules of the unified conceptual model.

Expert Questionnaire. Since all tool experts were familiar with the mapping procedure after our interviews (cf. Section 5.2), we refrained from employing explicit interviews again. Instead, we provided questionnaires for mapping tools to the conceptual model. Each questionnaire comprised three parts and was structured similarly to the interview guide. The first part introduced the unified conceptual model and definitions of the involved concepts and relations. The second part asked 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 the name of each tool construct. The third part listed and explained the well-formedness rules and asked for each rule whether it is satisfied by construction, enforced, evaluated, not covered, or not applicable.

Validation Mapping. As one aspect, our validation mappings covered the terminology used in the tools and the unified conceptual model. To obtain the mappings, we first had to identify relevant constructs in each tool and obtain an understanding of their semantics. We created each mapping based on the semantic equivalence of model concepts and tool

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 (*FT*)	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 (*P*)	Product	Variant	Product	Working Copy	Working Copy	Variant	Product	Product	Product	Product
Unified System (*US*)	Product Line	Product Line	Product Line	Repository	Repository	Repository	Repository	Product Line	Product Line	System
Mapping (*M*)	Mapping[1]	Restriction	Application Condition	Tree Object	Tree Node	Association	Mapping[1]	Mapping Model	Mapping Model	Mapping[1]
Feature (*F*)	Feature	Feature	Feature	—	—	Feature	Feature	Feature	Feature	Variant
System Revision (*SR*)	—	—	—	Commit	Revision	—	Revision	—	Temporal Validity	—
Feature Revision (*FR*)	—	—	—	—	—	Revision	—	Version	—	Version
Configuration (*C*)	Variant, Configuration	Configuration	Configuration	Commit	Revision	Configuration	Choice	Configuration	Configuration	Configuration[1]
Constraint (*CT*)	Constraint	Constraint, Relation	Constraint	—	—	—	Dependency	Constraint	Constraint	Constraint

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

constructs, not trivially based on name equivalence. This was necessary, since some tools use the same term for constructs that represent different concepts in the model. Note that we performed the mappings on the conceptual level of the tools (considering their semantics and expressiveness), and not on the implementation level. Moreover, we did not consider the abstract concepts (`Option` and `Revision`), since they cannot be instantiated.

Results. In Table 1, we show the mapping of concepts of the conceptual model to respective tool constructs. 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` and `Product Element`, respectively. All delta-oriented tools use the same types of `Fragments`: `Core Model` and `Delta Module`.

For some model concepts, the terms used for the respective tool constructs are almost uniform. Particularly for the concepts `Product` and `Configuration`, six and seven out of the ten tools use the same term. Still, three different terms are used for `Product` and five for `Configuration` across all tools. This shows that, even for the concepts with the highest consensus regarding terminology, there is still some variance.

In some cases, tools use the same term for constructs that represent different model concepts. For example, the construct `Variant` in VaVe maps to the concept `Feature`, the construct `Variant` in ECCO maps to the concept `Product`, and the concept `Variant` in FeatureIDE maps to the concept `Configuration`. This also shows that, even within the SPLE area, the same terms are used to represent different concepts.

Moreover, the mapping shows that the 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.

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

Relation \ Tool	FeatureIDE	pure::variants	SiPL	SVN	Git	ECCO	SuperMod	DeltaEcore	DarwinSPL	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.

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 since it maps only 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.

In Table 3, we show the mapping of the well-formedness rules of the conceptual model to each tool. We indicate whether a rule is satisfied by construction (■), enforced (●), evaluated but not enforced (◒), not evaluated (○), or does not apply (—). A rule is *satisfied by construction*, if a tool guarantees the respective well-formedness rule at any time and no check is necessary. For instance, the tools SVN and Git satisfy Rule 3 (all `Revisions` of a revision graph must be of the same type and have the same container) by construction, since they have only one type of revision (i.e., `System Revision`) and only one type of container (i.e., `Unified System` alias `Repository`). A rule is *enforced at all times*, if a tool guarantees the fulfillment of the respective well-formedness rule by means of checks. If a check detects a violation of the rule, the system either prohibits the change upfront or repairs its state. For instance, DeltaEcore satisfies Rule 4 (all `Options` in a `Configuration` must be contained in the `Unified System`) by evaluating and enforcing the well-formedness of a configuration upon saving it. Another example is ECCO, which adds any not yet existing `Feature Option` of the `Configuration` to the repository instead of prohibiting it. A rule is *evaluated but not enforced*, if a tool checks whether the rule is satisfied, but does not take

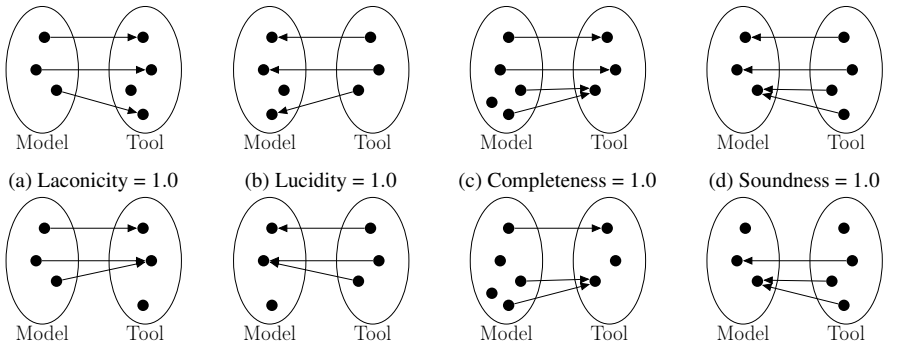
Table 3: *Validation Mapping:* Results of mapping the well-formedness rules of the conceptual model to each tool.

Rule \ Tool	FeatureIDE	pure::variants	SiPL	SVN	Git	ECCO	SuperMod	DeltaEcore	DarwinSPL	VaVe
Rule 1	—	—	—	■	■	—	■	■	■	■
Rule 2	—	—	—	■	■	—	■	■	■	■
Rule 3	—	—	—	■	■	—	■	■	■	■
Rule 4	■	●	■	●	●	●	■	●	●	■
Rule 5	○	◒	■	●	●	●	■	●	●	■
Rule 6	●	●	■	—	—	—	■	●	●	○
Rule 7	—	—	—	—	—	—	■	—	●	—
Rule 8	—	—	—	—	—	—	■	—	●	—
Rule 9	—	—	—	—	—	—	—	—	—	—
Rule 10	—	—	—	—	—	—	■	—	■	—

■ The rule is satisfied by construction. ● The rule is enforced at all times. ◒ The rule is evaluated, but not enforced. ○ The rule is neither evaluated nor enforced. — The rule does not apply.

or require any immediate action in case it is not. For instance, pure::variants evaluates Rule 5 (all `Fragments` and `Options` in a `Mapping` must be contained in the `Unified System`), but allows to import external `Fragments`. Furthermore, a rule can be *neither evaluated nor enforced*. For instance, the tool FeatureIDE neither checks nor enforces Rule 5. In the case of an annotation-based composer, there can be feature annotations in the source code that do not appear in the feature model. Finally, a rule *does not apply*, if concepts or relations it refers to do not exist in a tool. For instance, for FeatureIDE, pure::variants, and SiPL, Rules 1–3 and 7–10 do not apply, since these tools deal with variability in space only.

We can see in Table 3 that tools differ substantially with regard to the well-formedness rules. In particular, SuperMod satisfies most of the rules by construction, due to its metamodel and the product-wise editing process: First, SuperMod maintains a sequence of revisions and not a revision graph, allowing only up to one predecessor or successor per revision. A new revision receives a unique revision number and is added to its direct predecessor, which ensures Rule 1 and Rule 2. Second, SuperMod consists of `System Revisions` only, because explicit `Feature Revisions` are not modeled, which partially ensures Rule 3. Third, a repository does not allow mixing its contained constructs with another repository, since the local workspace can be populated with one product of one repository only, which ensures Rules 3–7. Lastly, the development process ensures that a `System Revision` is selected first, followed by selecting `Features` that are visible in this revision. In addition, `Mappings` for each `Fragment` in the repository are computed and updated upon each commit to the local workspace, consequently satisfying Rule 8 and Rule 10. DarwinSPL, which also uses `System Revisions`, achieves similar mapping results. Still, in contrast to SuperMod, it does enforce most rules. Furthermore, we observe for all tools that either all or none of the three rules regarding the well-formedness of the revision graph are satisfied (i.e., Rules 1–3). This could be due to the fact that internal operations satisfy these rules by construction and are immutable for users. Finally, Rule 9 is not satisfied by any of the selected tools. This is because no tool implements both `Feature Revisions` and `System Revisions`.



(a) Laconicity = 1.0 (b) Lucidity = 1.0 (c) Completeness = 1.0 (d) Soundness = 1.0

(e) Laconicity = $\frac{2}{3}$ = 0.67 (f) Lucidity = $\frac{2}{3}$ = 0.67 (g) Completeness = $\frac{2}{3}$ = 0.67 (h) Soundness = $\frac{2}{3}$ = 0.67

Fig. 6: Overview of the metrics we adapted from Guizzardi et al. (2005). *Model* refers to the conceptual model and *tool* refers to a tool's model.

7.2 Quantitative Analysis

We performed a quantitative analysis using metrics (cf. Step ⑥ in Figure 5) to quantify how well the conceptual model fits the selected tools based on the validation mapping.

