The Karlsruhe Series on
Software Design
and Quality
37

Consistent View-Based Management of Variability in Space and Time

Sofia Ananieva

VAVE
VITRUVIUS

KIT Scientific Publishing

Sofia Ananieva

Consistent View-Based Management of Variability in Space and Time

The Karlsruhe Series on Software Design and Quality
Volume 37

Dependability of Software-intensive Systems group
Faculty of Computer Science
Karlsruhe Institute of Technology

and

Software Engineering Division
Research Center for Information Technology (FZI), Karlsruhe

Editor: Prof. Dr. Ralf Reussner

Consistent View-Based Management of Variability in Space and Time

by
Sofia Ananieva

Scientific Publishing

Karlsruher Institut für Technologie
KASTEL – Institut für Informationssicherheit und Verlässlichkeit

Consistent View-Based Management of Variability in Space and Time

Zur Erlangung des akademischen Grades einer Doktorin der Ingenieurwissenschaften von der KIT-Fakultät für Informatik des Karlsruher Instituts für Technologie (KIT) genehmigte Dissertation

von Sofia Ananieva

Tag der mündlichen Prüfung: 1. Juli 2022
Erster Gutachter: Prof. Dr. Ralf H. Reussner
Zweite Gutachterin: Prof. Dr. Ina Schaefer

Impressum

KIT Scientific Publishing

Karlsruher Institut für Technologie (KIT)
KIT Scientific Publishing
Straße am Forum 2
D-76131 Karlsruhe

KIT Scientific Publishing is a registered trademark
of Karlsruhe Institute of Technology.
Reprint using the book cover is not allowed.

www.ksp.kit.edu

Print on Demand 2022 – Gedruckt auf FSC-zertifiziertem Papier

ISSN 1867-0067
ISBN 978-3-7315-1241-7
DOI: 10.5445/KSP/1000151349

To Valentina Ananieva, Mark Khaitan and Sören Langhorst

Abstract

Systems evolve rapidly and exist in many variations to address different and changing requirements. This leads to subsequent revisions (variability in time) and concurrent product variants (variability in space). Redundancies and dependencies between different products across various revisions on the one hand and heterogeneous types of artifacts on the other hand quickly lead to inconsistencies during the evolution of a variable system. Handling the complexity while managing both variability dimensions uniformly and consistently is a major challenge when developing large and long-living variable systems. Variability in space is primarily considered in software product line engineering (SPLE), while variability in time is mainly addressed in software configuration management (SCM). Consistency preservation between heterogeneous artifact types and view-based software development are major research topics in model-driven software development (MDSD). The isolation of these three related engineering disciplines has led to a plethora of research, approaches and tools from each area, which impedes a common shared understanding and causes redundant research. Therefore, tools from these areas are usually not well integrated, leading to a heterogeneous tooling landscape and high manual effort for evolving a variable system, which, in turn, harms the system's quality and increases maintenance costs.

Based on the current state of the art in these three disciplines, this thesis presents three main contributions to cope with the complexity of evolving variable systems.
The **unified conceptual model** documents and unifies concepts and relations for simultaneously coping with variability in space and time based on a diverse set of analyzed tools and approaches from SPLE and SCM. Beyond their mere combination, the unified conceptual model proposes novel ways of relating both variability dimensions.
The **unified operations** form the basis for operational management of variability in space and time using the unified conceptual model as data structure. Based on an analysis of diverse contemporary tools that follow different

modalities and paradigms, the unified operations provide functionality beyond the state of the art by preserving each tool's functionality while extending it to uniformly cope with both variability dimensions.
The **unified approach** refines the unified conceptual model and unified operations and additionally integrates consistency preservation mechanisms. To this end, different types of variability-related inconsistency types have been identified that can occur during the evolution of variable systems comprised of heterogeneous artifacts. The unified approach integrates automated consistency preservation for a selected subset of the identified inconsistency types to support consistent unified management of variable systems.

Every main contribution of this thesis has been empirically evaluated. The unified conceptual model and unified operations have been evaluated based on expert surveys, devised metrics to assess the appropriateness of a unification, and exemplary applications. Moreover, the functional suitability of the unified approach has been evaluated by applying it to two real-world case studies: the well-known ArgoUML-SPL data set that has been extracted from a snapshot of ArgoUML, a UML modeling tool, and MobileMedia, a mobile application for media management. The unified approach is implemented using the Eclipse Modeling Framework (EMF) and the Vitruvius approach.

With the presented contributions, this thesis broadens the body of knowledge on unified management of variability in space and time and bridges it with automated consistency preservation across heterogeneous artifact types.

Zusammenfassung

Systeme entwickeln sich schnell weiter und existieren in verschiedenen Variationen, um unterschiedliche und sich ändernde Anforderungen erfüllen zu können. Das führt zu aufeinanderfolgenden Revisionen (Variabilität in Zeit) und zeitgleich existierenden Produktvarianten (Variabilität in Raum). Redundanzen und Abhängigkeiten zwischen unterschiedlichen Produkten über mehrere Revisionen hinweg sowie heterogene Typen von Artefakten führen schnell zu Inkonsistenzen während der Evolution eines variablen Systems. Die Bewältigung der Komplexität sowie eine einheitliche und konsistente Verwaltung beider Variabilitätsdimensionen sind wesentliche Herausforderungen, um große und langlebige Systeme erfolgreich entwickeln zu können. Variabilität in Raum wird primär in der Softwareproduktlinienentwicklung betrachtet, während Variabilität in Zeit im Softwarekonfigurationsmanagement untersucht wird. Konsistenzerhaltung zwischen heterogenen Artefakttypen und sichtbasierte Softwareentwicklung sind zentrale Forschungsthemen in modellgetriebener Softwareentwicklung. Die Isolation der drei angrenzenden Disziplinen hat zu einer Vielzahl von Ansätzen und Werkzeugen aus den unterschiedlichen Bereichen geführt, was die Definition eines gemeinsamen Verständnisses erschwert und die Gefahr redundanter Forschung und Entwicklung birgt. Werkzeuge aus den verschiedenen Disziplinen sind oftmals nicht ausreichend integriert und führen zu einer heterogenen Werkzeuglandschaft sowie hohem manuellen Aufwand während der Evolution eines variablen Systems, was wiederum der Systemqualität schadet und zu höheren Wartungskosten führt.

Basierend auf dem aktuellen Stand der Forschung in den genannten Disziplinen werden in dieser Dissertation drei Kernbeiträge vorgestellt, um den Umgang mit der Komplexität während der Evolution variabler Systeme zu unterstützen.
Das **unifizierte konzeptionelle Modell** dokumentiert und unifiziert Konzepte und Relationen für den gleichzeitigen Umgang mit Variabilität in Raum und Zeit basierend auf einer Vielzahl ausgewählter Ansätze und Werkzeuge

aus der Softwareproduktlinienentwicklung und dem Softwarekonfigurationsmanagement. Über die bloße Kombination vorhandener Konzepte hinaus beschreibt das unifizierte konzeptionelle Modell neue Möglichkeiten, beide Variabilitätsdimensionen zueinander in Beziehung zu setzen.
Die **unifizierten Operationen** verwenden das unifizierte konzeptionelle Modell als Datenstruktur und stellen die Basis für operative Verwaltung von Variabilität in Raum und Zeit dar. Die unifizierten Operationen werden basierend auf einer Analyse diverser Ansätze konzipiert, welche verschiedene Modalitäten und Paradigmen verfolgen. Während die unifizierten Operationen die Funktionalität von analysierten Werkzeugen abdecken, ermöglichen sie den gleichzeitigen Umgang mit beiden Variabilitätsdimensionen.
Der **unifizierte Ansatz** basiert auf den vorhergehenden Beiträgen und erweitert diese um Konsistenzerhaltung. Zu diesem Zweck wurden Typen von variabilitätsspezifischen Inkonsistenzen identifiziert, die während der Evolution variabler heterogener Systeme auftreten können. Der unifizierte Ansatz ermöglicht automatisierte Konsistenzerhaltung für eine ausgewählte Teilmenge der identifizierten Inkonsistenztypen.

Jeder Kernbeitrag wurde empirisch evaluiert. Zur Evaluierung des unifizierten konzeptionellen Modells und der unifizierten Operationen wurden Expertenbefragungen durchgeführt, Metriken zur Bewertung der Angemessenheit einer Unifizierung definiert und angewendet, sowie beispielhafte Anwendungen demonstriert. Die funktionale Eignung des unifizierten Ansatzes wurde mittels zweier Realweltfallstudien evaluiert: Die häufig verwendete ArgoUML-SPL, die auf ArgoUML basiert, einem UML-Modellierungswerkzeug, sowie MobileMedia, eine mobile Applikation für Medienverwaltung. Der unifizierte Ansatz ist mit dem Eclipse Modeling Framework (EMF) und dem Vitruvius Ansatz implementiert.

Die Kernbeiträge dieser Arbeit erweitern das vorhandene Wissen hinsichtlich der uniformen Verwaltung von Variabilität in Raum und Zeit und verbinden diese mit automatisierter Konsistenzerhaltung für variable Systeme bestehend aus heterogenen Artefakttypen.

Acknowledgments

My deepest thanks go to my supervisor, Ralf Reussner, for giving me the opportunity to be part of his group. He envisioned my research, believed in me and guided me through this journey. I am thankful for his confidence in me, the scientific freedom I enjoyed when conducting my work, and for his lasting support in every way.
I also want to thank Ina Schaefer, my co-supervisor. She sparked my interest in research and software product lines and gave me the opportunity to be part of her institute at the TU Braunschweig during my time as a student there. The people I closely worked with during this time, in particular Thomas Thüm and Christoph Seidl, greatly inspired me and strengthened my decision to pursue the PhD.
I am grateful to my advisors, Erik Burger, who started this adventure with me, and Thomas Kühn, who helped me reach the finish line. I highly appreciated the exciting discussions, fruitful ideas, care and positivity throughout the years. Additionally, I thank Anne Koziolek for her guidance, sympathy and valuable feedback.
Moreover, I am thankful to the colleagues of my research community who worked with me on the unification of variability management. Particularly, I want to thank Lukas Linsbauer, who first supported me as colleague and later as partner. In our busy life, he inspired me, grounded me and always found ways to express his love.
Gratitude is owed to all my current and former colleagues at FZI Karlsruhe, FZI Berlin and KIT-DSiS. Special thanks go to the FZI SE department and particularly to Jörg Henß, for being an amazing division manager, for always having my back, and for believing in me. In addition, I want to thank FZI Berlin for welcoming me so warmly. Special thanks go to Judith Junker and Angelika Kaplan, in whom I have found wonderful friends.
Without all the great people who accompanied me in the last years, this thesis would not have been possible.
Most of all, I am sincerely grateful to my family. Everything I am and everything I have achieved so far I owe to them.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Part I.

Preliminaries

1. Introduction

This chapter builds on publications at VariVolution [9], SPLC [3, 7], Empirical Software Engineering [5], and VaMoS [10, 6].

The development and evolution of software-intensive systems is a difficult endeavor that faces many challenges [163]. Among others, these challenges involve a high diversity of artifacts, great degree of variation, and frequent evolution. In other words, engineers need to deal with *variable*, *evolving* systems composed of *heterogeneous* artifacts.

Most modern software-intensive systems exist in different *variations* to be able to offer tailored customization. For example, to fulfill varying customer requirements, regulations or hardware limitations. In the literature, concurrent variations (i.e., products) of a system at one point in time are commonly referred to as *variability in space* and are primarily addressed in the research area of software product line engineering (SPLE) [191, 12, 178, 53, 105, 206, 51]. A product line promotes the usage of a shared set of artifacts in all its products – thereby enabling a higher degree of efficiency compared to traditional software development approaches. *Features*, i.e., distinguishing characteristics of a product [191], are used to configure a custom-tailored product. For realizing a product based on a configuration, a variability mechanism must be employed, which assembles all implementation artifacts for building a particular product. The following three categories of variability mechanisms can be distinguished [12]: annotative (e.g., via conditional compilation with C/C++ preprocessor directives or in Java with the Antenna preprocessor [13]), compositional (e.g., with feature-oriented programming [28] or aspect-oriented programming [119]) and transformational (e.g., via delta modeling [205]). Over time, systems undergo various changes, such as changing customer requirements, bug fixes or refactorings, known as *software evolution* [134]. In the literature, sequential variations (i.e., revisions) of a system are commonly referred to as *variability in time* and are primarily addressed in the research area of software configuration management (SCM) [63, 48, 151, 190, 201].

A sub-discipline of SCM are version control systems (VCS) that are used to manage changes and control software evolution. Moreover, modern variable systems comprise not only one type of artifact, like source code, but different kinds of artifacts, such as UML diagrams or SysML models. Specifically, artifacts may originate from different engineering disciplines, such as electrical, mechanical or software engineering.