Metrics. We adapted the *framework for language evaluation* proposed by Guizzardi et al. (2005) that introduces the properties *laconic*, *lucid*, *complete*, and *sound*. For our validation, we extended these properties to metrics ranging from 0 to 1 to measure 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). In Figure 6, we provide a graphical overview of the four metrics. 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* $m \in M$. For our conceptual model with the concepts `Fragment` (FT), `Product` (P), `Unified System` (US), `Mapping` (M), `Feature` (F), `System Revision` (SR), `Feature Revision` (FR), `Configuration` (C), and `Constraint` (CT), we have

$$M = \{FT, P, US, M, F, SR, FR, C, CT\}$$

A tool $T \in \mathcal{T}$ is a set of *tool constructs* $t \in T$. For simplicity, we consider relations as concepts and constructs, too. $\mathbb{R}_T^M \subseteq M \times T$ denotes the set of *mappings* of concepts in M to constructs in T, which we show in Table 1 and Table 2.

A tool's construct t is *laconic*, iff it implements at most one concept m of the conceptual model M. *Laconicity* $\in [0..1]$ (higher is better) is then the fraction of *laconic* tool constructs. Low laconicity indicates that concepts of the conceptual model may be too fine-grained, i.e., there are redundant concepts in the model that should be merged. In Figure 6a, all four tool constructs are laconic, leading to a laconicity of 1. In Figure 6e, only two out of three tool constructs are laconic, leading to a laconicity of 0.67:

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

$$\text{laconicity}(M,T) = \frac{\sum_{t \in T} \text{laconic}(M,T,t)}{|T|}$$

As an example, consider the tool Git, which implements six constructs, yielding the set $T_{\text{Git}} = \{\text{Blob}, \text{Tree Object}, \text{Working Copy}, \text{Repository}, \text{Tree Node}, \text{Revision}\}$. According to the concept mapping in Table 1, two model concepts (`System Revision` and `Configuration`) are implemented by the same construct in Git (`Revision`). The model concepts that do not map to any construct in Git (i.e., `Feature`, `Feature Revision`, and `Constraint`) do not affect the metric. The laconicity for Git (only considering concepts and not relations) is thus fairly high, albeit not perfect:

$$\text{laconicity}_{\text{Git}}(M, T_{\text{Git}}) = \frac{1+1+1+1+1+0}{6} = \frac{5}{6} = 0.833$$

A model's concept m is *lucid*, iff it is implemented by at most one construct t of a tool T. *Lucidity* $\in [0..1]$ (higher is better) is then the fraction of *lucid* model concepts. Low lucidity indicates that concepts of the conceptual model may be too coarse-grained, meaning that there are unspecific concepts in the model that should be split up. In Figure 6b, all four model concepts are lucid, leading to a lucidity of 1. In Figure 6f, only two out of three model concepts are lucid, leading to a lucidity of 0.67:

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

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

As an example, consider again Git. The model concept `Fragment` is implemented by two constructs in Git (`Blob` and `Tree Object`). All other model concepts are either implemented by exactly one tool construct or by no tool construct. The lucidity of the model with respect to Git (considering only concepts, not relations) is also fairly high, but not perfect:

$$\text{lucidity}_{\text{Git}}(M, T_{\text{Git}}) = \frac{0+1+1+1+1+1+1+1+1}{9} = \frac{8}{9} = 0.889$$

A tool's construct t is *complete*, iff it is represented by at least one concept m in the conceptual model M. *Completeness* $\in [0..1]$ (higher is better) is then the fraction of *complete* tool constructs. Low completeness indicates that the conceptual model may be missing concepts that should be added. In Figure 6c, all three tool constructs are complete, leading to a completeness of 1. In Figure 6g, only two out of three tool constructs are complete, leading to a completeness of 0.67:

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

$$\text{completeness}(M, T) = \frac{\sum_{t \in T} \text{complete}(M, T, t)}{|T|}$$

In the example of Git, according to the concept mapping in Table 1, there are no constructs in Git that do not map to any model concept. All six of its constructs can be mapped to at least one model concept, namely to the concepts `Fragment`, `Product`, `Unified System`, `Mapping`, `System Revision`, and `Configuration`. The completeness for Git (considering only concepts) is therefore ideal:

$$\text{completeness}_{\text{Git}}(M, T_{\text{Git}}) = \frac{1+1+1+1+1+1}{6} = \frac{6}{6} = 1.0$$

A model's concept m is *sound*, iff it is implemented by at least one construct t in the tool T. *Soundness* $\in [0..1]$ (higher is better) is then the fraction of *sound* model concepts. Low soundness indicates that the conceptual model may include unused concepts that should be removed. In Figure 6d, all three model concepts are sound, leading to a soundness of 1. In Figure 6h, only two out of three model concepts are sound, leading to a soundness of 0.67:

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

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

Regarding Git, out of the nine model concepts, only six (i.e., `Fragment`, `Product`, `Unified System`, `Mapping`, `System Revision`, `Configuration`) are implemented by at least one construct in Git according to the concept mapping in Table 1. Thus, the soundness of the model with respect to Git (considering only concepts) is fairly low:

$$\text{soundness}_{\text{Git}}(M,T_{\text{Git}}) = \frac{1+1+1+1+0+1+0+1+0}{9} = \frac{6}{9} = 0.667$$

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 our 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.[3] 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 4, we display the values for the four metrics (columns) per tool (rows), separated by concepts and relations that belong to space, time, both, and the unified dimension. 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 lucidity, which is represented by the two constructs `Blob` and `Tree Object`. The soundness values are generally lower, because no tool implements *all* concepts and relations.

[3] 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 $S \cap T = \varnothing$).

In Table 5, 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) and 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 in the aggregation, because every concept of the model is implemented by at least one construct in at least one tool.

7.3 Formal Concept Analysis

We performed a FCA (Ganter and Wille 1999; Ganter et al. 2005) to further explore and broaden our understanding of commonalities and differences between tools that deal with variability in space, time, and both. FCA is an algebraic theory for data analysis that defines a hierarchical relationship in the form of a *concept lattice* based on objects and their attributes specified in an *input matrix*. To perform the FCA, we used the same data as for computing the metrics, specifically the mappings between model concepts and tool constructs, and model relations and tool relations we described in Section 7.1. While the metrics are a quantitative representation of how each tool relates to the conceptual model, the FCA provides a graphical representation of how the tools relate to each other *in addition* to how each tool relates to the conceptual model. Consequently, the visualization of the FCA provides a comprehensible overview of commonalities and differences between the tools.

In Figure 7, we show the concept lattice between the objects (i.e., tools) and attributes (i.e., concepts of the conceptual model). Each node represents a pair of a set of tools and a set of concepts. If the upper semicircle of a node is filled blue, there is at least one concept attached to this node. If the lower semicircle of a node is filled black, there is at least one tool attached to this node. We highlight all edges based on the coloring of concepts in the conceptual model, namely concepts for variability in space (green), concepts for variability in time (orange), concepts for variability in both dimensions (purple), and unified concepts (black instead of white). Reading the concept lattice from top to bottom, edges that lead to nodes that have a colored concept attached have the same color. Edges that leave a node that has both green and orange edges as input are colored in purple (i.e., orange and green merge to purple). Likewise, edges that have at least one purple edge as input are colored in purple (i.e., purple always remains purple). We can see that nodes and edges on the left represent variability in time, those in the middle represent variability in space, and those at the bottom and on the right represent variability in space and time. Initially, nodes for space and time remain separate until they eventually merge when they get closer to the conceptual model.

The top node of the lattice represents the concepts that are common to all tools (i.e., `Unified System`, `Configuration`, `Fragment`, `Mapping`, and `Product`). The bottom node of the lattice represents the conceptual model. From the concept lattice, we can learn two things: First, how closely tools are related to the conceptual model and which concepts differ. Second, we can see how closely tools are related to each other. We can see that there is no tool involving all concepts of the conceptual model. The four closest tools to the conceptual model are DarwinSPL, SuperMod, VaVe, and DeltaEcore. Additionally, the tools are grouped according to the variability dimensions they deal with, namely variability

Table 4: *Metric Results* for each tool individually.

	for	laconicity	lucidity	completeness	soundness
FeatureIDE	*Space*	100% (5/5)	100% (5/5)	100% (5/5)	100% (5/5)
	Time	– (0/0)	100% (3/3)	– (0/0)	0% (0/3)
	Both	– (0/0)	100% (4/4)	– (0/0)	0% (0/4)
	Unified	100% (12/12)	100% (12/12)	100% (12/12)	100% (12/12)
	Total	100% (17/17)	100% (24/24)	100% (17/17)	71% (17/24)
pure:: variants	*Space*	100% (5/5)	100% (5/5)	100% (5/5)	100% (5/5)
	Time	– (0/0)	100% (3/3)	– (0/0)	0% (0/3)
	Both	– (0/0)	100% (4/4)	– (0/0)	0% (0/4)
	Unified	100% (12/12)	100% (12/12)	100% (12/12)	100% (12/12)
	Total	100% (17/17)	100% (24/24)	100% (17/17)	71% (17/24)
SiPL	*Space*	100% (5/5)	100% (5/5)	100% (5/5)	100% (5/5)
	Time	– (0/0)	100% (3/3)	– (0/0)	0% (0/3)
	Both	– (0/0)	100% (4/4)	– (0/0)	0% (0/4)
	Unified	100% (11/11)	92% (11/12)	100% (11/11)	100% (12/12)
	Total	100% (16/16)	96% (23/24)	100% (16/16)	71% (17/24)
SVN	*Space*	– (0/0)	100% (5/5)	– (0/0)	0% (0/5)
	Time	100% (3/3)	100% (3/3)	100% (3/3)	100% (3/3)
	Both	– (0/0)	100% (4/4)	– (0/0)	0% (0/4)
	Unified	92% (11/12)	92% (11/12)	100% (12/12)	100% (12/12)
	Total	93% (14/15)	96% (23/24)	100% (15/15)	63% (15/24)
Git	*Space*	– (0/0)	100% (5/5)	– (0/0)	0% (0/5)
	Time	100% (3/3)	100% (3/3)	100% (3/3)	100% (3/3)
	Both	– (0/0)	100% (4/4)	– (0/0)	0% (0/4)
	Unified	92% (12/13)	92% (11/12)	92% (12/13)	100% (12/12)
	Total	94% (15/16)	96% (23/24)	94% (15/16)	63% (15/24)
ECCO	*Space*	100% (2/2)	100% (5/5)	100% (2/2)	40% (2/5)
	Time	– (0/0)	100% (3/3)	– (0/0)	0% (0/3)
	Both	100% (2/2)	100% (4/4)	100% (2/2)	50% (2/4)
	Unified	100% (12/12)	100% (12/12)	92% (11/12)	100% (12/12)
	Total	100% (16/16)	100% (24/24)	94% (15/16)	67% (16/24)
SuperMod	*Space*	100% (5/5)	100% (5/5)	100% (5/5)	100% (5/5)
	Time	100% (3/3)	100% (3/3)	100% (3/3)	100% (3/3)
	Both	100% (2/2)	100% (4/4)	100% (2/2)	50% (2/4)
	Unified	100% (11/11)	100% (12/12)	100% (11/11)	100% (12/12)
	Total	100% (21/21)	100% (24/24)	100% (21/21)	92% (22/24)
DeltaEcore	*Space*	100% (5/5)	100% (5/5)	100% (5/5)	100% (5/5)
	Time	100% (1/1)	100% (3/3)	100% (1/1)	0% (0/3)
	Both	100% (2/2)	100% (4/4)	100% (2/2)	50% (2/4)
	Unified	100% (12/12)	92% (11/12)	100% (12/12)	100% (12/12)
	Total	100% (20/20)	96% (23/24)	100% (20/20)	79% (19/24)
DarwinSPL	*Space*	100% (5/5)	100% (5/5)	100% (5/5)	100% (5/5)
	Time	100% (3/3)	100% (3/3)	100% (3/3)	100% (3/3)
	Both	100% (2/2)	100% (4/4)	100% (2/2)	50% (2/4)
	Unified	100% (12/12)	92% (11/12)	100% (12/12)	100% (12/12)
	Total	100% (22/22)	96% (23/24)	100% (22/22)	92% (22/24)
VaVe	*Space*	100% (5/5)	100% (5/5)	100% (5/5)	100% (5/5)
	Time	100% (1/1)	100% (3/3)	100% (1/1)	0% (0/3)
	Both	100% (2/2)	100% (4/4)	100% (2/2)	50% (2/4)
	Unified	100% (11/11)	92% (11/12)	100% (11/11)	100% (12/12)
	Total	100% (19/19)	96% (23/24)	100% (19/19)	79% (19/24)

Table 5: *Metric Results* over all tools.

Kind	for	laconicity	lucidity	completeness	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)

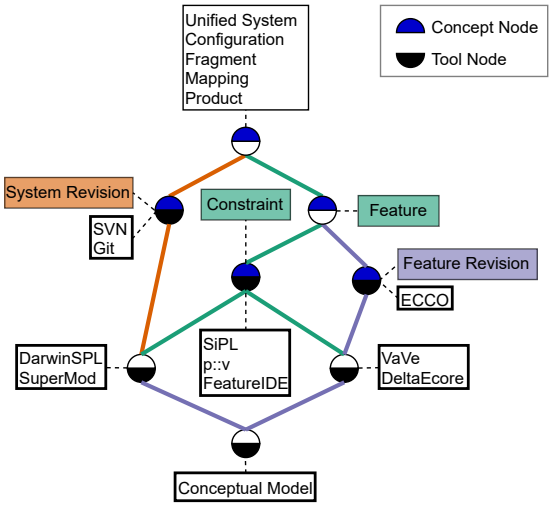


Fig. 7: FCA of tools based on conceptual model concepts.

in time (i.e., SVN, Git), variability in space (i.e., SiPL, pure::variants, FeatureIDE), and both. In the last case, tools are grouped based on whether they use `System Revisions` (i.e., DarwinSPL, SuperMod) or `Feature Revisions` (i.e., VaVe, DeltaEcore). ECCO is an exception, because it is the only tool that deals with variability in space and time, but has no `Constraints`. While we could derive these observations from the mapping in Table 1 and metric results in Table 4, the concept lattice provides a comprehensive overview of the commonalities and differences between tools and the conceptual model.

In Figure 8, we display the concept lattice between the tools and the concepts and relations of the conceptual model (as opposed to Figure 7, where we considered only concepts as attributes). To reduce the visual overhead, we omit the concept labels. This representation allows to further distinguish tools that do not differ regarding the concepts used, but that

differ with respect to the relations they employ. In summary, there are six relations that are common to all tools. Nodes and edges on the top left represent unified concepts and relations, while nodes and edges on the bottom right combine variability in space and time. In between, nodes and edges for variability in space and time remain separate until they eventually merge as they get closer to the conceptual model.

We can see that DarwinSPL and DeltaEcore are actually closest to the conceptual model. Furthermore, all tools that involve `Features`, `Constraints`, and `System Revisions` also have relations where `System Revisions` enable `Constraints` and `Feature Options`. Consequently, tools that deal only with variability in time, such as SVN and GIT, involve `System Revisions`, but no `Features` or `Constraints`—which is why *enable*-relations do not exist in these tools.

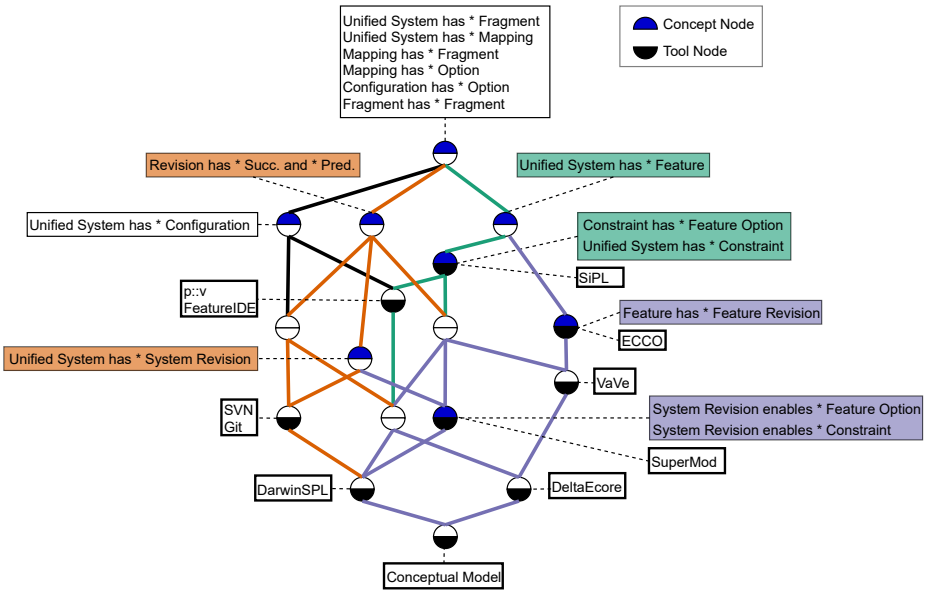


Fig. 8: FCA of tools based on conceptual model concepts and relations.

7.4 Discussion

In the following, we discuss the validation results based on 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 `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 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, since we intended 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 can be split up (i.e., made more specific) without becoming too tool-specific (e.g., splitting `Fragment` into `Blob` and `Tree Object`), and thus leading to worse values for other metrics.

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 is highlighted by the aggregated values in Table 5, 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. In fact, the addition of a `Unified System refers to * Unified System` relation is the only remaining change to the model that would yield an overall improvement in metric values.

7.5 Threats to Validity

One threat to the construct validity is the level of abstraction at which we mapped tool constructs 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 the 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 the construct and external validity is whether the selected tools are representative for both, SPLE and SCM. We argue that our tool selection covers a representative body of existing tools from both areas. 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.

A potential threat to the internal validity is that some tool experts are authors of this article, which could introduce bias towards their tools. However, involving experts is a recommendation for building conceptual models (Ahlemann and Riempp 2008). We aimed

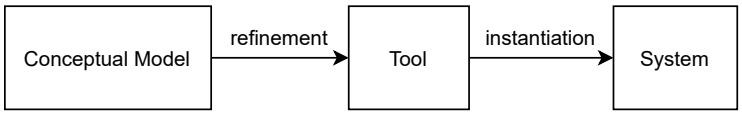


Fig. 9: Application stages of the conceptual model.

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. To enable other researchers to check our results and derive their own conclusions, we publish our data in an open-access repository.[1]

8 Applying the Unified Conceptual Model in Practice

In this section, we demonstrate how to apply the conceptual model in practice, for example, to develop a new tool. For this purpose, we introduce illustrative applications of the conceptual model. Initially, we explain how we envision the conceptual model to be used when designing and implementing a conforming tool. Then, we exemplify two tool implementations based on the conceptual model and explain their construction process. The first example illustrates an application of the conceptual model using `Feature Revisions` and a feature model. In contrast, the second example illustrates a design choice that cannot be found in any of our studied tools, which is the combination of the concepts `System Revision` and `Feature Revision`. Finally, we demonstrate a validation of the two exemplary tools by applying the same metrics we used to validate the unified model to assess their conformance to that model.