1.1. Problem Statements

Variability in space and time affect almost every software-intensive system nowadays. Nonetheless, the research areas of SPLE and SCM have developed largely independently of each other, and various techniques originating from one of the areas aim at coping with the respective other variability dimension. On the one hand, when developing a system with common VCSs (e.g., SVN [190] or Git [145]), variability is often managed in parallel development branches, employing one branch per product. Since the concept of a feature does not exist, merging between branches is required in order to maintain the products. Although this strategy is simple, it does not scale and leads to high manual effort [61, 198]. On the other hand, the missing proactive management of variability in time in SPLE has led to numerous approaches [59, 78, 124, 179, 182, 211], such as mining information (e.g., feature changes) from VCSs retroactively [59]. To this end, Krüger et al. [127] analyze the costs of development strategies for variable systems, underpinning that missing explicit tracking of feature evolution incurs high additional costs. Nonetheless, combining existing approaches that cope with variability in space and time does not suffice. A heterogeneous tooling landscape (e.g., a VCS for managing the evolution of a system, a product-line management tool for managing its products, and a mining tool for recovering feature evolution information) requires developers to often switch context along with manual effort, which is error-prone and increases maintenance costs. Moreover, cross-dimensional variability modeling and analyses, such as modeling feature revisions and analyzing their frequency of change, are not supported. As a consequence, due to the lack of methodology and common understanding for effective unified management, a plethora of approaches and tools has emerged that tackle the same problems independently [26, 137, 185, 201]. This not only leads to redundant research and development, but also hampers understanding how contemporary tools that cope with variability in space and time differ in detail as well as the design of novel effective techniques for unified variability management. Thus, the following first problem statement is derived:

P1 Lack of methodology and common understanding for uniform management of variability in space and time leads to redundant research and development, hampers the comparison of existing approaches as well as the design of new ones, and increases the complexity and manual effort for evolving a variable system, which is error-prone and incurs additional costs.

Besides uniformly and effectively dealing with variability in space and time, current practices in SPLE face several challenges [141, 137]. On the one hand, the development of variable systems by means of traditional variability mechanisms lacks automation and is a mostly manual task, as variation points (e.g., preprocessor directives) need to be added and updated manually. Furthermore, some variability mechanisms, such as aspect-oriented programming, are not commonly known and require expertise of developers to be able to effectively use them. Consequently, current practices for managing variable systems are cognitively complex [214, 212]. One the other hand, practices lack support for handling heterogeneous artifacts: The different categories of variability mechanisms (e.g., a transformational mechanism such as delta modeling) must be specialized for concrete types of artifacts to be applicable in practice (e.g., DeltaJ [119] as delta language for the Java programming language). In the worst case, this requires developers to employ as many variability mechanisms as there are different types of artifacts in their system, which demands expert knowledge and requires high manual effort. Thus, a high diversity of artifact types makes it increasingly difficult to realize variability. Moreover, evolving a variable system can lead to different types of variability-related inconsistencies (e.g., the consistent evolution of a variability model [176, 17, 100] or the consistent co-evolution of the variability model and configurations [78, 174]) that impair the system's quality and increase maintenance costs. Their detection and repair is an open research problem that has been tackled by numerous approaches with varying notions of consistency, leading to an unorganized research landscape which impedes communication and scoping of research [3]. In case of heterogeneous artifacts, additional manual effort is required to also keep variability consistent across the different types of artifacts, which is challenging and error-prone [64, 141]. For example, changing a feature in one artifact type of one product may impact other artifacts types of the same product as well as other products that must all be kept consistent. While automated consistency preservation between heterogeneous artifact types is a major research topic in model-driven software development (MDSD) [60, 62, 95, 116, 196, 242], these approaches have hardly been leveraged in SPLE research yet. From these observations, the second problem statement is derived:

P2 Current practices in variability management require high manual effort and expert knowledge while lacking support for heterogeneous artifact types when developing and evolving variable systems. This not only increases the complexity of managing such systems, but also the chance of variability-related inconsistencies, which both harm the system's quality and increase maintenance costs.

To summarize, dealing with systems that vary in *space* and *time* while being composed of *heterogeneous* artifacts is highly demanding and may lead to different variability-related inconsistency types. Every aspect is challenging to deal with on its own, yet various industrial branches demand their unified management [180, 68, 38, 74].

1.2. Research Goal and Questions

Based on the described problems, the research goal of this thesis is formulated and presented in the following.

Research Goal
Define a common foundation to deal with variability in space and time that addresses gaps in state of the art. Based on this foundation, conceive an approach that is capable of handling variability-related inconsistencies during the evolution of a system composed of heterogeneous artifacts. Ultimately, the goal is to enable unified consistent management of variable systems.

Achieving the research goal requires to deal with the two described problems. Referring to these problems, research questions are formulated which are introduced in the following.

The strong demand for coping with variability in space and time has led to various management techniques and contemporary approaches of both the SPLE and SCM community. However, both research areas have developed independently of each other which has resulted in various approaches that target the management of variability in space, time, or both, while their commonalities and differences are unclear. Gaining a deep understanding of the existing approaches and tools not only allows to reason about their compatibility, but also enables to detect gaps in the state of the art of managing both variability dimensions. Therefore, the first research question is introduced and further subdivided:

RQ 1 How can existing approaches that cope with variability in space, time, and both be unified to provide a common foundation that also advances state of the art by addressing the identified gaps?

- **RQ 1.1** Which concepts and relations exist to cope with either or both variability dimensions in the studied approaches and how can they be unified?
- **RQ 1.2** Which operations are provided to cope with either or both variability dimensions in the studied approaches and how can they be unified?
- **RQ 1.3** How can the appropriateness of a unification with respect to the studied approaches be quantified?

Answering RQ 1 helps to uniformly manage variability in space and time simultaneously by means of concepts, their relations, and operations based on state of the art. Nonetheless, the consistent evolution of systems dealing with both variability dimensions, that are additionally comprised of heterogeneous artifacts, is still a major problem. To this end, the Vitruvius approach [116] supports view-based consistency preservation for heterogeneous artifacts. Leveraging the consistency preserving mechanisms of Vitruvius and understanding which variability-related inconsistency types (including their causes and possible repairs) can occur during evolution allows for dealing with variability-related inconsistencies. Therefore, the second research question is introduced and further subdivided:

RQ 2 How can the consistent evolution of variable systems comprised of heterogeneous artifacts be supported?

RQ 2.1 What types of inconsistency can occur in variable systems?

RQ 2.2 How can unified operations be combined with consistency preservation of variability-related inconsistency types?

RQ 2.3 How can the Vitruvius approach be leveraged to support variability in space and time and preserve consistency in variable systems comprised of heterogeneous artifacts?

Answering RQ 2 finally enables to support consistent evolution of systems that vary in space and time comprising heterogeneous artifacts.

1.3. Envisioned Solution and Contributions

In this thesis, a unified solution is proposed to address the described problems and answer the research questions. In the following, every part of the envisioned solution along with the respective contributions is described.

Unifying concepts of SPLE and SCM to explicitly and proactively manage variability in space and time is gaining traction [34, 110, 140, 172, 126, 82, 235]. In this regard, pioneer work by Conradi and Westfechtel [48, 241] extends version models and relates concepts of variability in space and time. However, this works prescribe either development processes or implementation specifics (e.g., propositional logic or deltas). While recent work primarily focuses on classifying and comparing approaches coping with variability in space or variability in time [26, 77, 185], contemporary approaches from the upcoming research area of *Variation Control Systems (VarCS)* [141] (e.g., SuperMod [215] or ECCO [73]), manage the evolution of a variable system in a uniform manner. Thus, VarCS play a significant role in the first part of the solution: A *classification and unification of concepts of approaches dealing with variability in space or/and time, and alignment of terminology used in the research areas of SPLE and SCM.* In contrast to prior approaches, unified concepts shall be devised that appropriately cover and describe all relevant concepts of contemporary tools for dealing with variability in space and time. This leads to the first contribution of this thesis:

C1 A *unified conceptual model* that serves as common base for communication, scoping and comparison. The conceptual model goes beyond the mere combination of concepts of SPLE and SCM and introduces novel hybrid relations to uniformly cope with variability in space and time, guiding the conception of novel approaches (RQ 1.1).

Besides a unified data structure, appropriate operations are required for the operational management of both variability dimensions. Many of the VarCS, that deal with variability in space and time, provide such operations. However, they vary considerably in their behavior regarding how a system can be edited (i.e., directly or via well-defined views) and the paradigm followed to develop a variable system (i.e., product-oriented or platform-oriented). A survey on the operations of VarCS [137, 141] revealed that many provide view-based editing capabilities due to various advantages (e.g., a higher degree of automation). Moreover, view-based editing is well-suited for handling heterogeneous artifacts as well as variable systems, since it can represent both a specific type of artifact [22, 72, 116] and a particular product [73, 141, 215, 229]. Consequently, developers work on a specific product to evolve a variable system. This does not only reduce the complexity of applying changes to the product line, but also liberates from the burden of employing a suitable variability mechanism for each artifact type, as described in Section 1.1. Therefore, *operations of approaches dealing with variability in space or/and time are classified. Additionally, novel, view-based unified operations are conceived.* The static structure (i.e., concepts and relations) of the unified conceptual model together with unified view-based operations provide a foundation to develop novel solutions that deal with both variability in space and time, leading to the second contribution of this thesis:

C2 *Unified operations* as operational management of variability in space and time that operate on the unified conceptual model as data structure (RQ 1.2).

To goal of the unification of elicited approaches is to unify their concepts, relations and operations such that they are neither too specific nor too generic. Guizzardi et al. [90] propose a *framework for language evaluation* and introduces properties to assess the design of modeling languages. Since no means exist to quantify and evaluate the appropriateness of abstractions with respect to concrete approaches yet, the respective framework is extended and metrics for unification are defined, leading to the third contribution of this thesis:

C3 *Metrics for unification* to evaluate the appropriateness of granularity and coverage of a unification with respect to a set of selected tools (RQ 1.3).

Contributions C1 and C2 fully cover the unified management of variability in space and time. However, evolving a system that copes with both variability dimensions and consists of heterogeneous artifacts easily leads to variability-related inconsistencies. Such inconsistencies can occur on the level of abstraction of the product line, e.g., in the variability model [120, 100] (commonly referred to as the *problem space* [191]), within the implementation of a product consisting of heterogeneous artifacts and between products [39, 186] (commonly referred to as the *solution space* [191]), or involving both spaces, often referred to as software product line co-evolution [21, 40, 65, 112]. Numerous approaches tackle their detection and repair, but the research landscape has not been mapped, which impedes communication and scoping of research. Therefore, a literature survey is performed and its results are generalized and mapped to a classification schema to obtain a set of complete and disjoint variability-related inconsistency types that can occur during the evolution of a variable system. This leads to the fourth contribution of this thesis.

C4 A *classification and enumeration of variability-related inconsistency types* along with their possible causes, effects, and repair options (RQ 2.1).

Although a considerable amount of research has been conducted on inconsistency detection and repair in variable systems [146, 149, 100, 85, 208], consistency preservation in a system dealing with variability in space and time simultaneously has been much less addressed while consistency preservation between heterogeneous artifacts of a product is hardly considered in SPLE research yet [64, 141]. With this research, the aim is to bridge the gap between existing approaches for consistency preservation between heterogeneous artifacts and variability-related inconsistencies that can occur in the solution space. The key idea is to embed the consistency preserving mechanisms of Vitruvius in the evolution process of a system dealing with variability in space and time. This contribution is based on the unified concepts and operations of C1 and C2 as well as on the identified variability-related inconsistency types and repairs of C3. Therefore, *the unified conceptual model is refined* and *the unified operations are augmented with consistency-preservation capabilities.* This leads to the fifth contribution.

C5 A *unified approach* for automated consistency preservation during view-based evolution of variable heterogeneous systems (RQ 2.2, RQ 2.3).

This contribution leverages and extends the Vitruvius approach [116] to cope with variability and handle variability-related inconsistencies. Specifically, artifact views in Vitruvius are generalized to product views and existing consistency preservation mechanisms are utilized for different artifact types.

To evaluate the unified approach, real-world data sets of variable systems must meet various requirements: i) provide an evolution history, ii) consist of multiple types of artifacts, iii) employ a variability model and, iv) be realistic and publicly available. Since such data sets are barely available, the widely used ArgoUML-SPL data set [49] (a snapshot of the ArgoUML modeling tool) has been manually evolved by retroactively replaying the revision history of ArgoUML on the ArgoUML-SPL. Thus, a data set that constitutes the final contribution of this thesis is contributed:

C6 An *evolved data set of the ArgoUML-SPL* comprising nine revisions.

The contributions C1, C2, and C4 represent main contributions towards the principal contribution C5. The contributions C3 and C6 represent subordinate contributions.

1.4. Assumptions

For the principal contribution that constitutes the unified approach (C5), several assumptions are made that are introduced in this section.

The development process of variable software-intensive systems is model-driven and view-based. Different models describe the system of discourse containing (partially overlapping) information about the system. Thus, models represent views on different parts of the system, either on the abstraction of the variable system (i.e., the problem space) or on its implementation (i.e., the solution space).

Variability-related inconsistencies may not always be preserved fully automatically. Since SPLE enables intensional versioning (i.e., the specification of configurations where features can be combined before having ever been combined in a product [48]), their implementation may conflict. In such cases, consistency cannot be ensured at all time but instead is assumed to be repaired by the developer.

Furthermore, a metamodel must be available for every type of engineering artifact in the solution space. Moreover, for every pair of metamodels, consistency preservation rules must be available that use model transformations to preserve consistency in other models, thus allowing to consistently cope with arbitrary artifact types. Consequently, the preservation of consistency ranges from suggestions to developers to fully automated repairs of the system under construction.

1.5. Research Overview

Figure 1.1 depicts the relation between the described research goal, the problem statements, research questions and contributions. Addressing the first problem statement (P1) requires to classify, compare and unify existing approaches. In consequence, it allows to answer RQ 1 by providing a common foundation of concepts, their relations and operations, and address gaps that currently exist in state of the art. A unified conceptual model is contributed (C1), unified operations (C2) (that operate on the unified conceptual model as data structure), and metrics to quantify the appropriateness of granularity and coverage of a unification with respect to the studied approaches (C3).

Addressing the second problem statement (P2) requires to understand which variability-related inconsistency types may occur during evolution, how to augment unified operations with automated consistency preservation and how to leverage the Vitruvius approach to preserve consistency among heterogeneous artifacts. In consequence, it allows for answering RQ 2 by enumerating variability-related inconsistency types, their causes and possible repairs (C4) and, ultimately, a unified approach to support the consistent evolution of variable systems composed of heterogeneous artifacts (C5). The functional suitability of the unified approach is evaluated based on the two real-world data sets MobileMedia [71] and the widely used ArgoUML-SPL, which is extended with an evolution history from the original ArgoUML (C6). Note that the contributions of this thesis are not independent. Instead, every contribution serves as input to the following ones or is used for evaluation.

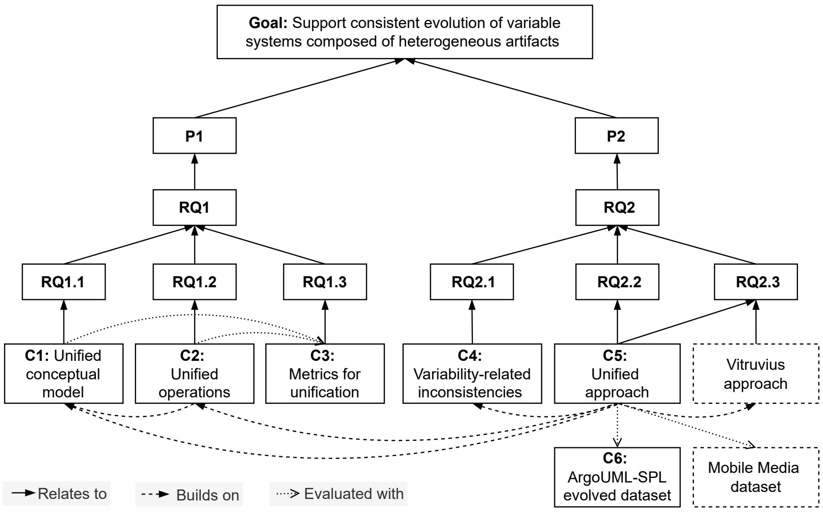

Goal: Support consistent evolution of variable systems composed of heterogeneous artifacts
P1
P2
RQ1
RQ2
RQ1.1
RQ1.2
RQ1.3
RQ2.1
RQ2.2
RQ2.3
C1: Unified conceptual model
C2: Unified operations
C3: Metrics for unification
C4: Variability-related inconsistencies
C5: Unified approach
Vitruvius approach
C6: ArgoUML-SPL evolved dataset
Mobile Media dataset
Relates to
Builds on
Evaluated with

Figure 1.1.: Overview of goal, problems (P), research questions (RQ) and contributions (C).

1.6. Thesis Outline

The remainder of this thesis is structured as follows.

Part I. The first part introduces the preliminaries of the thesis. Chapter 2 provides a running example that is used throughout this thesis, introduces basic definitions and comprises a description of contemporary tools for variability in space, time, and both as well as of relevant engineering paradigms.

Part II. The second part comprises the main contributions to support unified consistency-aware variability management. Along with the general unification process in Chapter 4, it presents the *unified conceptual model* in Chapter 5 and the *unified operations* in Chapter 6. Moreover, this part encompasses a classification and enumeration of *variability-related inconsistency types* in Chapter 7 and, ultimately, the *unified approach* building upon the preceding contributions in Chapter 8.

Part III. The third part of this thesis presents an empirical evaluation of the main contributions, structured according to the GQM method [27]. It comprises the general evaluation process along with the *metrics for unification* in Chapter 10. The specialized evaluation process, questions and results are presented for the unified conceptual model in Chapter 11, for the unified operations in Chapter 12, and for the unified approach Chapter 13, comprising the *evolved data set of the ArgoUML-SPL*.

Part IV. The fourth and final part reflects on this thesis' contributions. It encompasses brief answers to the research questions in Section 15.1, comprises a discussion of related work in Chapter 14, and concludes the thesis with a consideration of the industrial relevance of the conducted research in Chapter 15.

2. Foundations

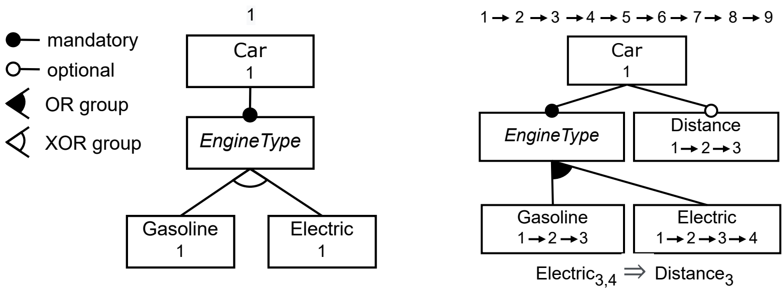
This chapter encompasses the fundamentals of this thesis. Section 2.1 introduces the running example that is used throughout the thesis. This is followed by foundations of variability in space in Section 2.2, variability in time in Section 2.3 and their combination in Section 2.4 comprising basic concepts and contemporary tools. In Section 2.5 and Section 2.6, the basics of model-driven software engineering and view-based software engineering are briefly described. To this end, the Vitruvius approach is introduced which is employed in this research. Finally, in Section 2.7, variability-related consistency notions are defined that are used throughout this thesis.

2.1. Running Example

Figure 2.1 shows an illustrative example of a simple `Car` system that exists in nine revisions. In its first revision, it comprises either a `Gasoline (Gas)` or an `Electric (Ele)` engine. In later revisions, an optional output of remaining `Distance (Dist)` is added, along with a constraint that `Ele` requires `Dist`. Ultimately, the feature `Car` has one revision, features `Gas` and `Dist` have three revisions, and the feature `Ele` has four revisions. `EngineType (ET)` is an abstract feature without implementation and thus without revisions. Figure 2.2 shows a Java source code and a SysML block definition diagram view on the implementation of the system in its final revision. For each line of code, the respective comment (highlighted in green) depicts the mapping to a revision of a feature or feature interaction.

2.2. Variability in Space

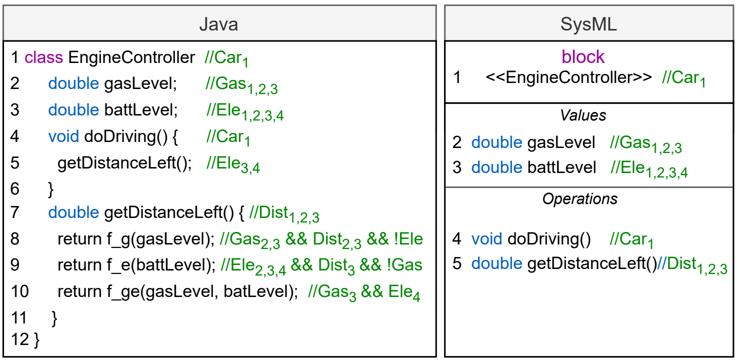
With an increasing demand for customized systems due to a variety of requirements, variability has become a key characteristic of many systems.



(a) Initial feature model.

(b) Final feature model.

Figure 2.1.: Feature model evolution. Adapted from [3, Fig. 1].



Java

```
class EngineController  //Car_1
    double gasLevel;      //Gas_{1,2,3}
    double battLevel;     //Ele_{1,2,3,4}
    void doDriving() {    //Car_1
      getDistanceLeft();  //Ele_{3,4}
    }
    double getDistanceLeft() { //Dist_{1,2,3}
      return f_g(gasLevel); //Gas_{2,3} && Dist_{2,3} && !Ele
      return f_e(battLevel); //Ele_{2,3,4} && Dist_3 && !Gas
      return f_ge(gasLevel, batLevel);  //Gas_3 && Ele_4
    }
}
```

SysML

```
block
<<EngineController>>  //Car_1
Values
double gasLevel   //Gas_{1,2,3}
double battLevel  //Ele_{1,2,3,4}
Operations
void doDriving()    //Car_1
double getDistanceLeft()//Dist_{1,2,3}
```

Figure 2.2.: System implementation (Car.java left and Car.sysml right). Adapted from [3, Fig. 1].

The term *variability in space* constitutes concurrent variations of a system in terms of products or features. In the running example of the `Car` system, the developer is able to choose between a gasoline engine (i.e., feature `Gas`) or an electric engine ((i.e., feature `Gas`). In this thesis, Definition 2.1 for variability in space is used.

Definition 2.1 (Variability in space) *"Variability in space is the existence of an artefact in different shapes at the same time." [191, p. 66]*

Variability in space is extensively studied in the context of software product line engineering that is introduced in the following.

2.2.1. Software Product Line Engineering

Software Product Line Engineering (SPLE) [107, 191, 45] is an established engineering paradigm to systematically engineer and manage the reusability and extensibility of software. It promotes a *reusable platform* (i.e., a set of reusable artifacts) that can be configured individually for each product [191]. The commonalities and differences across products are commonly represented by *features*, where a feature represents a "prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system" [107, p. 3]. A product line covers a *domain* (e.g., cars). The domain abstraction of a product line helps customers and developers understand the commonality and the variability of the product line. In this thesis, the domain abstraction of a product line is simply referred to as *domain*.

Variability Models. A *variability model* represents the configurable knowledge of an SPL where *constraints* between features govern which combinations are valid or invalid [107, 29, 51]. The variability model captures the entirety of constraints and, as a consequence, describes all configuration rules that must be satisfied when deriving a product. Different notions of variability models exist, for instance, feature models [107] or decision models [164]. Feature models represent the commonalities and differences of an SPL in terms of a hierarchical feature tree topology, illustrated in the running example in Figure 2.1. `Tree constraints` impose dependencies between features in a group or between a parent feature and its child feature. A *mandatory* feature is always selected together with its parent feature in a configuration (e.g., feature `ET`). Contrary, an *optional* feature requires its parent feature to be selected as well, while the parent feature may be selected without the optional child feature (e.g., feature `Dist`). Moreover, *alternative* or *or* groups govern the number of features in a group that must be present in a configuration. Feature models might also comprise *core* features that itself are mandatory as well as all of its predecessors in the feature tree. Thus, they must be present in every configuration and represent a commonality of an SPL. Finally, `Cross-tree Constraints` represent dependencies between features and span across the feature tree (e.g., the `Ele` feature implies the `Dist` feature so that both must be selected in a configuration).

Problem Space and Solution Space. Czarnecki and Eisenecker [50] introduce the problem space and the solution space of an SPL. The problem space represents the variability of a domain and comprises artifacts, such as the variability model. The solution space represents the implementation in the form of a reusable platform comprising different types of artifacts, such as models or source code (e.g., Java or SysML). To manage variability holistically, artifacts of both spaces must be linked.

Development Processes. There are three ways for engineering an SPL: the proactive, reactive, and extractive approach [191, 125]. The proactive approach describes a development of the product line from scratch. This requires to initially perform *domain engineering*, e.g., specifying the variability model and implementing reusable artifacts among all products. Subsequently, *application engineering* is performed for every product, e.g., configuring the product and implementing product-specific artifacts. The reactive approach starts by specifying the variability model and implementing artifacts for only a few products. Incrementally, more products are added. Finally, the extractive approach uses existing products (often realized by clone-and-own [200]) as base and extracts the variability model from these products. Depending on which engineering strategy is employed by an organization, the dominant model varies. In case of the proactive approach, the dominant models origin from the specification and implementation of the domain engineering, such as the variability model. Consequently, the application engineering builds on the domain engineering, e.g., the variability model guides the derivation of viable products. In case of the reactive approach, the dominant models change, as domain engineering and application engineering alternate. In case of the extractive approach, the products are the dominant models which prescribe the features, their dependencies and reusable artifacts which are specified and implemented for domain engineering.

Variability Mechanisms. The derivation of a product from a reusable platform depends on the employed variability mechanism in the solution space, out of which three categories exist: annotative, compositional and transformational [14, 232]. Annotative mechanisms [13], such as preprocessor directives, assemble all variations of implementation artifacts of an SPL in one monolithic artifact that is commonly referred to as the *150%* model. Annotations relate parts of the 150% model with features or feature interactions. Based on a configuration, only those parts are retained whose annotations are satisfied by a configuration. Compositional mechanisms [41] are employed by different paradigms such as feature-oriented programming [28] or aspect-oriented programming [111]. Based on a *core model*, commonly referred to as the *75%* model, and a configuration, relevant implementation artifacts are assembled (usually by a composer) to create a product. Finally, transformational mechanisms employ transformation operations that add, modify, or remove elements to transition from one product to another. Delta modeling [205] is a prominent realization of the transformational mechanism. It is based on a *core module* and a set of *delta modules* that apply changes (i.e., *deltas*) to the core module to obtain a valid product.