We show the two stages of applying the conceptual model in practice in Figure 9. The first stage is the refinement of the model into a conforming tool using concrete constructs based on the (abstract) model concepts. This task is performed by tool developers. The second stage is the instantiation of the tool for a specific variable system. This task is performed by users of the developed tool and happens implicitly by applying the tool during the development of a system out in the field. We illustrate the first step using UML class diagrams and the second step via UML object diagrams for each of the two exemplary tools.

8.1 Refinement Process of the Conceptual Model

When applying the conceptual model in practice to develop a conforming tool, its non-abstract concepts need to be extended by creating concrete subclasses. For example, for tools that use feature models to model constraints, the `Constraint` concept may be refined by creating multiple concrete subclasses to represent mandatory or optional children. Another example is `Fragment`, which could be refined into two concrete subclasses, `Core Model` and `Delta Module`, for tools that use deltas. Abstract concepts in the model are not intended to be subclassed in tools. We define the following degrees of freedom (design choices) that developers have to decide on when refining the model:

Revision. Developers may use `Feature Revisions`, `System Revisions`, or both in combination. The concept `Revision` must be refined accordingly.

Constraint. This concept can be refined either by adding attributes or by extending it in subclasses to express more sophisticated constraints between `Features`, such as mandatory, optional, or cross-tree constraints in a feature model.

Fragment. Identically, `Fragments` can be refined either by adding attributes or by extending them in subclasses to express more specific types. As we described in Section 2.2, the derivation process of a `Product` is based on a specific variability mechanism. This also defines the type of `Fragments` that can be used, for instance, `Core Model` and `Delta Modules` are needed for current delta-oriented (transformational) variability mechanisms.

Mapping. Developers can refine this concept to express differently complex relations between `Options` and `Fragments`, for example, using a simple list or Boolean expressions.

Configuration. Refining this concept is optional. It can be refined similar to `Mapping`, but it can also be used directly as it is in the conceptual model.

In the following, we demonstrate two exemplary tool refinements. For each, we describe the decisions for the degrees of freedom, compute the validation metrics, and show an instantiation for the PPU example.

8.2 Feature-Revision / Transformational Tool T_T

We created a tool T_T, which incorporates feature revisions for a transformational variability mechanism, based on the conceptual model by extending and refining its concepts as described above. In Figure 10, we display the tool's conceptual model.

Construction Process. We depict concepts and relations that are identical to the conceptual model as they are within that model. In contrast, we highlight added concepts (i.e., subclasses of concepts in the conceptual model) with a hatched area (i.e., `Change`, `DeltaModule`, `Expression`, `Cross-tree Constraint`, and `Tree Constraint`). Moreover, we display unused concepts of the conceptual model in red (i.e., `System Revision`). To derive the tool's model, we incorporated the following design decisions:

Revision: Feature Revisions. For this example, we use `Feature Revisions` as the only type of revision. We chose not to use `System Revisions`, because this can be achieved by combining the tool with any tool that supports `System Revisions`, such as Git or SVN. Also, we chose not to combine both types of revisions to reduce complexity.

Constraint: Feature Model. We use feature modeling to express `Constraints`, as represented by the new tool constructs `Tree Constraint` and `Cross-tree Constraint`; both subclasses of the model concept `Constraint`. `Tree Constraints` in the feature model can only refer to `Features` and not to `Feature Revisions`, while `Cross-tree Constraints` can refer to both, `Features` and `Feature Revisions`. To manage the Boolean expressions of `Cross-tree Constraints`, we introduce a new construct `Expression`. Specifically, the `Expression` would be represented as a tree in which inner nodes align to operators and leafs to `Feature Options`. For space reasons, we omit these details in the tool's model.

Fragment: Delta Module, Change. We use deltas to implement `Fragments` and compose `Products`. This choice is represented by the new constructs `Delta Module` and `Change`. A delta module comprises changes, which can be either additive or subtractive, respectively adding or deleting a `Fragment` (given as *value*) at a certain position (given as a *path*). In this example, we use a String value for textual `Fragments`, such as the source code in our PPU example. Still, the value could also represent any non-textual `Fragment`.

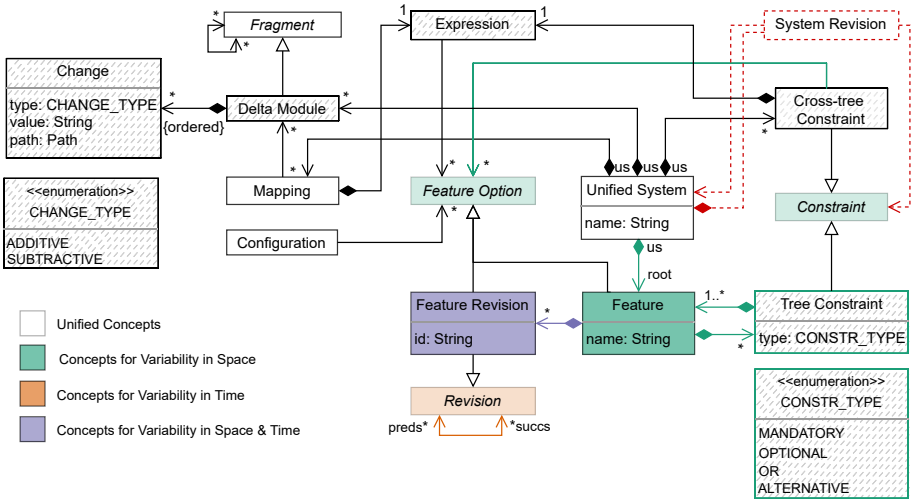


Fig. 10: Feature-revision / transformational tool model T_T.

Mapping: Boolean Expression. For this example, we represent the relation between a `Mapping` and `Options` as a Boolean expression.

Configuration: No Containment in Unified System. We made the deliberate choice to have the `Configuration` *not* contained in the `Unified System`. This corresponds to non-persistent `Configurations`.

Validation. For the validation of T_T, we apply the same metrics we used in Section 7.2. Note that we do not treat enumeration types as concepts, and thus ignore them when computing the metrics. Overall, the tool T_T implements ten non-abstract constructs, yielding the set $T_T = \{$ `Unified System`, `Feature`, `Tree Constraint`, `Cross-tree Constraint`, `Feature Revision`, `Configuration`, `Mapping`, `Expression`, `Delta Module`, `Change` $\}$. No construct in T_T implements more than one concept of the conceptual model. The constructs that do not implement any model concept do not affect laconicity. The laconicity for T_T (only considering concepts and not relations) thus is:

$$\text{laconicity}_{T_T}(M, T_T) = \frac{10}{10} = 1.0$$

The model concept `Constraint` is implemented by two constructs in T_T (i.e., `Tree Constraint` and `Cross-tree Constraint`). All other model concepts are either implemented by exactly one tool construct or by no tool construct. Thus, the lucidity for T_T (only considering concepts and not relations) is:

$$\text{lucidity}_{T_T}(M, T_T) = \frac{8}{9} = 0.889$$

The two constructs `Expression` and `Change` in T_T do not implement any model concept. In contrast, the remaining eight constructs map to at least one model concept. Note that the construct `Delta Module` is a specialization of the concept `Fragment`, which itself has become abstract. Consequently, `Delta Module` can be considered to map to `Fragment`. The

completeness for the tool T_T (considering only concepts) is:

$$\text{completeness}_{T_T}(M, T_{T_T}) = \frac{8}{10} = 0.846$$

Out of the nine model concepts, only seven map to at least one construct in T_T (i.e., `System Revision` and `Product` are not implemented by T_T). The soundness of the model with respect to the tool T_T (considering only concepts) thus is:

$$\text{soundness}_{T_T}(M, T_{T_T}) = \frac{7}{9} = 0.889$$

In summary, the tool T_T refines and splits some model concepts to make them more concrete (lower lucidity), adds some additional constructs (lower completeness), and does not make use of all model concepts (lower soundness).

Instantiation. We show an instance of the tool T_T for a small part of the PPU example in the form of an object diagram in Figure 11. It consists of one instance of the `Unified System` with the name *PPU*. The root feature of the feature model (named *PPU*) is the only feature directly contained in the `Unified System`. Additional feature instances with the names *Crane* and *Stack* are children of the *PPU* root feature based on two `Tree Constraint` instances of type *mandatory*. Feature instances with the names *MicroSwitch* and *InductiveSensor* represent children of the *Crane* feature through a `Tree Constraint` instance of type *alternative*. Finally, the feature instance *OpticalSensor* represents an optional child (via a `Tree Constraint` instance of type *optional*) of feature *Stack*. While the features *PPU*, *Crane*, *Stack*, *MicroSwitch*, and *InductiveSensor* each are available in one `Feature Revision`, the feature *OpticalSensor* is available in two `Feature Revisions`. The revision with the identifier 1 is the first revision and is succeeded in the revision graph by the revision with the identifier 2. Furthermore, the `Unified System` contains one `Cross-tree Constraint` comprising the expression $\neg OpticalSensor \vee \neg InductiveSensor$.