2.2.2. Contemporary Tools

Variability in space is supported by different tools in research and industry, employing different variability mechanisms.

Several limitations of current variability management practices increase the complexity of managing variable systems [141], such as the manual modification of *mappings* (that represent a relation between features and implementation artifacts), the manual integration of changes into the platform, the missing uniform management of product variants and revisions, or concrete variability mechanisms that are specific for a certain type of artifact and further complicate the variability management of systems composed of heterogeneous artifacts. The upcoming research area of *Variation Control Systems (VarCS)* aims to overcome these limitations. In this thesis, Definition 2.2 for VarCS is used.

Definition 2.2 (Variation Control System) *"A Variation Control System (VarCS) supports managing variant-rich systems in terms of features. It supports editing variant subsets, which are represented by a selection of features, and it automatically integrates the edited variant subsets back into the variant-rich system in a transactional way." [141, p. 2].*

In the following, four SPLE tools including one VarCS are presented.

FeatureIDE [130, 157] is a popular open-source framework for modeling, analyzing and developing SPLs. It builds on the Eclipse platform and is used both in academia and industry. FeatureIDE supports development phases of SPLs such as domain analysis (that, for example, comprises extensive feature modeling and analyses), domain implementation and semi-automated generation of configurations, compilation and testing (i.e., product-based analysis). FeatureIDE is extensible via composers that enable the use of different variability mechanisms and thus support variability mechanisms of all three categories.

VTS (Variation Tracking System) [228] is a VarCS that manages variability in space. It allows for editing products and partial views on the SPL, and integrating changes automatically based on a manually provided *ambition* that specifies the edited features. To conform to basic consistency principles given by the *lens laws* [76], its *get* and *put* operations are used to derive views from the platform and integrate changes back into the SPL, while any change performed in the view shall not affect features that are not part of it. VTS employs an annotative variability mechanism with proprocessor annotations in text files to represent the SPL as well as the derived views.

SiPL [187, 188] is a delta-based modeling framework for SPLE that employs a transformational variability mechanism. Variability in space is captured by delta modules. A delta is refined by a consistency-preserving edit script generated by comparing two models. Thus, deltas are not specified manually but computed fully automatically. Based on edit scrips, SiPL provides analyses of deltas such as dependencies between deltas (i.e., deltas can only be applied in a certain order) or conflicts (i.e., deltas cannot be applied together). While delta modeling constitutes the solution space of SiPL, the tool integrates FeatureIDE to describe the problem space.

pure::variants [35] is a commercial SPL tool used in industry. It provides support for developing, testing and maintaining SPLs while employing several variability mechanisms. In particular, it focuses on the annotative variability mechanism by means of preprocessor directives. Note that in this thesis, the *pure::variants evaluation edition* is considered. Further proprietary tools similar to pure::variants exist, such as Gears from BigLever [37].

2.3. Variability in Time

Over the course of time, a system evolves due to refactorings, bug fixes or changed requirements [134]. The term *variability in time* constitutes sequential variations of a system in terms of revisions. Thus, it represents a further dimension of variability. In the running example of the `Car` system, the developer is able to choose from the nine (system) revisions of the feature model. The first revision enables the `Car` feature, its mandatory child `Engine Type`, and its alternative feature children `Gasoline` and `Electric`. In this thesis, Definition 2.3 for variability in time is used.

Definition 2.3 (Variability in time) *"Variability in time is the existence of different versions of an artifact that are valid at different times." [191, p. 65]*

Variability in time is extensively studied in the context of software configuration management that is introduced in the following.

2.3.1. Software Configuration Management

Software Configuration Management (SCM) [48] is an engineering paradigm that supports identification, controlling, and traceability of the software system during its evolution. For example, some of the fundamental aspects of SCM constitute the unambiguous identification of any software artifact, their storage and access, the concurrent modification of artifacts, and the alignment with a specific development process.

Version Models. Analogously to variability models, SCM promotes version models that manage changes within directories or files and provide operations to retrieve old versions and construct new ones [48]. In SCM, a version describes an abstract concept that is specialized by either a *revision* (i.e., a version that is intended to supersede its predecessor) or a *variant* (i.e., versions that are intended to coexist). Although the management of variants has been recognized in SCM [241], it has been largely sidestepped [151, 141].

Extensional and Intensional Versioning. The SCM community differentiates between two kinds of version space organization: *extensional versioning* and *intensional versioning* [48]. While extensional versioning describes the sole retrieval of versions that have been previously constructed, intensional versioning allows for retrieving versions in a flexible manner such that new combinations of software artifacts are constructed on demand.

Version Control. *Version Control (VC)* [201] represents a sub-discipline of SCM. It supports several functionalities such as persisting and identifying different revisions of software artifacts in a *repository*, or providing *working copies* to developers that can be modified and integrated back into the repository leading to a new version. *Revisions graphs* are employed that vary from an ordered sequence of revisions to an acyclic graph.

2.3.2. Contemporary Tools

SVN and Git are two well-established *Version Control Systems (VCS)* that both rely on extensional versioning.

Subversion (SVN) [190] is an open-source centralized VCS. Typically for version control, developers are able to *checkout* a central repository at a certain point in time (i.e., a specific revision identified by a `revision number`) in a local workspace. Performed modifications can be *commited* back to the central repository, leading to a new revision. SVN supports branches (that are created as directories) and their merging.

Git [145] is an open-source decentralized VCS. In contrast to SVN, Git supports multiple distributed repositories. Upon the distributed operation *clone*, an exact copy of a repository is provided to a user, leading to a distributed network of repositories. Modifications are first to be integrated into a local copy of the repository via *commit* and then distributed to other copies of the repository via *push* to synchronize clones.

2.4. Variability in Space and Time

Variability in space and variability in time are highly intertwined. The evolution of a system may affect available configuration options which, vice versa, may guide the evolution of the system [231, 216]. In the following, state of the art is briefly recapped and contemporary tools dealing with both variability dimensions are introduced.

2.4.1. State of the Art

A missing common foundation and techniques for proactively managing variability in space and time have led to a plethora of approaches [59, 124, 182, 179, 160]. For instance, VCSs do not support the concept of a feature. Instead, the products or features are managed in branches, which requires high maintenance effort [186, 200, 243, 131]. Thus, retroactively mining feature evolution information is the focus of several research [160, 59, 128, 199], leading to high additional costs [127]. As a consequence, new approaches have emerged in the last years that promote the explicit and proactive management of variability in space and time. For example, by extending feature models to additionally document revisions of individual features (i.e., *hyper feature models* [218]), or the upcoming research area of VarCS that encourages an integrated and uniform view on research from SPLE and SCM.

2.4.2. Contemporary Tools

The following five tools cope with both variability in space and time:

ECCO [75, 73, 139, 138] is a VarCS that supports a feature-oriented and distributed development of variable systems. Originally, ECCO was an approach used for feature location and has evolved to a VarCS following the *checkout-modify-commit* workflow of products while still employing feature location for computing mappings. ECCO promotes feature revisions in the problem space but no variability model (and, consequently, no constraints). Moreover, the tool can be extended via adapters to support different types of artifacts.

SuperMod [215, 214, 212] represents a model-driven VarCS that is based on an annotative variability mechanism. It builds upon and extends the *uniform version model* [241]. Analogously to ECCO or VTS, SuperMod follows the same transactional workflow of evolving an SPL product-wise. In contrast, it supports system revisions and provides a feature model representing one particular revision to the user that can be used for configuring a product. For integrating the changes performed on a product and similar to VTS, SuperMod requires an *ambition* (i.e., a logical expression) upon any commit. The ambition is used to annotate the elements of the product line with visibility information including the system revision.

DeltaEcore [219, 220, 216] is a model-driven tool-suite that employs a transformational variability mechanism based on delta-modeling. It introduces the *hyper feature model* [218] which constitutes an extended feature model with feature revisions that represent further configurable units for product definition. Moreover, it is possible to specify constraints in the problem space between features and ranges of feature revisions. DeltaEcore automatically derives delta languages that comprise delta operations based on metamodels. Developers can use the delta languages to specify concrete delta modules that they can map to features and feature revisions.

DarwinSPL [170, 169] is a model-driven tool suite that integrates DeltaEcore for product derivation. In contrast to DeltaEcore, DarwinSPL introduces the *temporal feature model* in the problem space that captures the evolution of the whole system (in contrast to feature revisions, that are not explicitly supported). In addition, it supports the planning of future evolution of the SPL as well as its re-planning. To consider changing functionality based on a different environment (i.e., context), DarwinSPL integrates contextual information that restricts the configurable space.

VaVe [10] is a model-driven tool that realizes a uniform management of variability in space and time via features and feature revisions (referred to as *VA*riants (space) and *VE*rsions (time)). Similar to ECCO and SuperMod, the tool supports the product-wise development of an SPL while automatically integrating the changes into the platform, constituting to a VarCS. Moreover, it provides import and export functionalities with FeatureIDE to make use of its advanced feature modeling capabilities.

Some of the tools dealing with variability in space, time, or both could be used in combination to support both variability dimensions simultaneously, such as FeatureIDE and Git. However, such combinations are not considered in this research because they do not contribute new concepts or relations for uniform variability management.

2.5. Model-Driven Software Development

According to Stachowiak's general model theory [223], a *model* exposes three main characteristics: *representation*, *abstraction*, and *pragmatism*.

Representation. A model is a mapping from or a representation of the *original* it reflects. An original might be natural or artificial, such as an existing entity or a concept.

Abstraction. A model comprises only a seemingly relevant subset of the properties of the *original* it represents. Thus, models are abstract representations of originals.

Pragmatism. A model is designed for a specific purpose and be capable of replacing the original regarding that purpose.

Moreover, a model can have one or more roles that reflect its purpose [46]: *descriptive* (reflecting a system and its current or past properties), *predictive* (reflecting the future system behavior to allow, for instance, decision-making and simulations) and, *prescriptive* (reflecting the system's design to be eventually taking into account for realization).

Model-Driven Software Development (MDSD) [86, 224] systematically employs the idea of models and raises them as first-class artifacts for software development. As a consequence, models are not solely used for documentation and communication, but also to generate executable code from. In this thesis, any artifact (regardless of whether in the problem space in the solution space) is considered a model. This comprises, for instance, employing a metamodel for Java code in the solution space of a variable system. In the following, foundations of modeling languages and metamodels are introduced.

2.5.1. Metamodels, Modeling Formalisms and Languages

In MDSD, each model conforms to a certain *metamodel*. A metamodel specifies all types of model elements and relations between them. As a consequence, each model can be considered an instance of a metamodel. The *Meta-Object Facility (MOF)* is a standard from the *Object Management Group (OMG)*. It describes the relation between models and their metamodels as a four-layer hierarchy that comprises models at Layer M1, metamodels at Layer M2 and a self-describing meta-metamodel at Layer M3 representing the highest level of abstraction. Layer M0 comprises real world objects.

According to Völter et al. [239, p. 26], a *modeling language* specifies the *concrete syntax*, the *abstract syntax*, the *static semantics*, and the *execution semantics*:

Concrete Syntax. A user-facing representation of a model (e.g., textual or graphical) that developers can interact with directly.

Abstract Syntax. An internal representation of a model (e.g., tree or graph) that is not visible to developers.

Static Semantics. Additional constraints of a model (e.g., specified with the Object Constraint Language [181]) that must hold beyond the model's structural well-formedness.

Execution Semantics. The behavior of a model during its execution.

To be able to treat source code as models, the grammar of the respective textual language (e.g., the Java programming language) can be represented as a metamodel. The concrete syntax (e.g., concrete Java source code) must be parsed and transformed into the abstract syntax representation (i.e., the Abstract Syntax Tree (AST)), which is a model that is an instance of the respective metamodel.

2.5.2. Modeling Frameworks

The *Essential MOF (EMOF)* standard is a subset of the comprehensive MOF standard and is implemented by the meta-metamodel Ecore. The *Eclipse Modeling Framework (EMF)* [225] is a framework that contains Ecore and that offers a wide range of tools to leverage MDSD. The EObject of the Ecore meta-metamodel is the super class of all Ecore model elements (analogously to all Java classes extending Object). Ecore defines model elements such as EClasses (a class with zero or more attributes and references), EAttributes (an attribute with a name and a type) and EReferences (one end of an association between two classes, with an explicit containment attribute). Every Ecore model has an EPackage as root element. EPackages can contain further EPackages as well as EClassifiers (i.e., EClass or EDataTypes). To this end, the implementation of the devised concepts in this thesis is based on EMF.

2.6. View-Based Software Development

The complexity of systems is ever increasing, challenging their development, evolution, and maintenance [163]. Many approaches have been proposed to cope with large and complex systems, such as automotive systems, that are comprised of heterogeneous artifact types (e.g., CAD diagrams from mechanical engineering, circuit diagrams from electrical engineering, or source code from software engineering). View-based development has the potential to keep cognitive complexity at bay via the usage of tailored views, each showing a particular part of a system [72, 43]. As a consequence, models can be changed only through well-defined views. This, in particular, provides distinct advantages for coping with variability [141]. Views may only show a part (e.g., a specific product) of the entire variable system, and, as a consequence, reduce the complexity of developing and maintaining it. Moreover, view-based development allows for higher automation, such as the automated computation of mappings (i.e., relations between features and their implementation-specific artifacts), which otherwise is tedious and error-prone [214].

The ISO 42010 standard for architecture description distinguishes the synthetic and the projective approach for the construction of views [102]. In a synthetic approach, information about the system is distributed over several distinct views and their relations. In a projective approach, information is centralized in a so-called *Single Underlying Model (SUM)* [22]. The SUM conforms to the *Single Underlying Metamodel (SUM metamodel)* [116]. In the context of view-based software development and variability modeling, *projectional editing* [240, 228] corresponds to the projective view construction approach and represents a product of the variable system derived from the reusable platform. For both the synthetic and projective approach and view-based software development in general, the preservation of consistency between views is a major challenge and extensively researched [197, 226, 150].

2.6.1. Orthographic Software Modeling

Orthographic Software Modeling (OSM) constitutes an approach for view-based development [23, 22]. It employs a SUM that is assumed to be neither exposed to redundancies nor dependencies and thus be free of any inconsistency thereof. The OSM realizes a projective approach for the creation and management of views that are created on demand. A view conforms to a metamodel, which is also referred to as *view type* [84]. The *dimension-based view navigation* introduces a scheme for navigating along different perspectives of the system. A dimension represents a property of a system's description, such as its composition (e.g., the (de)composition of components into sub-components) or its abstraction level (e.g. the platform independent model (PIM) or implementation (e.g., Java)). Moreover, a dimension may enumerate the product variants of the system. The number of dimensions is not limited and induces a multi-dimensional cube, each view representing a cell in it. The direction of a dimension-based view navigation is defined based on a dominance hierarchy between the dimensions, while choices made of a higher level dimension may affect the remaining choices of a lower level dimension. For instance, selecting the abstraction level (which, in the MDSD context, could adhere to the most dominant dimension) affects the possible notations. For instance, selecting Java as level of abstraction only allows for a syntax tree (instead of a graphical notation). Consistency is achieved by transformations that create well-formed views from the SUM and integrate modifications on the view back into the SUM that, in turn, can be checked against well-formedness rules. Thus, views do not need to be pair-wise kept consistent as long as the view creation and change integration adheres to consistency constraints that constitute well-formedness rules.

2.6.2. The Vitruvius Approach

The model-driven and delta-based Vitruvius approach supports consistent view-based development of systems comprised of heterogeneous artifact models [116]. It bases on OSM and thus promotes the usage of a SUM. However, maintaining and developing a monolithic SUM that describes different types of engineering artifacts of the system while keeping it free from redundancies and dependencies is overly complex [156]. As a consequence, Vitruvius introduces the *Virtual Single Underlying Model (V-SUM)* conforming to the *Virtual Single Underlying Metamodel (V-SUM metamodel).* While the V-SUM metamodel appears again as a monolithic SUM, internally, the V-SUM metamodel consists of several coupled metamodels. Moreover, the V-SUM metamodel can contain redundancies and dependencies in contrast to the SUM metamodel.

To ensure consistency, metamodels are pair-wise coupled by manually defined incremental model transformations, i.e., *Consistency Preservation Rules (CPRs).* Performed changes by a developer (referred to as *original changes* throughout this thesis) are recorded by a change monitor and processed by CPRs to check and enforce consistency of instances of a V-SUM metamodel (referred to as *consequential changes* throughout this thesis). Changes comprise all modifications that transition one valid instance of a metamodel to another. Vitruvius uses a metamodel that represents all possible types of changes that can be applied to Ecore-based models, e.g., the insertion or removal of model elements. Moreover, it is possible to compose *atomic* changes to *compound* changes, such as moving a model element which is the composition of the two atomic changes *remove* and *insert.*
Moreover, Vitruvius inductively guarantees consistency. In particular, models are assumed to be in a consistent state before changes are processed that trigger consistency preserving mechanisms. Consistency preservation in Vitruvius can be defined by several languages: the declarative *Mappings* language [121], which is the foundation for the *Commonalities* language [114], and the imperative *Reactions* language [115], which can be used to realize

CPRs. Whenever a CPR modifies a model in response to changes in another model, correspondences between elements whose consistency shall be preserved are created, as specified in the CPRs, in a traceability model called *correspondence model*.

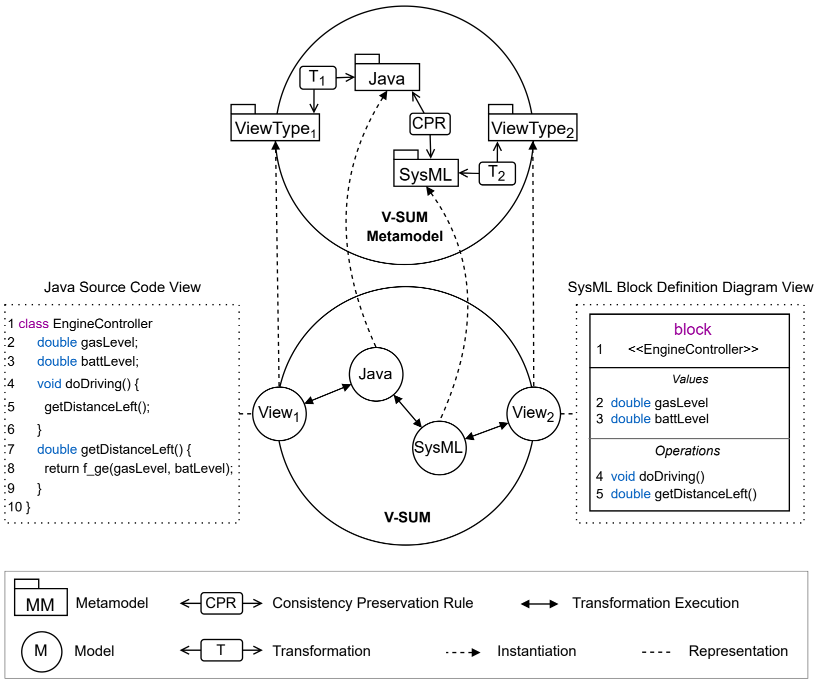
Figure 2.3 shows the V-SUM metamodel and V-SUM with derived views for the running example of the `Car` system. The V-SUM metamodel consists of a Java metamodel and a SysML metamodel. A transformation T specifies how instances of one or more metamodels can be transformed to a view of a certain type. The respective views View_1 and View_2 can be instantiated from ViewType_1 and ViewType_2 by executing the corresponding transformations T_1 and T_2. View_1 and View_2 correspond to a source code representation of Java and a SysML block definition diagram. A CPR is specified for a pair of metamodels and ensures that, for example, every time a class is added to the Java model (e.g., *EngineController*), a block with the same name is added to the SysML model and vice versa. In this thesis, CPRs are specified in the Reactions language and used to consistently propagate changes from a Java model to a UML model.

Vitruvius does not consider variability but provides consistency preservation mechanisms between heterogeneous artifact types while promoting projective views, which are well-suited for managing variable systems via views on particular products [141].

2.7. Consistency Notions

Numerous works focus on variability-related consistency management that ranges from inconsistency detection to their fully automated repair. However, there is no standardized or explicit definition of consistency in SPLE [132, 12]. Rather, consistency notions differ depending on whether the system evolves in the problem space, in the solution space, or in both spaces [182, 3]. Henceforth, three consistency notions of SPLE are distinguished throughout this thesis that are introduce in the following.

Increasing complexity of a variability model impairs its maintainability, making it prone to *anomalies* or *defects* [33]. Two types of anomalies are considered that can occur in a variability model. First, *structural anomalies* violate well-formedness rules of a variability model [100]. For instance, such rules



```
class EngineController
    double gasLevel;
    double battLevel;
    void doDriving() {
      getDistanceLeft();
    }
    double getDistanceLeft() {
      return f_ge(gasLevel, batLevel);
    }
}
```

Figure 2.3.: A V-SUM metamodel and V-SUM of the `Car` system with derived views. Adapted from [114, Fig. 2.3].

could specify that every feature must have a unique name and that a feature group must consist of at least two features. Second, variability models can also be prone to *semantic* anomalies [120, 33, 94]. Void feature models (i.e., no products can be derived from the product line), dead features (i.e., features can never be selected in any product of the product line), or redundant constraints (i.e., the removal of a constraint does not change the configurable space) are examples of such inconsistencies. Another form of problem space inconsistencies represent configurations that violate any constraint of the variability model [174, 78, 244]. Thus, a configuration must satisfy all constraints of the variability model. In this thesis, Definition 2.4 for problem space consistency is used.

Definition 2.4 (Problem space consistency) *A necessary criterion for problem space consistency is the absence of structural or semantic anomalies in the variability model. Moreover, a configuration c must be* valid *with respect to the variability model, i.e., for no constraints of the variability model ct_{vm}, the formula $c \wedge ct_{vm}$ is unsatisfiable.*

Changes in the implementation of a variable system can harm the solution space consistency. While different artifact types within one product of the variable system describe partially overlapping information that must be kept consistent [116, 62, 56, 242], products also share partially overlapping information in the form of features. Thus, if a feature changes in one product, all other products with the same feature must be changed accordingly [39, 186]. Consistency rules specify conditions that must hold in any consistent system. Such consistency rules can refer to artifacts of one particular type, or to artifacts of different types. Such rules can, for example, concern the syntactic correctness of Java source code with respect to its grammar, the conformance of a model to its metamodel and OCL rules [181] (i.e., within an artifact type), or a correspondence of a Java class structure to the class structure in a UML class diagram [116, 121] (i.e., between artifact types). Similarly, the products and the reusable artifacts must co-evolve consistently [81], e.g., if a feature is added or changed in a product, the artifacts from which the product was derived must be changed accordingly. In this thesis, Definition 2.5 for solution space consistency is used.

Definition 2.5 (Solution space consistency) *A necessary criterion for solution space consistency is the absence of violations of any of the given consistency rules. A consistency rule specifies a condition that must hold in any consistent system.*

Considering the variable system entirely, the problem space and the solution space may evolve independently from each other. In particular, after a product has been derived from the variable system, it evolves independently from the reusable platform and the variability model, which is subject to evolution as well. This is also commonly referred to as *product and product-line co-evolution* [198, 81, 112, 144, 21, 65, 211, 40]. Specifically, each valid configuration in the problem space must lead to a consistent product. This also entails that (non-abstract) features in the variability model must have an implementation. In this thesis, Definition 2.6 for problem space–solution space consistency is used.

Definition 2.6 (Problem space–Solution space consistency) *A necessary criterion for problem space–solution space consistency is that all valid configurations must lead to a consistent product.*

Part II.

Unified Consistency-Aware Variability Management

3. Overview

Part II presents the main contributions of this thesis. Chapter 4 explains the general unification process. Chapter 5 introduces concepts and relations that were identified from elicited tools in the SPLE and SCM research field, contributing a unified conceptual model (C1). Chapter 6 extends the unification work by identifying and devising unified operations that build on the unified conceptual model as data structure and constitute the operational management of variability in space and time (C2). Chapter 7 presents an enumeration and classification of identified variability-related inconsistency types that may occur during the evolution of a variable system including causes, effects and repair options (C4). Finally, the unified view-based approach is introduced in Chapter 8. It builds on the prior contributions by concretizing the unified conceptual model as well as the unified operations. Beyond that, the unified approach integrates variability-aware consistency preservation during the evolution of a variable system comprised of heterogeneous artifacts (C5).

4. Unification Process

This chapter builds on a publication at SPLC [3] and Empirical Software Engineering [5].

This chapter presents a general unification process to convey how the results were obtained with the purpose of reproducability and transparency and with the aim of improving validity. For the unified conceptual model (C1) and the unified operations (C2), the same general unification process was applied while being tailored to the specific contribution.

The construction of the unified conceptual model and specification of the unified operations was inspired by the construction process for a conceptual reference model proposed by Ahlemann and Riempp [1]. Specifically, the authors propose a four-phase research process. The first phase comprises the problem definition. The second phase aims at exploring and generating hypotheses, i.e., the construction of an initial model and an analysis of related approaches and standards, followed by a refinement of the initial model. The third phase targets the evaluation of the constructed model. Specifically, this is performed by conducting interviews with selected domain experts upon which the model is refined until the experts reach consensus. The evaluation also involves an application of the constructed model and a follow-up refinement of the model based on the insights. Finally, the fourth phase involves documentation which contains the model itself, a description of the construction process and a documentation of the interview results.

Based on the described phases, a general process tailored to the purpose of unification is presented in Section 4.1. An overview of selection criteria of contemporary tools that cope with variability in space, time, or both follows in Section 4.2. A summary in Section 4.3 closes this chapter.

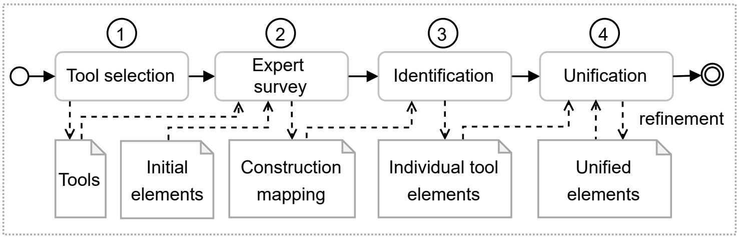


Figure 4.1.: General unification process [3, Fig. 2].