The `Mapping` with the identifier 1 contains the `Expression` instance $MicroSwitch.1 \wedge \neg InductiveSensor$, which refers to revision 1 of feature *MicroSwitch* and to any revision of feature *InductiveSensor*. A respective `Delta Module` instance comprises two `Change` instances of type *additive*. `Change` c1 adds the line `MicroSwitch ms;` to Line 3 of the file `Crane.java`. `Change` c2 adds the line `Crane(MicroSwitch ms)` to Line 4 of the file `Crane.java`. Finally, there is one `Configuration` instance referring to revision 1 of feature *PPU*, revision 1 of feature *Crane*, and revision 1 of feature *Stack*.

8.3 Both-Revisions / Compositional Tool T_C

In contrast to T_T, we now discuss another tool T_C based on quite different design decisions. Specifically, this tool employs both `System Revisions` and `Feature Revisions` in combination. It follows a compositional variability mechanism to derive `Products` from `Fragments`, and uses Boolean expressions for mapping `Fragments` to `Options` as well as for formulating `Constraints`. We aimed to keep T_C as minimalistic and as close to the conceptual model as possible. Therefore, we employ as many concepts directly from the model as possible, without modifying, adding, or deleting concepts and relations. In Figure 12, we display the tool's model.

Construction Process. We added the constructs `Mapping Expression` and `Constraint Expression` to represent the relations between `Mapping` and `Options` as well as between `Constraint` and `Feature Options`, respectively. Additionally, we distinguish between

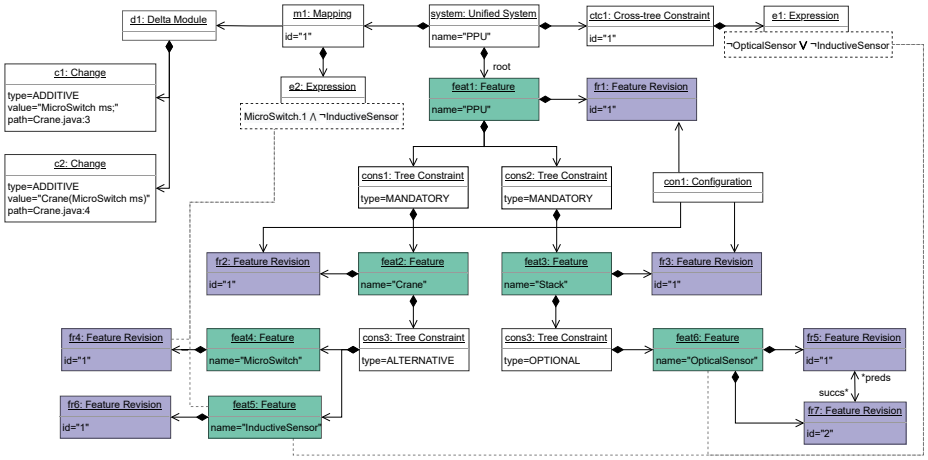


Fig. 11: Object diagram of tool T_T applied to the PPU example.

selected and deselected `Options` in `Configurations`. Furthermore, we added attributes, such as *value*, *name*, or *id* to some concepts. Finally, `Fragments` are contained directly in `Products` (instead of the indirect derivation of the `Product` from `Fragments`). The Boolean expressions would actually be expression trees with inner nodes representing Boolean operations (e.g., *and*, *or*, *negation*) and leafs representing literals (`Options` for `Mappings` and `Feature Options` for `Constraints`). However, we depict this as a single concept (i.e., `Expression`) for the sake of simplicity. T_C is based on the following design decisions:

Revision: Feature Revisions and System Revisions. For this example, we aimed to use `System Revisions` and `Feature Revisions` in combination. None of the tools we analyzed has attempted this. Arguably, combining both types of revisions is not as simple as using either concept independently, and would bring only little benefit while complicating workflows for users. To tackle this problem, we aimed to use `System Revisions` so that they support the user in managing `Feature Revisions`. When using only `Feature Revisions`, there is no automated mechanism for tracking which revisions of which `Features` can be combined, for example, is revision 3 of feature *Crane* compatible with revision 3 or revision 4 of feature *Stack*? Instead, the user would have to manually track such dependencies, for example, by manually specifying respective `Constraints`. Considering CVS and SVN, this resembles a similar problem CVS encountered: Revisions are tracked per file, making it difficult to track which revisions of which files can be combined to form a valid configuration. While file revisions are not the same as `Feature Revisions` (i.e., files are part of the solution space, whereas `Features` are part of the problem space), the underlying problem is the same. CVS exemplifies that disconnected (file) revisions are problematic, because they are not connected via a (global) `System Revision`. SVN avoids this problem by not using (file) revisions at all. Instead, it uses only `System Revisions`, also on files. Consequently, this leads to files having gaps in their sequence of revision numbers (e.g., 1, 2, 5, 7), since they are mapped to `System Revisions`. By combining `System Revisions` and `Feature Revisions` in this tool instance of the conceptual model, we aim to combine the advantages of both concepts.

Constraint: Boolean Expression. For the construct `Constraint`, we use simple Boolean expressions with `Feature Options` (i.e., `Features` and `Feature Revisions`) as liter-

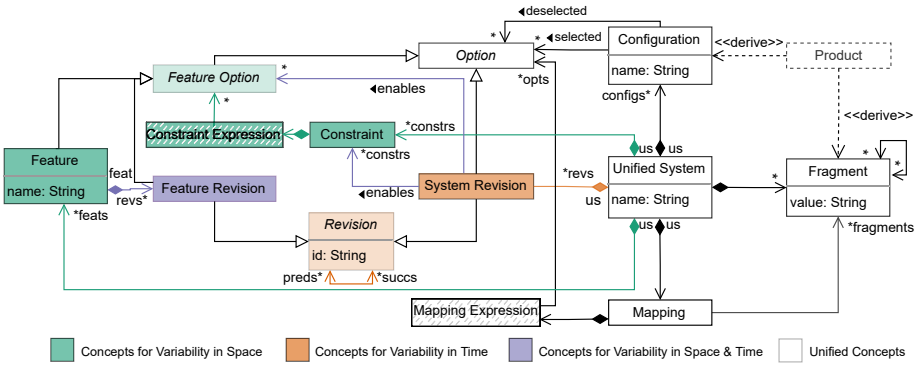


Fig. 12: Both-Revisions / Compositional tool model T_C.

als. Consequently, we represent the relation between `Constraint` and `Feature Option` with the new construct `Constraint Expression`.

Fragment: Implementation Artifacts. We chose `Fragments` to directly represent implementation artifacts (such as lines of source code). Such `Fragments` are either present or missing from a `Product`. The order in which they are added to a `Product` does not affect the resulting `Product`. This is in contrast to a delta-oriented mechanism (as implemented by T_T), where `Fragments` (i.e., deltas) themselves are not a direct part of a `Product`. Instead, deltas represent operations that need to be executed in a particular order to incrementally construct a `Product`. In that case, `Fragments` are a set of (ordered) instructions for building a `Product` rather than implementation artifacts that are part of a `Product`.

Mapping: Boolean Expression. We represent the relation between `Mapping` and `Options` with a Boolean expression in the form of the new construct `Mapping Expression`.

Configuration: Selected and Deselected Options. For the relation of `Configuration` and `Options`, we distinguish explicitly *selected* and *deselected* `Options`, and `Options` for which no choice has been made, yet. This allows to express *partial* `Configurations`.

Validation. We again use the metrics from Section 7.2 to validate T_C. Overall, T_C implements eleven non-abstract constructs, yielding the set of constructs T_C = { `Unified System`, `Feature`, `Feature Revision`, `System Revision`, `Constraint`, `Constraint Expression`, `Configuration`, `Mapping`, `Mapping Expression`, `Fragment`, `Product` }. No construct in T_C implements more than one concept of the conceptual model. The laconicity for T_C (considering only concepts) thus is:

$$\text{laconicity}_{T_C}(M, T_{T_C}) = \frac{11}{11} = 1.0$$

No model concept is implemented by more than one construct in T_C. Consequently, the lucidity for T_C (considering only concepts) is:

$$\text{lucidity}_{T_C}(M, T_{T_C}) = \frac{9}{9} = 1.0$$

The two constructs `Constraint Expression` and `Mapping Expression` in T_C do not implement any model concept. All remaining nine constructs map to at least one model concept. Therefore, the completeness for the tool T_C (considering only concepts) is:

$$\text{completeness}_{T_C}(M, T_{T_C}) = \frac{9}{11} = 0.818$$

All nine model concepts are implemented by at least one construct in T_C. As a result, the soundness of the model with respect to the tool T_C (considering only concepts) is:

$$\text{soundness}_{T_T}(M, T_{T_T}) = \frac{9}{9} = 1.0$$

In summary, T_C aligns very well to the conceptual model. Solely the addition of the construct `Expression` for formulating mappings and constraints causes lower completeness.