4.1. General Unification Process

Figure 4.1 shows the general unification process. The goal of the unification comprises two main aspects: First, a classification of individual tool elements, i.e., tool constructs and tool operations, to support an understanding of their commonalities and differences. Second, their unification in a redundancy-free and unambiguous manner, in order to i) provide a common base for researchers and practitioners to compare, communicate and scope work and research and, ii) support the development of novel techniques that cope with both variability dimensions simultaneously.

Step ① encompasses a selection of relevant tools based on specified selection criteria. Then, expert surveys using interviews or questionnaires are conducted in Step ②. Based on the gathered results from the expert survey, a construction mapping is obtained. It maps tool elements (such as constructs, their relations, well-formedness rules or operations) of the individual tools to initial elements that serve as initial hypothesis and starting point for the identification phase of unified elements. In the follow-up Step ③, semantically equivalent elements across the contemporary tools are identified and a classification is created. The final Step ④ encompasses a unification of the individual tool elements that result in unified elements.

Note that the general unification process is specialized for the unified conceptual model (C1) and unified operations (C2).

4.2. Selection Criteria for Contemporary Tools

To conduct the unification processes for the unified conceptual model and unified operations, a definition of selection criteria for contemporary tools to be used in the unification is presented in the following.

1. Support for variability in space, time, or both
2. Open source or expert availability
3. Consideration of the problem space *and* the solution space

The set of selected tools should be representative by means of involving tools for each variability dimension and combination as well as for each category of variability mechanism (i.e., annotative, transformational, and compositional) to collect a diverse set of tools. Consequently, tools are excluded that support *only* the solution space (e.g., FeatureHouse [16, 15]) or the problem space (e.g., tools that only address variability modeling or analyses [19, 30, 83, 207]). These requirements constrain the set of suitable tools from the SPLE community. Additionally, a study by Horcas et al. [101] reports that out of 97 tools supporting software product line engineering, only 19% are available online. Out of these, most systems only deal with the problem space and do not consider the solution space, which significantly limits the number of suitable tools.

4.3. Summary

This chapter presented the conducted unification process to foster reproducability. The process was inspired by the construction process for a conceptual reference model proposed by Ahlemann and Riempp [1]. Figure 4.1 shows the general unification process of the unified elements (i.e., the unified conceptual model (C1) and the unified operations (C2)). This chapter also provided selection criteria of contemporary tools for unification, such as support for either or both variability dimensions, and the consideration of the problem space as well as the solution space.

5. Unified Conceptual Model

This chapter builds on publications at VariVolution [9], SPLC [7] and Empirical Software Engineering [5]. An open-access repository comprises artifacts related to the construction and evaluation of the unified conceptual model.[1]

This chapter presents a conceptual model that unifies and relates concepts established in SPLE and SCM, and aligns their terminology. The goal is to provide a foundation for researchers and practitioners to compare, communicate and scope their work, obtain understanding of existing approaches coping with variability in space and time, and design novel techniques for managing both variability dimensions.

Referring to problem statement P1, the following research question is asked:

RQ 1.1 Which concepts and relations exist to cope with either or both variability dimensions and how can they be unified?

First, Section 5.1 presents the specialized unification process. Section 5.2 introduces the unified conceptual model along with main design decisions and well-formedness rules. Expected benefits of the unified conceptual model are described in Section 5.3. A summary in Section 5.4 closes this chapter.

This chapter thus constitutes the contribution C1.

5.1. Specialized Unification Process

Figure 5.1 presents a specialization of the general unification process shown in Figure 4.1 for the unified conceptual model. In the following, each step is described in detail.

[1] `https://doi.org/10.5281/zenodo.5751916`

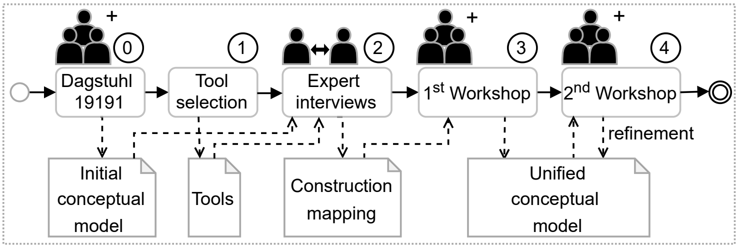


Figure 5.1.: Unification process of the unified conceptual model. Adapted from [5, Fig. 3].

Initial Conceptual Model

In the conceptual modeling group of the Dagstuhl seminar 19191 (Software Evolution in Time and Space: Unifying Version and Variability Management [34]) (Step ⓪), we conceived the initial conceptual model shown in Figure 5.2 [9]. It models concepts that were found common to systems supporting variability in space, time, or both (white) as well as concepts for variability in time (orange) and space (green). The `System Space` describes any software-intensive system. It is composed of `Fragments` that describe a system and represent different levels of granularity (e.g., a file or line of code). Due to a generalization of the composite design pattern, `Fragments` can be composed in different combinations. The `Revision Space` conceptually corresponds to the `System Space` and is a set of all systems under revision control. A `Versioned System` is described by its `Revisions`, each representing the system at a particular point in time. Subsequent and preceding revisions form a revision graph, while multiple direct successors and predecessors enable branching and merging. The `Variant Space` consists of all possible `Product Lines`. A `Variation Point` describes an option of a `Product Line` and associates its realizing `Fragment`. A `Product` results from a configuration that requires every `Variation Point` to be realized by its corresponding `Fragments`. The configuration is modeled with a ternary association of the `Product Line`, the `Variation Point`, and the `Fragment`. A `Versioned Item` represents the versioning of concepts of all three spaces. It is realized by the concepts `Product Line`, `Variation Point`, and `Fragment`.

Although the initial conceptual model documents concepts and their relations for variability in space and time, concepts are combined but not unified yet,

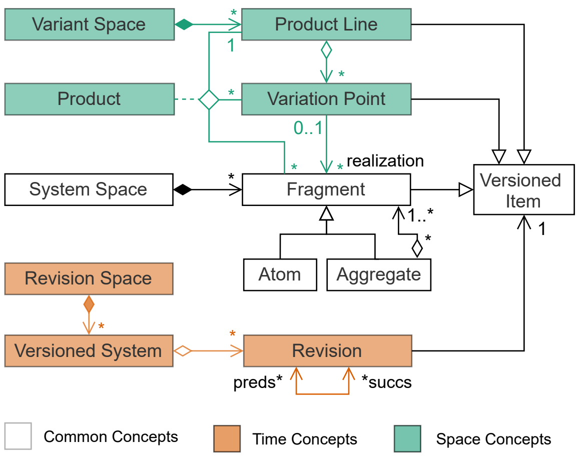


Figure 5.2.: UML class diagram of the initial conceptual model for variability in space and time [5, Fig. 2].

and have not been systematically selected and conceived. Therefore, the initial conceptual model serves as foundation for the unified conceptual model.

Tool Selection

To systematically refine the initial conceptual model, relevant contemporary tools were elicited based on the criteria described in Section 4.2 (Step ①). Table 5.1 shows the selected tools, and how they differ regarding their support of variability in space and/or time, and regarding the employed variability mechanism. FeatureIDE [130, 157], pure::variants [35] and SiPL [187, 188] represent tools that support variability in space. FeatureIDE supports different categories of variability mechanisms by means of integrated composers, e.g., the preprocessor *Antenna* as annotative mechanism or *AspectJ* as a compositional variability mechanism. The tool pure::variants employs an annotative mechanism by means of tagging model elements. Finally, the tool SiPL uses a transformational variability mechanism via deltas. SVN [190] and Git [145] are widely used VCSs that cope with variability in time. Since both tools do not explicitly cope with variability in space (e.g., features and constraints), they do not employ any variability mechanism. Finally, contemporary tools

Table 5.1.: Distinguishing characteristics of the selected tools.

Property / Tool	Space	Time	Annotative	Transformational	Compositional	Open Source
FeatureIDE	●	–	●	●	●	●
pure::variants	●	–	●	–	–	–
SiPL	●	–	–	●	–	●
SVN	–	●	–	–	–	–
Git	–	●	–	–	–	–
SuperMod	●	●	●	–	–	–
DarwinSPL	●	●	–	●	–	●
DeltaEcore	●	●	–	●	–	●
ECCO	●	●	–	–	●	●
VaVe	●	●	–	–	●	–

that deal with variability in space and time are SuperMod [215, 214], DarwinSPL [170], DeltaEcore [219, 220], ECCO [75, 73, 139], and VaVe [10]. As these tools employ different concepts, modalities, and development paradigms, diverse perspectives for unifying variability in space and time could be incorporated. A more detailed description of the tools for variability in space is provided in Section 2.2.2, of the tools for variability in time in Section 2.3.2, and of tools coping with both dimensions in Section 2.4.2. Although some tools could be used in combination to support variability in space and time, such as FeatureIDE and Git, they do not provide novel concepts or relations for dealing with both dimensions. Therefore, such combinations are considered to be covered by analyzing each tool individually.

Expert Interviews

To refine the initial conceptual model accordingly, I conducted semi-structured interviews, once per tool with one expert of the respective tool (Step ②). Experts were invited based on their involvement in the conceptual design or implementation of an elicited tool and thus are highly knowledgeable of it. The tool experts are mostly researchers from academia with one expert being from industry. Experts for SVN or Git were not invited as both tools are widely used with profound documentation available. The eight interviews lasted 83 minutes on average and were based on a blank interview guide that was jointly completed to ensure consistency, eliminate misunderstandings and clarify questions. The interview guide consisted of four successive parts. The first part presented the initial conceptual model. The second part comprised a mapping between the concepts of the initial conceptual model and constructs of the respective tool to be jointly completed, resulting in a *construction mapping*. Moreover, the mapping asked for any tool constructs that are not covered by concepts of the initial conceptual model. To obtain a holistic understanding of each tool and distinguish it from other tools, the third part of the guide additionally asked for the tools' main use cases, and the supported operations in the fourth part.

Construction Mapping

The construction mappings (one per tool) provided insights regarding each tool, its employed concepts, their relations and used terminology. Mappings between model concepts and tool constructs were obtained based on semantic equivalence and not merely on names, because some tools use the identical name for constructs that map to different model concepts. For example, the term `Variant` represents a `Configuration` in FeatureIDE, and a `Product` in pure::variants and ECCO. Based on insights from the mappings, we improved the initial conceptual model by removing, adding, merging and splitting up concepts. Specifically, in tools dealing with variability in space, the constructs `Feature` and `Constraint` did not correspond to any concept of the initial conceptual model. Moreover, tools that cope with variability in space and time simultaneously do not differentiate between concepts of a `Versioned System` and `Product Line`. Instead, a single construct is used (e.g., `Repository` in ECCO or `Product Line` in DeltaEcore). Finally, many tools use the concept of a `Mapping` (i.e., a relation between `Features` or `Revisions` and `Fragments`) and a `Configuration`. The initial model represents both constructs only implicitly via relations (i.e., the `Mapping` is represented with a directed association relationship between the `Variation Point` and the `Fragment`, and the `Configuration` is represented with a ternary association of the `Product Line`, the `Variation point`, and the `Fragment`).

Workshops

I organized a series of closed and specialized workshops to construct the unified conceptual model based on the insights from the construction mappings. All tool experts as well as further researchers and practitioners that voiced their interest to participate were invited. Specifically, two subsequent workshops took place (Steps ③ and ④). The format of the first workshop was a one-day open discussion with 15 participants. I prepared interview results and impulse questions upon which we gradually refined the initial conceptual model. The format of the second workshop was an online meeting with 12 participants for 1.5 hours. Important cornerstones of the meeting involved a retrospective of the performed changes to the initial conceptual model based on the first workshop, and discussion of remaining open issues. Major agreed changes involved the unification of concepts. Particularly, while tools that cope with either variability in space and time employ a `Product Line` or `Versioned System`, tools that cope with variability in space and time simultaneously use one unified concept to represent both. Moreover, the initial conceptual model specifies the concept of `Configuration` only for variability in space, which is however also present in tools dealing with variability in time [48] (e.g., a commit hash in `Git`). Thus, we extended the `Configuration` to also be able to refer to concepts of variability in time as well, thereby representing a unified concept. Further concepts were added or made explicit based on the construction mappings. Specifically, the concept of a `Constraint` was added to constrain the validity of configurations. The concept of a `Feature` was added to represent design options, since we found the concept `Variation Point` relevant in the implementation only (and, thus, lowering the degree of abstraction). Also, the `Mapping` was modeled explicitly, since this concept carries significance in most of the tools. Finally, hybrid concepts were introduced that were found in tools focusing on both variability dimensions as well as new relations that do not exist in any of the studied tools.

5.2. Unified Conceptual Model

This section presents the unified conceptual model (C1). It starts by describing the main design decisions that impacted the unified model. Then, its concepts and relations, a formal notation and static semantics in the form of well-formedness rules are introduced.

5.2.1. Design Decisions

The design process of the unified conceptual model involved several design decisions during the unification process. In the following, the main design decisions regarding terminology and modeling are presented.