Instantiation. We display the instantiation of the tool T_C for a small part of the PPU as an object diagram in Figure 13. In the center, an instance of the `Unified System` with the name *PPU* is located. Beneath, T_C contains the features *PPU*, *Crane*, *Stack*, *MicroSwitch*, *OpticalSensor*, and *InductiveSensor* that, in turn, contain instances of their `Feature Revisions`. Additionally, the `Unified System` itself contains two `System Revisions`. `System Revision` 1 enables `Feature Revisions` 1 of features *PPU*, *Crane*, *Stack*, *MicroSwitch*, *OpticalSensor*, and *InductiveSensor*, expressing that these six revisions together form a valid state of the PPU. Combinations of other features or revisions may result in an inconsistent state of the PPU. `System Revision` 2 enables `Feature Revisions` 1 of features *PPU*, *Crane*, *Stack*, *MicroSwitch*, *InductiveSensor*, and `Feature Revision` 2 of feature *OpticalSensor*. On its right, the `Unified System` contains two instances of `Constraint`. The first `Constraint` c1 refers to the feature *PPU* via its contained `Constraint Expression` *PPU* and indicates that PPU is the root feature. The second `Constraint` c2 refers to features *PPU* and *Crane* via the `Constraint Expression` *PPU* $\Leftrightarrow$ *Crane* and expresses that both features *PPU* and *Crane* are always present simultaneously in a `Product`. Since the PPU is always present (according to c1), *Crane* is a mandatory feature. The `Constraint Expressions` are depicted in a simplified manner as formulas instead of expression trees.

The `Unified System` contains two `Fragment` instances, each representing a line of code given by their respective *value* attribute. `Mapping` connects the `Feature Revision` *MicroSwitch.1* and the feature *InductiveSensor* through its `Expression` MicroSwitch.1 $\wedge$ $\neg$InductiveSensor with both `Fragments`. The two `Fragments` are contained in any `Product` with a `Configuration` that satisfies the `Mapping` expression. In fact, the `Configuration` instance with the name *Customer1* explicitly selects `Feature Revisions` *PPU.1*, *Crane.1*, *Stack.1*, *MicroSwitch.1*, *OpticalSensor.2* (indicated by the "+" symbol), and explicitly deselects the feature *InductiveSensor* (indicated by the "-" symbol)—yielding a `Product` that contains both `Fragments`.

8.4 Summary

We demonstrated how the conceptual model can be applied in practice by refining it into two exemplary tools. Based on the particular type of tool, for instance, whether a tool supports variability in space or time, the conceptual model is open for modifications via refinements in two ways. First, by using only the concepts of the respective variability dimension that shall be managed by a tool. Second, existing concepts can be specialized (e.g., `Fragments` are specialized by `Delta Modules` in the first exemplary tool). We also demonstrated how the metrics we introduced in Section 7.2 can be used to validate and compare novel tools against the unified conceptual model, which indicates the conformance of a tool to the conceptual model. Furthermore, the metrics can also be used to compare two tools with each other in the same way, thereby providing a means to compare tools based on the conceptual model.

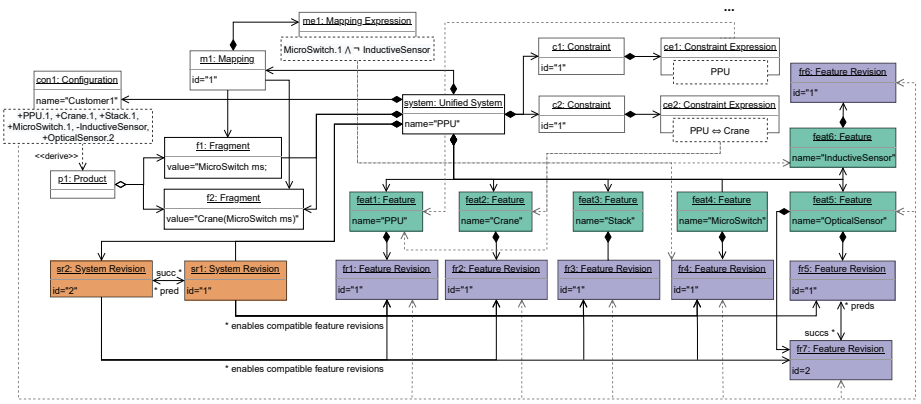


Fig. 13: Object diagram of tool T_C applied to the PPU example.

9 Conclusion

Most of today's systems are highly configurable and evolve rapidly, challenging a uniform management of variability in space and time. In this article, we extended our previous conference paper (Ananieva et al. 2020) in which we proposed a conceptual model for unfiying variability in space and time. Besides additional details on the construction process, design decisions, and validation, we also contributed static semantics, illustrative applications, analyses, and showed how to apply the model when developing conforming tools. The conceptual model achieves high coverage and appropriate granularity regarding established concepts of SPLE and SCM. We showed that the model can provide guidance for researchers and developers intending to work on the combination of these research areas, for instance, for assessing the conformance of a new tool to the respective dimensions of variability. As a consequence, the conceptual model fills a gap that is increasingly subject to new research and tools, providing a means to compare works, identify gaps, and support communication.

In future work, we intend to work on formalizing the operations of the conceptual model, helping tool developers understand, implement, and validate those. Moreover, it would be interesting to investigate to what extent combinations of existing tools conform to the conceptual model and apply the conceptual model and/or potential conforming tools to a set of real-world case studies. Finally, we showed that most tools have an individual combination of concepts and relations, which asks for more research considering which combinations are used for what purpose, and which of the not implemented combinations could provide benefits beyond the current state-of-the-art in both research areas.

Acknowledgements

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

References

Ahlemann F, Riempp G (2008) Refmod^{pm}: A conceptual reference model for project management information systems. Wirtschaftsinformatik 50(2), DOI 10.1365/s11576-008-0028-y

Ananieva S, Klare H, Burger E, Reussner R (2018) Variants and versions management for models with integrated consistency preservation. In: International Workshop on Variability Modelling of Software-Intensive Systems, ACM, VaMoS, DOI 10.1145/3168365.3168377

Ananieva S, Berger T, Burger A, Kehrer T, Klare H, Koziolek A, Lönn H, Ramesh S, Taentzer G, Westfechtel B (2019a) Conceptual Modeling Group. In: Berger T, Chechik M, Kehrer T, Wimmer M (eds) Software Evolution in Time and Space: Unifying Version and Variability Management (Dagstuhl Seminar 19191), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, DOI 10.4230/DagRep.9.5.1

Ananieva S, Kehrer T, Klare H, Koziolek A, Lönn H, Ramesh S, Burger A, Taentzer G, Westfechtel B (2019b) Towards a conceptual model for unifying variability in space and time. In: International Systems and Software Product Line Conference, ACM, SPLC, DOI 10.1145/3307630.3342412

Ananieva S, Greiner S, Kühn T, Krüger J, Linsbauer L, Grüner S, Kehrer T, Klare H, Koziolek A, Lönn H, Krieter S, Seidl C, Ramesh S, Reussner R, Westfechtel B (2020) A conceptual model for unifying variability in space and time. In: International Systems and Software Product Line Conference, ACM, pp 15:1–12

Apel S, Kästner C (2009) An Overview of Feature-Oriented Software Development. Journal of Object Technology 8(5):49–48, DOI 10.5381/jot.2009.8.5.c5

Apel S, Janda F, Trujillo S, Kästner C (2009a) Model superimposition in software product lines. In: International Conference on Theory and Practice of Model Transformations, Springer, ICMT, DOI 10.1007/978-3-642-02408-5_2

Apel S, Kästner C, Lengauer C (2009b) FEATUREHOUSE: language-independent, automated software composition. In: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings, IEEE, pp 221–231, DOI 10.1109/ICSE.2009.5070523, URL `https://doi.org/10.1109/ICSE.2009.5070523`

Apel S, Batory D, Kästner C, Saake G (2013) Feature-Oriented Software Product Lines. Springer, DOI 10.1007/978-3-642-37521-7

Asikainen T, Männistö T, Soininen T (2006) A unified conceptual foundation for feature modelling. In: International Software Product Line Conference, IEEE, SPLC, DOI 10.1109/SPLINE.2006.1691575

Bashroush R, Garba M, Rabiser R, Groher I, Botterweck G (2017) Case tool support for variability management in software product lines. ACM Computing Surveys 50(1), DOI 10.1145/3034827

Batory D (2005) Feature models, grammars, and propositional formulas. In: International Conference on Software Product Lines, Springer, SPLC, DOI 10.1007/11554844_3

Beek MHt, Schmid K, Eichelberger H (2019) Textual variability modeling languages: An overview and considerations. In: International Systems and Software Product Line Conference, ACM, SPLC, DOI 10.1145/3307630.3342398

Berger T, Chechik M, Kehrer T, Wimmer M (2019) Software Evolution in Time and Space: Unifying Version and Variability Management (Dagstuhl Seminar 19191). Dagstuhl Reports, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, DOI 10.4230/DagRep.9.5.1

Beuche D (2013) pure::variants. In: Capilla R, Bosch J, Kang KC (eds) Systems and Software Variability Management - Concepts, Tools and Experiences, Springer, DOI 10.1007/978-3-642-36583-6_12

Bosch J (2010) Toward compositional software product lines. IEEE Software 27(3), DOI 10.1109/MS.2010.32

Clements P, Northrop L (2001) Software Product Lines: Practices and Patterns. Addison-Wesley

Conradi R, Westfechtel B (1998) Version models for software configuration management. ACM Computing Surveys 30(2), DOI 10.1145/280277.280280

Czarnecki K, Hwan C, Kim P, Kalleberg K (2006) Feature models are views on ontologies. In: International Software Product Line Conference, IEEE, SPLC, DOI 10.1109/SPLINE.2006.1691576

Czarnecki K, Grünbacher P, Rabiser R, Schmid K, Wąsowski A (2012) Cool features and tough decisions: A comparison of variability modeling approaches. In: International Workshop on Variability Modelling of Software-Intensive Systems, ACM, VaMoS, DOI 10.1145/2110147.2110167

Dintzner N, van Deursen A, Pinzger M (2016) Fever: Extracting feature-oriented changes from commits. In: Proceedings of the 13th International Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR '16, p 85–96, DOI 10.1145/2901739.2901755, URL https://doi.org/10.1145/2901739.2901755

Dubinsky Y, Rubin J, Berger T, Duszynski S, Becker M, Czarnecki K (2013) An exploratory study of cloning in industrial software product lines. In: European Conference on Software Maintenance and Reengineering, IEEE, CSMR, DOI 10.1109/CSMR.2013.13

Estublier J (2000) Software configuration management: A roadmap. In: Conference on the Future of Software Engineering, ACM, FOSE, DOI 10.1145/336512.336576

Fischer S, Linsbauer L, Lopez-Herrejon RE, Egyed A (2014) Enhancing clone-and-own with systematic reuse for developing software variants. In: International Conference on Software Maintenance and Evolution, IEEE, ICSME, DOI 10.1109/icsme.2014.61

Fischer S, Linsbauer L, Lopez-Herrejon RE, Egyed A (2015) The ecco tool: Extraction and composition for clone-and-own. In: International Conference on Software Engineering, IEEE, ICSE, DOI 10.1109/ICSE.2015.218

Gacek C, Anastasopoules M (2001) Implementing product line variabilities. In: Symposium on Software Reusability, ACM, SSR, pp 109–117, DOI 10.1145/375212.375269

Galster M, Weyns D, Tofan D, Michalik B, Avgeriou P (2014) Variability in software systems—a systematic literature review. IEEE Transactions on Software Engineering 40(3), DOI 10.1109/TSE.2013.56

Gamez N, Fuentes L (2011) Software product line evolution with cardinality-based feature models. In: Schmid K (ed) Top Productivity through Software Reuse, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 102–118

Ganter B, Wille R (1999) Formal Concept Anlaysis – Mathematical Foundations. Springer

Ganter B, Stumme G, Wille R (eds) (2005) Formal Concept Analysis, Foundations and Applications, Lecture Notes in Computer Science, vol 3626, Springer

Gheyi R, Massoni T, Borba P (2008) Algebraic laws for feature models. Journal of Universal Computer Science 14(21)

Guizzardi G, Pires LF, van Sinderen M (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, Springer, MODELS, DOI 10.1007/11557432_51

Horcas JM, Pinto M, Fuentes L (2019) Software product line engineering: A practical experience. In: International Systems and Software Product Line Conference, ACM, SPLC, DOI 10.1145/3336294.3336304

Johansen MF, Fleurey F, Acher M, Collet P, Lahire P (2010) Exploring the synergies between feature models and ontologies. In: International Conference on Software Product Lines, SPLC

Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS (1990) Feature-oriented domain analysis (foda) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Carnegie-Mellon University

Kang KC, Kim S, Lee J, Kim K, Shin E, Huh M (1998) FORM: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering 5:143–168, DOI 10.1023/A:1018980625587, URL https://doi.org/10.1023/A:1018980625587

Kehrer T, Kelter U, Taentzer G (2013) Consistency-preserving edit scripts in model versioning. In: International Conference on Automated Software Engineering, IEEE, ASE, DOI 10.1109/ASE.2013.6693079

Kehrer T, Kelter U, Taentzer G (2014) Propagation of software model changes in the context of industrial plant automation. at-Automatisierungstechnik 62(11):803–814

Kehrer T, Thüm T, Schultheiß A, Bittner P (2021) Bridging the gap between clone-and-own and software product lines. In: ICSE-NIER 2021: 43rd International Conference on Software Engineering, New Ideas and Emerging Results

Klare H, Kramer ME, Langhammer M, Werle D, Burger E, Reussner R (2021) Enabling consistency in view-based system development – The Vitruvius approach. Journal of Systems and Software 171

Kramer ME, Burger E, Langhammer M (2013) View-centric engineering with synchronized heterogeneous models. In: International Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling, ACM, VAO, DOI 10.1145/2489861.2489864

Kröher C, Gerling L, Schmid K (2018) Identifying the intensity of variability changes in software product line evolution. In: Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1, Association for Computing Machinery, New York, NY, USA, SPLC '18, p 54–64, DOI 10.1145/3233027.3233032, URL https://doi.org/10.1145/3233027.3233032

Krueger C, Clements P (2012) Systems and software product line engineering with biglever software gears. In: Proceedings of the 16th International Software Product Line Conference - Volume 2, Association for Computing Machinery, New York, NY, USA, p 256–259, DOI 10.1145/2364412.2364458

Krüger J (2019) Are you talking about software product lines? an analysis of developer communities. In: International Workshop on Variability Modelling of Software-Intensive Systems, ACM, VaMoS, DOI 10.1145/3302333.3302348

Krüger J, Berger T (2020) An empirical analysis of the costs of clone- and platform-oriented software reuse. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2020, p 432–444, DOI 10.1145/3368089.3409684, URL https://doi.org/10.1145/3368089.3409684

Krüger J, Berger T (2020) An empirical analysis of the costs of clone- and platform-oriented software reuse. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, ESEC/FSE, pp 432–444, DOI 10.1145/3368089.3409684

Krüger J, Ananieva S, Gerling L, Walkingshaw E (2020) Third international workshop on variability and evolution of software-intensive systems (varivolution 2020). In: International Systems and Software Product Line Conference, ACM, SPLC, DOI 10.1145/3382025.3414944

Kästner C, Thüm T, Saake G, Feigenspan J, Leich T, Wielgorz F, Apel S (2009) Featureide: A tool framework for feature-oriented software development. In: International Conference on Software Engineering, IEEE, ICSE, DOI 10.1109/ICSE.2009.5070568

Linsbauer L, Egyed A, Lopez-Herrejon RE (2016) A variability aware configuration management and revision control platform. In: International Conference on Software Engineering, ACM, ICSE, DOI 10.1145/2889160.2889262

Linsbauer L, Berger T, Grünbacher P (2017a) A classification of variation control systems. In: International Conference on Generative Programming: Concepts & Experience, ACM, GPCE, DOI 10.1145/3136040.3136054

Linsbauer L, Lopez-Herrejon RE, Egyed A (2017b) Variability extraction and modeling for product variants. Software and Systems Modeling 16(4), DOI 10.1007/s10270-015-0512-y

Linsbauer L, Malakuti S, Sadovykh A, Schwägerl F (2018) 1st intl. workshop on variability and evolution of software-intensive systems (varivolution). In: International Systems and Software Product Line Conference, SPLC, DOI 10.1145/3233027.3241372

Linsbauer L, Schwägerl F, Berger T, Grünbacher P (2021) Concepts of variation control systems. J Syst Softw 171:110796, DOI 10.1016/j.jss.2020.110796, URL `https://doi.org/10.1016/j.jss.2020.110796`

Loeliger J, McCullough M (2012) Version Control with Git. O'Reilly

MacKay SA (1995) The state of the art in concurrent, distributed configuration management. In: International Workshop on Software Configuration Management, Springer, SCM, DOI 10.1007/3-540-60578-9_17

Meinicke J, Thüm T, Schröter R, Benduhn F, Leich T, Saake G (2017) Mastering Software Variability with FeatureIDE. Springer, DOI 10.1007/978-3-319-61443-4

Nešić D, Krüger J, Stănciulescu c, Berger T (2019) Principles of feature modeling. In: Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, ESEC/FSE, DOI 10.1145/3338906.3338974

Nieke M, Engel G, Seidl C (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, VaMoS, DOI 10.1145/3023956.3023962

Nieke M, Linsbauer L, Krüger J, Leich T (2019) Second international workshop on variability and evolution of software-intensive systems (varivolution 2019). In: International Systems and Software Product Line Conference, ACM, SPLC, DOI 10.1145/3336294.3342367

Northrop LM (2002) Sei's software product line tenets. IEEE Software 19(4), DOI 10.1109/ms.2002.1020285

Nunes C, Garcia A, Lucena C, Lee J (2012) History-sensitive heuristics for recovery of features in code of evolving program families. In: Proceedings of the 16th International Software Product Line Conference - Volume 1, Association for Computing Machinery, New York, NY, USA, SPLC '12, p 136–145, DOI 10.1145/2362536.2362556, URL `https://doi.org/10.1145/2362536.2362556`

Object Management Group (2014) Object Constraint Language

Parnas DL (1976) On the design and development of program families. IEEE Transactions on Software Engineering SE-2(1), DOI 10.1109/TSE.1976.233797

Passos L, Czarnecki K, Apel S, Wąsowski A, Kästner C, Guo J (2013) Feature-oriented software evolution. In: Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems, Association for Computing Machinery, New York, NY, USA, VaMoS '13, DOI 10.1145/2430502.2430526, URL `https://doi.org/10.1145/2430502.2430526`

Pereira JA, Constantino K, Figueiredo E (2015) A systematic literature review of software product line management tools. In: International Conference on Software Reuse, Springer,