Terminology

The construction mappings revealed that terminology used for concepts of variability in space and time in SPLE and SCM suffers from ambiguity and homonyms. Therefore, we aimed for generic and unambiguous terms not commonly used in either research area. This particularly concerned the term `Variant`. For instance, it corresponds to the `Configuration` construct in FeatureIDE, and to the `Product` construct in ECCO, according to the tool experts. Thus, even within the SPLE community, terms are used inconsistently. Furthermore, the term `Variant` is also used in literature to represent an option of a `Variation Point`. Therefore, this term is avoided and `Product` is used instead. Moreover, the term `Variation Point` is generally associated in the SPLE area with implementation. Therefore, `Option` is used as generic term to describe any variation. Another decision on terminology concerns the system representing variability in space, time, or both. In the initial conceptual model, `Product Line` and `Versioned System` contain most other concepts. While both terms are associated with SPLE and SCM, respectively, tools that cope with both variability dimensions represent the system with a single construct. As the term `Repository` is close to implementation (i.e., persistence), the term `Unified System` is used.

Modeling Pragmatics

The first design decision relates to the structure of `Fragments`. The initial conceptual model allows to form a complex structure with a generalized composite design pattern. Based on the selected tools, some organize `Fragments` as a tree structure (e.g., Git), and some as a graph structure (e.g., DeltaEcore). To cover all tool constructs and relations, the `Fragment` structure is modeled as a graph, which generalizes the tree structure. The second design decision concerns the modeling of different types of revisions. Specifically, some tools employ `Revisions` of the `Unified System` (e.g., Git and SuperMod) while others use revision control of each `Feature` (e.g., DeltaEcore and ECCO). To clearly differentiate between `Revisions` of both concepts and separate the concerns, the concept of `System Revision` and its counterpart `Feature Revision` is introduced. The third design decision relates to `Constraints`. While this concept is common in most tools coping with variability in space to restrain combinations of `Features`, `Constraints` can be formulated on `Feature Revisions` (e.g., DeltaEcore). Therefore, the concept `Feature Option` is introduced as a generalization of the `Feature` and `Feature Revision` concept, and `Constraints` to be defined over `Feature Options`. The fourth design decision concerns the relationship between a `Feature` and its `Feature Revisions`. Since the existence of a `Feature Revision` strongly depends on its `Feature`, the relation is modeled as a composition. The final design decision relates to the versioning of `Configurations` and `Mappings`. However, since both can refer to the same `Options` that would be used for versioning them (e.g., a `System Revision` could enable a `Mapping` and, at the same time, be referred to by this `Mapping`), this would have introduced cycles. Note that abstract classes are used to indicate that they should not be specialized. Instead, the respective sub-classes should be used for further instantiation.

5.2.2. Concepts and Relations

Figure 5.3 shows a UML class diagram of the resulting unified conceptual model. Concepts for variability in space are highlighted in green, concepts for variability in time in orange, concepts of both variability dimensions in purple, and unified concepts are white. Lighter colors and an italic font is used to represent abstract classes. The left part of the model comprises the concepts located in the problem space in SPLE or version space in SCM (i.e., the abstraction of the domain). The right side comprises the concepts located in the solution space in SPLE or product space in SCM (i.e., the implementation artifacts of the system). Notably, mostly all of the unified concepts are located in the solution space or on the border to the problem space, which, in turn, contains the remaining concepts for variability in space, time, or both. In the following, the concepts and relations of each variability dimension and their combination are introduced, gradually describing the model from left to right and referring to the running example of the `Car` system presented in Section 2.1.

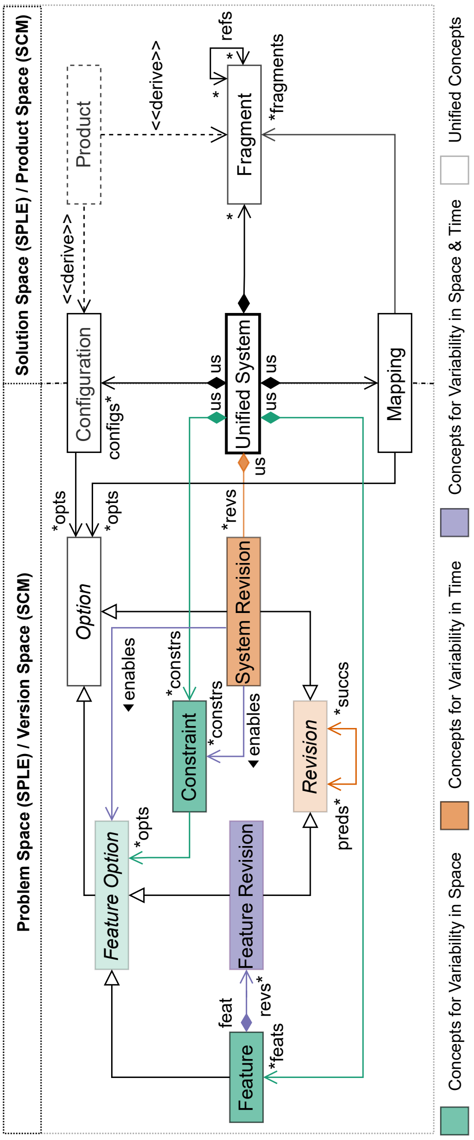


Figure 5.3.: UML class diagram of the unified conceptual model [5, Fig. 4].

Concepts and Relations for Variability in Space

Variability in space is supported by all studied tools except for Git and SVN (note that in this research, branches are considered a temporal divergence for concurrent development only). It is represented by the concepts `Feature`, `Constraint`, and `Feature Option`.

A `Feature` is considered a "prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system" [107, p. 3] that represents a concrete specialization of the abstract concept `Feature Option`. *Example:* In the final feature model of the `Car` system in Figure 2.1b, four concrete features exist: the `Car` feature, the `Gas` feature, the `Ele` feature, and the `Dist` feature.

A `Constraint` is formulated over `Feature Options`. It governs which `Feature Options` can, should, or should not be combined together. *Example:* The `Ele` feature implies the `Dist` feature so that both must be selected in a valid `Configuration`.

Concepts and Relations for Variability in Time

Variability in time is supported by Git, SVN, SuperMod and DarwinSPL. It is represented by the concepts `Revision` and `System Revision`.

The abstract concept `Revision` describes the evolution history and is used by the unified conceptual model as an abstract representation of time. The `Revision` refers to its preceding and successive revisions, forming a revision graph. Multiple directly predecessors and successors represent branches and merges.

A `System Revision` specializes a `Revision` and describes a particular state of the system at one point in time. *Example:* Nine subsequent `System Revisions` describe the evolution history of the `Car` system.

Concepts and Relations for Variability in Space and Time

Concepts for variability in space and time relate to concepts from both variability dimensions. The concept `Feature Revision` was identified as such, which is employed by the tools ECCO, DeltaEcore, and VaVe.

A `Feature Revision` specializes the `Feature Option` and the `Revision`. In contrast to a `System Revision` that represents the state of an entire system, a `Feature Revision` is specific to a single `Feature`. Thus, every feature has its own revision graph. *Example:* In the final `Car` system, the `Gas` feature has three subsequent revisions: $\texttt{Gas}_1$, $\texttt{Gas}_2$, $\texttt{Gas}_3$.

Interestingly, no tool employs both `Feature Revisions` and `System Revisions` simultaneously. Instead, `Feature Revisions` are retrospectively mined from `System Revisions` or the other way round, leading to high additional costs [59, 127]. This missing relation between `System Revisions` and `Feature Revisions` is a gap in state of the art. The combination of both revision types is challenging, since they cannot be managed independently and new structure as well as behavior must be defined with respect to their interaction. To address this problem, `System Revisions` are used in the unified conceptual model to support the management of `Feature Revisions`. Specifically, this research proposes to let a `System Revision` *enable* `Feature Revisions`. Consequently, it is explicitly modeled which `Feature Revisions` relate to a `System Revision`. This enables cross-dimensional analyses, for instance, drawing conclusions about compatibility of `Feature Revisions` or tracking the frequency of feature changes whereby a high frequency might indicate a poor design. *Example:* The first `System Revision` of the `Car` system enables $\texttt{Car}_1$, $\texttt{Gas}_1$ and $\texttt{Ele}_1$.

Unified Concepts

Unified concepts are relevant for either variability dimension and are supported by all studied tools. The `Unified System` (highlighted with a black border) represents the entire variable system that describes variability space, time, or both, such as the `Car` system. It is located on the border of both the problem space and the solution space since it contains concepts located in either space.

The abstract concept `Option` represents any variation of either variability dimension. Thus, it can be specialized as a `Feature`, a `System Revision`, or as a `Feature Revision`.

Analogously to the initial model, a `Fragment` represents an implementation artifact of any level of granularity, e.g., models, delta modules, or entire files. Using a self-reference, the unified conceptual model employs a graph structure of Fragments. *Example:* In the Car system, Fragments are represented by the Java and SysML file as well as by lines of code.

Analogously to the Unified System, the Mapping is also located on the border of both spaces. It relates concepts from the problem space (i.e., Options) with concepts from the solution space (i.e., Fragments). Note that Options can exist that are not present in any Mapping. For instance, such Options could represent abstract features. *Example:* The Fragment which represents Line 10 in the Car system in Figure 2.2 maps to the feature interaction $\texttt{Gasoline}_3\&\&\texttt{Electric}_4$. The abstract feature ET is not referred to by any Mapping.

The concept of a Configuration exists in both SPLE and SCM. While a Configuration is rather complex in SPLE (i.e., the selection of particular Features), in SCM, it is simply represented by a System Revision (e.g., a commit hash in Git). Therefore, a Configuration is unified such that it is a selection of Options. *Example:* $\{\texttt{Car}_1, ET, \texttt{Gas}_1\}$.

A Product is derived by tool-specific variability mechanisms, such as delta modeling, that specify which Fragments are composed according to a Configuration. Since the Unified System is not effected by the Product, it is detached from the system after its derivation.

5.2.3. Notation

The following notation is used to refer to concepts and relations of the unified conceptual model and to provide an example based on the running Car system.

Definition 5.1 *A Unified System US is comprised of a set of System Revisions SR, a set of Features F, a set of Constraints CT, a set of Configurations C, a set of Fragments FT, and a set of Mappings M.*

The Car system in its final state has the set of System Revisions $SR = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the set of Features $F = \{\texttt{Car}, \texttt{ET}, \texttt{Dist}, \texttt{Gas}, \texttt{Ele}\}$, of Constraints $CT = \{\texttt{Car}, \texttt{Car} \Leftrightarrow \texttt{ET}, \texttt{Dist} \Rightarrow \texttt{Car}, \texttt{Gas} \Rightarrow \texttt{ET}, \texttt{Ele} \Rightarrow \texttt{ET}, \texttt{Gas} \lor \texttt{Ele}, \texttt{Ele}_{3,4} \Rightarrow \texttt{Dist}_3\}$, the set of Configurations $C = \{\}$, the set of Fragments $F = \{\texttt{class EngineController}, \texttt{double gasLevel}, ...\}$, and the set of Mappings $M = \{(\texttt{class EngineController}, \texttt{Car}_1), (\texttt{double gasLevel}, \texttt{Gas}_{1,2,3}), ...\}$.

Definition 5.2 *FR_f denotes the set of Feature Revisions of Feature $f \in F$.*

For example, the feature `Gas` has the set of feature revisions $FR_{\texttt{Gas}} = \{\texttt{Gas}_1, \texttt{Gas}_2, \texttt{Gas}_3\}$.

Definition 5.3 *A System Revision $sr \in SR$ enables a set of Features $F_{sr} \subseteq F$, a set of Feature Revisions $FR_{sr,f} \subseteq FR_f$ for every feature $f_{sr} \in F_{sr}$, and a set of Constraints $CT_{sr} \subseteq CT$. The set of all Feature Options enabled by sr is $FO_{sr} = F_{sr} \cup \bigcup_{f \in F_{sr}} FR_{sr,f}$.*

The `System Revision` 9 enables features $F_9 = F = \{\texttt{Car}, \texttt{ET}, \texttt{Dist}, \texttt{Gas}, \texttt{Ele}\}$, `Feature Revisions` of `Gas` $FR_{9,\texttt{Gas}} = \{\texttt{Gas}_3\}$, and `Constraints` $CT_9 = CT$.

Definition 5.4 *$m_{sr,ft} \in M$ is used to denote the mapping of Fragment $ft \in FT$ at System Revision $sr \in SR$. A Mapping can be treated as a Boolean expression over Options. $FT_m \subseteq FT$ denotes the set of Fragments of a Mapping $m \in M$, and $FT_{sr,m} \subseteq FT$ the set of Fragments of a Mapping $m \in M$ at System Revision $sr \in SR$.*

For example, the `Mapping` of `Fragment class EngineController` $\in FT$ at `System Revision` 9 $\in SR$ is denoted as $m_{9,\texttt{class EngineController}} = \texttt{Car}_1$, and the set of `Fragments` that map to $\texttt{Car}_1$ at `System Revision` 9 as $FT_{9,\texttt{Car}_1} = \{\texttt{class EngineController}, \texttt{void doDriving()}\}$.

Definition 5.5 *A Configuration c is a set of positive or negative features. It can be treated as a conjunction of its positive and negative feature literals $\bigwedge_{l \in c} l$.*

An example of a valid `Configuration` at the final `System Revision` is $c_1 = \{\texttt{Car}_1, \texttt{ET}, \texttt{Dist}_3, \texttt{Gas}_3, \neg\texttt{Ele}\}$ which is equivalent to the expression $\texttt{Car}_1 \wedge \texttt{ET} \wedge \texttt{Dist}_3 \wedge \texttt{Gas}_3 \wedge \neg\texttt{Ele}$.