ICSR, DOI 10.1007/978-3-319-14130-5_6
Pietsch C, Kehrer T, Kelter U, Reuling D, Ohrndorf M (2015) Sipl – a delta-based modeling framework for software product line engineering. In: International Conference on Automated Software Engineering, IEEE, ASE, DOI 10.1109/ASE.2015.106
Pietsch C, Reuling D, Kelter U, Kehrer T (2017) A tool environment for quality assurance of delta-oriented model-based spls. In: Proceedings of the Eleventh International Workshop on Variability Modelling of Software-Intensive Systems, ACM, p 84–91, DOI 10.1145/3023956.3023960
Pietsch C, Kelter U, Kehrer T, Seidl C (2019) Formal foundations for analyzing and refactoring delta-oriented model-based software product lines. In: International Systems and Software Product Line Conference, ACM, SPLC, DOI 10.1145/3336294.3336299
Pietsch C, Seidl C, Nieke M, Kehrer T (2020) Delta-oriented development of model-based software product lines with deltaecore and sipl: A comparison. In: Tekinerdogan B, Önder Babur, Cleophas L, van den Brand M, Akşit M (eds) Model Management and Analytics for Large Scale Systems, Elsevier, DOI 10.1016/B978-0-12-816649-9.00017-X
Pilato CM, Collins-Sussman B, Fitzpatrick BW (2008) Version Control with Subversion: Next Generation Open Source Version Control. O'Reilly
Pohl K, Böckle G, Linden FJvd (2005) Software Product Line Engineering. Springer, DOI 10.1007/3-540-28901-1
Rubin J, Chechik M (2013) A framework for managing cloned product variants. In: International Conference on Software Engineering, ICSE, DOI 10.1109/ICSE.2013.6606686
Ruparelia NB (2010) The history of version control. ACM SIGSOFT Software Engineering Notes 35(1):5–9, DOI 10.1145/1668862.1668876
Schaefer I, Bettini L, Bono V, Damiani F, Tanzarella N (2010) Delta-oriented programming of software product lines. In: International Conference on Software Product Lines, Springer, SPLC, DOI 10.1007/978-3-642-15579-6_6
Schaefer I, Rabiser R, Clarke D, Bettini L, Benavides D, Botterweck G, Pathak A, Trujillo S, Villela K (2012) Software diversity: State of the art and perspectives. International Journal on Software Tools for Technology Transfer 14(5), DOI 10.1007/s10009-012-0253-y
Schobbens PY, Heymans P, Trigaux JC, Bontemps Y (2007) Generic semantics of feature diagrams. Computer Networks 51(2), DOI 10.1016/j.comnet.2006.08.008
Schulze S, Schulze M, Ryssel U, Seidl C (2016) Aligning coevolving artifacts between software product lines and products. Association for Computing Machinery, New York, NY, USA, VaMoS '16, p 9–16, DOI 10.1145/2866614.2866616, URL `https://doi.org/10.1145/2866614.2866616`
Schwägerl F (2018) Version control and product lines in model-driven software engineering. PhD thesis, University of Bayreuth
Schwägerl F, Westfechtel B (2016) Supermod: Tool support for collaborative filtered model-driven software product line engineering. In: International Conference on Automated Software Engineering, ACM, ASE, DOI 10.1145/2970276.2970288
Schwägerl F, Westfechtel B (2019) Integrated revision and variation control for evolving model-driven software product lines. Software and Systems Modeling 18(6), DOI 10.1007/s10270-019-00722-3
Seidl C, Schaefer I, Aßmann U (2014a) Capturing variability in space and time with hyper feature models. In: International Workshop on Variability Modelling of Software-Intensive Systems, ACM, VaMoS, DOI 10.1145/2556624.2556625
Seidl C, Schaefer I, Aßmann U (2014b) Deltaecore - A model-based delta language generation framework. In: Modellierung, GI

Seidl C, Schaefer I, Aßmann U (2014c) Integrated management of variability in space and time in software families. In: International Software Product Line Conference, ACM, SPLC, DOI 10.1145/2648511.2648514

Strüber D, Mukelabai M, Krüger J, Fischer S, Linsbauer L, Martinez J, Berger T (2019) Facing the truth: Benchmarking the techniques for the evolution of variant-rich systems. In: International Systems and Software Product Line Conference, ACM, SPLC, DOI 10.1145/3336294.3336302

Stănciulescu c, Schulze S, Wąsowski A (2015) Forked and integrated variants in an open-source firmware project. In: International Conference on Software Maintenance and Evolution, IEEE, ICSME, DOI 10.1109/icsm.2015.7332461

Svahnberg M, van Gurp J, Bosch J (2005) A taxonomy of variability realization techniques. Software: Practice and Experience 35(8):705–754, DOI 10.1002/spe.652

Thüm T, Teixeira L, Schmid K, Walkingshaw E, Mukelabai M, Varshosaz M, Botterweck G, Schaefer I, Kehrer T (2019) Towards efficient analysis of variation in time and space. In: International Software Product Line Conference, ACM, SPLC, DOI 10.1145/3307630.3342414

Vogel-Heuser B, Legat C, Folmer J, Feldmann S (2014) Researching evolution in industrial plant automation: Scenarios and documentation of the pick and place unit. Tech. Rep. TUM-AIS-TR-01-14-02, Technical University of Munich

Westfechtel B, Munch BP, Conradi R (2001) A layered architecture for uniform version management. IEEE Transactions on Software Engineering 27(12), DOI 10.1109/32.988710

Sofia Ananieva is a doctoral researcher at the chair for Software Design and Quality (SDQ) at Karlsruhe Institute of Technology (KIT) since 2016, employed by the FZI Research Center for Information Technology. Her research focuses on a uniform management of variability in space and time for highly configurable systems, with particular interests in developing model-driven tools and approaches that allow engineering of software product lines from different engineering view points while ensuring incremental consistency propagation between views.

Sandra Greiner received her M.Sc. degree in Computer Science from the University of Bayreuth in 2015 and commenced her PhD. studies at the Chair of Software Engineering, University of Bayreuth. In 2021, she visited the Software Quality and Research Grout at IT University of Copenhagen, Denmark. Her research interests are dedicated to the maintenance of evolving software systems, particularly those developed in a model-driven way. One main focus lies on employing automated techniques in model-driven software product lines to guarantee a consistent evolution of their artifacts.

Timo Kehrer is Professor at Humboldt-Universität zu Berlin (Germany), heading the Model-Driven Software Engineering Group at the Department of Computer Science. Before that, Kehrer was working as research assistant in the Software Engineering and Database Systems Group at University of Siegen (Germany) from 2010 to 2015, and as postdoctoral research fellow in the Dependable Evolvable Pervasive Software Engineering Group at Politecnico di Milano (Italy) from 2015 to 2016. He has active research interests in various fields of model-based software and system engineering, with a particular focus on software and systems evolution.

Jacob Krüger is associated researcher at the Software Engineering group of the Ruhr-University Bochum, and obtained his PhD degree in 2021 at the Otto-von-Guericke University Magdeburg, Germany. He worked as research associate at the Otto-von-Guericke University Magdeburg as well as the Harz University of Applied Sciences Wernigerode, and visited Chalmers | University of Gothenburg in Sweden as well as the University of Toronto in Canada. His research addresses feature-oriented software development, with particular focus on software evolution, program comprehension, and human factors.

Thomas Kühn is a post-doc at the Software Design and Quality Group at Karlsruhe Institute of Technology. His research focuses one new ways to model and program future software systems challenged by increased complexity, heterogeneity, rate of change and longevity. As a result, he developed a family of role-based modelling and a family of role-oriented programming languages supported by a feature-aware modelling editor and a basic IDE, respectively. Currently, he improves tool support for view-based, model-driven software development building on the Vitruvius approach. Contact him at thomas.kuehn@kit.edu.

Lukas Linsbauer is currently a postdoctoral researcher at the Institute of Software Engineering and Automotive Informatics at the Technische Universität Braunschweig in Germany. He received his Doctorate in 2016 from the Institute for Software Systems Engineering at the Johannes Kepler University Linz in Austria under the supervision of Prof. Alexander Egyed and Dr. Roberto Erick Lopez-Herrejon. His research interests include highly variable and configurable systems, software product lines, feature-oriented software and systems development, traceability, and version control systems.

Anne Koziolek is a full professor of software engineering at Karlsruhe Institute of Technology (KIT), Germany. She received her PhD degree from KIT in 2011 and was a Postdoc at University of Zurich until 2013. Her current research interest is how to reconcile agile, code-centric software development with model-based software engineering, especially regarding models for quality prediction as well as design models, including those with information on variability.

Henrik Lönn has a PhD in Computer Engineering from Chalmers University of Technology, Sweden, with a research focus on safety-critical real-time systems. At Volvo, he has worked on various aspects on vehicle electronic systems including architecture modelling,system integration and V&V. He is also participating in national and international research collaborations on embedded systems development. Previous project involvement includes X-by-Wire, FIT, EAST-EEA, ATESST, TIMMO and MAENAD as well as Swedish projects like Synligare, HeavyRoad and EMISYS.

Ramesh S is a Senior Technical Fellow at General Motors Global R&D, in Warren, MI, US, where he provides technical leadership in R&D. His areas of interests include rigorous modeling, verification and validation of software and systems for automotive embedded control. Before moving to USA, he managed a research group that looked into rigorous verification and validation of automotive control software at the GM India Science Labs in Bangalore. Earlier, he was on the faculty of the Department of Computer Science at the Indian Institute of Technology Bombay India as a Professor, for more than fifteen years. At IIT Bombay, he played a major role in setting up a National Centre for Formal Design and Verification of Software. He has published more than 125 research papers and has more than 10 patents in the area of software engineering and verification.

Ralf Reussner is a computer science professor at Karlsruhe Institute of Technology (KIT). He holds the chair for Software Design and Quality (SDQ) since 2006 and heads the Institute for Program Structures and Data Organization. His research group works in the interplay of software architecture and predictable software quality as well as on view-based design methods for software-intensive technical systems.