5.2.4. Static Semantics

During interviews and discussions with the tool experts, several constraints defined on tool constructs became obvious. For example, the tool `ECCO` employs an acyclic revision graph. Since the UML notation does not provide the required level of conciseness and expressiveness, the *Object Constraint Language (OCL)* is used to specify the static semantics of the unified conceptual model in a formal way [181]. In the following, auxiliary definitions are introduced that specify well-formedness rules.

```
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 5.1: Auxiliary definitions for well-formedness rules [5, Listing 4].

Auxiliary Definitions

Listing 5.1 comprises the definitions of three auxiliary operations. The first definition specifies the operation `getAllOptions` that returns a set of all `Options` in a `Unified System` (i.e., `System Revisions`, `Features`, and `Feature Revisions`). The second definition specifies the operation `getAllSystemRevisions` for collecting all `System Revisions` in a `Configuration`. The third definition specifies the operation `getAllFeatureOptions` for collecting all `Feature Options` (i.e., `Features` and `Feature Revisions`) in a `Configuration`.

Well-Formedness

In the following, ten well-formedness rules are introduced that specify the static semantics of the unified conceptual model. Listing 5.2 provides three rules that define the well-formedness of the revision graph.

Rule 1 states that every direct predecessor of a `Revision` r must have r as successor, and that every direct successor of r must have r as a predecessor. This rule essentially enforces a bidirectional relationship between predecessor and successor revisions.
Rule 2 defines revision graphs to be directed and acyclic by stating that the transitive closure over the successor revisions of a `Revision` r must exclude the `Revision` r itself. It ensures that no revision can be visited multiple times while traversing a revision graph.

```
context Revision
-- Rule 1: Every predecessor of a Revision must have the Revision as
      successor and vice versa.
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 5.2: Well-formedness of the revision graph [5, Listing 5].

Rule 3 specifies that all `Revisions` in a revision graph must have matching types (i.e., either only `System Revisions` or only `Feature Revisions`) and have the same container (i.e., the `Unified System`, in the case of `System Revisions`, or the same `Feature`, in the case of `Feature Revisions`). A revision graph can thus either only contain `System Revisions` of the same `Unified System` or only `Feature Revisions` of the same `Feature`.

In Listing 5.3, three rules are defined that specify the use of concepts that are contained in the same `Unified System`.

Rule 4 states that all `Options` that a `Configuration` refers to must be part of the same `Unified System` as the `Configuration` itself. A `Configuration` can thus not refer to `Options` that are contained in another `Unified Systems` or in no `Unified System` at all.
Rule 5 defines that all `Options` and `Fragments` to which a `Mapping` refers must be part of the same `Unified System` as the `Mapping` itself. A `Mapping` can thus neither refer to `Options` nor to `Fragments` that are contained in another or in no `Unified System`.

```
-- Rule 4: All Options of a Configuration must be contained in the Unified
     System.
context Configuration
inv:
  self.opts -> forAll(o:Option | self.us.getAllOptions -> includes(o))

-- Rule 5: All Fragments and Options of a Mapping must be contained in the
     Unified System.
context Mapping
inv:
  self.opts -> forAll(o:Option | self.us.getAllOptions -> includes(o))
inv:
  self.fragments -> forAll(f:Fragment | self.us.fragments -> includes(f))

-- Rule 6: All Feature Options of a Constraint must be contained in the
     Unified System.
context Constraint
inv:
  self.opts -> forAll(o:FeatureOption | self.us.getAllOptions -> includes(o))
```

Listing 5.3: Well-formedness of containments in a `Unified System` [5, Listing 6].

Rule 6 specifies that all `Feature Options`, that a `Constraint` refers to, must be part of the same `Unified System` as the `Constraint` itself. A `Constraint` can thus not refer to `Feature Options` contained in another or in no `Unified System`.

Listing 5.4 shows three well-formedness rules of the *enables* relations from `System Revision` to `Feature Option` and to `Constraint`.

Rule 7 states that all `Feature Options` and `Constraints` that a `System Revision` enables must be part of the same `Unified System` as the `System Revision` itself. A `System Revision` cannot enable `Feature Options` or `Constraints` of another `Unified System`.
Rule 8 defines that `Constraints` can only refer to `Feature Options` that are enabled by the same `System Revision` as the `Constraints` itself. This rule is only relevant for tools that support multiple `System Revisions`. It ensures that, at no point in time, a `Constraint` is visible that refers to `Feature Options` that are not visible.
Rule 9 specifies that, for every `Feature` f that has at least one `Feature Revision` and that is enabled by a `System Revision` r, the same `System Revision` r must also enable at least one `Feature Revision` of `Feature` f. This

```
-- Rule 7: All Feature Options and Constraints enabled by a System Revision
     must be contained in the same Unified System as the System Revision.
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: Constraints may only refer to 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 that
     Feature.
inv:
  self.opts -> select(o:Option | o.oclIsTypeOf(Feature)) ->
    forAll(f:Feature |
      f.revs -> isEmpty() or f.revs -> exists(fr:FeatureRevision |
        self.opts -> includes(fr)
      )
    )
```

Listing 5.4: Well-formedness of the *enables* relations of `System Revision` [5, Listing 7].

rule is only relevant for tools that support `System Revisions` and `Feature Revisions` simultaneously. It ensures that, at any point in time where a `Feature` is visible, also at least one of its `Feature Revisions` is visible. Yet, it permits `Features` to be visible that do not have any `Feature Revision`, which is the case either for abstract or newly created `Features` that have not been implemented yet.

Finally, **Rule 10** defines the well-formendess of a `Configuration` in Listing 5.5. In case a `Configuration` refers to at least one `System Revision`, all `Feature Options` that it refers to must be enabled by at least one of the `System Revisions` that it refers to. In case a `Configuration` does not refer to any `System Revision`, it may refer to any `Feature Options`. This rule ensures that only `Feature Options` are selected in a `Configuration` that are visible at one or more points in time. This rule specifies the minimal requirements for the well-formedness of a `Configuration`, without making any statement about completeness or validity of a `Configuration` with respect to a set of `Constraints`.

```
-- Rule 10: A Configuration may only refer to Feature Options that are
     enabled by at least one of the System Revisions it refers to.
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 5.5: Well-formedness of `Configuration` [5, Listing 8].

5.3. Expected Benefits

So far, the unified conceptual model, its unification process, design decision and well-formedness rules have been introduced and described. This section comprises a discussion of the expected benefits of the unified conceptual model.

A model can have one or more roles (i.e., a descriptive, prescriptive, or predictive role) that reflect its purpose [46]. To this end, the unified conceptual model can play a *descriptive* as well as a *prescriptive role*.

On the one hand, it systematically builds on and describes concepts and relations for dealing with variability in space, time, and both of contemporary approaches and tools. Based on this common foundation, researchers and practitioners gain understanding of the state of the art, and can communicate and compare their work based on common and distinguishing concepts and relations of the unified conceptual model.

On the other hand, the unified conceptual model can play a prescriptive role since it does not only cover identified concepts and relations, but also provides meaningful novel relations between concepts that do not appear in combination in any of the studied approaches (i.e., `System Revisions` and `Feature Revisions`). Thus, it prescribes the concepts and relations of a system aiming to deal with variability in space and time simultaneously. Based on that, cross-dimensional variability modeling and analyses become possible, for instance, tracking revisions per features to analyze their frequency of change (for which expensive and approximate mining techniques are currently used).

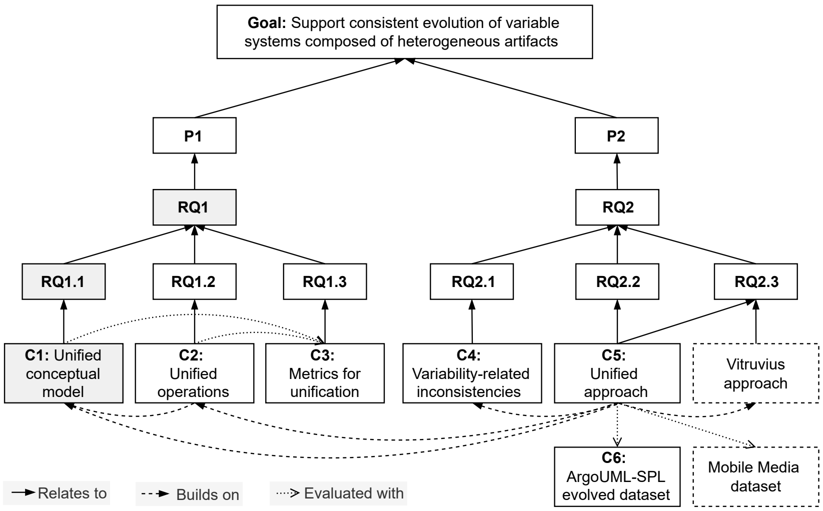

Goal: Support consistent evolution of variable systems composed of heterogeneous artifacts
P1
P2
RQ1
RQ2
RQ1.1
RQ1.2
RQ1.3
RQ2.1
RQ2.2
RQ2.3
C1: Unified conceptual model
C2: Unified operations
C3: Metrics for unification
C4: Variability-related inconsistencies
C5: Unified approach
Vitruvius approach
C6: ArgoUML-SPL evolved dataset
Mobile Media dataset
Relates to
Builds on
Evaluated with

Figure 5.4.: Contribution of Chapter 5 of the thesis.

In conclusion, the unified conceptual model can be employed to drive the construction of a system by providing a systematically devised foundation for dealing with both variability dimensions.

5.4. Summary

This chapter presented the unified conceptual model for uniformly describing and relating concepts of variability in space and time (C1). it comprises concepts of variability in space, variability in time, unified concepts (that can be used to deal with either variability dimension) and hybrid concepts for dealing with both variability dimensions simultaneously.
Major design decisions that impacted terminology and modeling have been explained. In addition, a formal notation of the unified conceptual model and OCL rules to ensure the well-formedness of the unified model have been provided. To support reproducability of this unification effort, the unification process of the unified conceptual model has been documented and related artifacts were made available.

The unified conceptual model provides means for researchers and practitioners to clarify, communicate and compare their work based on the model's concepts and relations. Beyond that, the unified conceptual model can guide the design of novel approaches for managing variability in space and time. To this end, none of the studied tools support the explicit combination of `Feature Revisions` and `System Revisions`. This a gap in tool support for managing both variability dimensions that is currently mitigated by retrospective mining of feature evolution, which leads to additional costs. This research proposes to use `System Revisions` to govern which `Feature Revisions` are available. In conclusion, the unified conceptual model extends the body of knowledge on unified variability management.

Thus, this contribution addresses RQ 1.1. Figure 5.4 shows an overview of all contributions and highlights the contribution of this chapter in grey.

6. Unified Operations

This chapter builds on a publication at VaMoS [6]. An open-access repository comprises all artifacts related to the unification process and evaluation of the unified operations.[1]

This chapter presents the unified operations to manage variability in space and time simultaneously. So far, ten existing tools from the SPLE and SCM communities have been analyzed and a conceptual model for unifying concepts of variability in space and time has been conceived (C1). However, tools operate not only on different data structures, but follow different edit modalities and development paradigms when it comes to managing variability, even if they consider the same variability dimensions [141]. For instance, some tools enforce (partially) contradicting pre-conditions for the same operations. Thus, understanding the different ways of handling both dimensions allows for establishing a common foundation for the operational management of variability in space and time.

Referring to problem statement P1, the following research question is asked:

RQ 1.2 Which operations are provided to cope with either or both variability dimensions in the studied tools and how can they be unified?

First, Section 6.1 presents the specialized unification process. The identification of the commonalities and differences of the individual tool's operations is described in Section 6.2. Then, the unified operations are introduced in Section 6.3, along with their pre and post-conditions, for coping with variability in space and time. Expected benefits of the unified operations are described in Section 6.4. A summary in Section 6.5 closes this chapter.

This chapter thus constitutes the contribution C2.

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

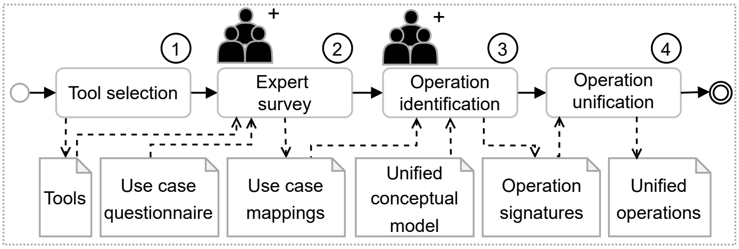


Figure 6.1.: Unification process of the unified operations [6, Fig. 2].

6.1. Specialized Unification Process

Figure 6.1 presents the specialization of the general unification process shown in Figure 4.1 for the unified operations. In the following, each step is described in detail.

Tool Selection

We elicited tools based on the selection criteria described in Section 4.2 (Step ①). Table 6.1 shows the selected tools, and how they differ regarding their supported concepts for variability in space (i.e., `Feature, Constraint`) variability in time (i.e., `System Revision`), and both dimensions (i.e., `Feature Revision`). Note that these tools are mostly the same as the ones selected for the unified conceptual model (see Section 5.1). Tools for variability in space, i.e., FeatureIDE [130, 157], VTS [228] and SiPL [187, 188], support `Features` and, except for VTS, `Constraints`. Tools for variability in time, i.e, SVN [190] and Git [145], support `System Revisions`. Tools that support variability in space and time via `Features`, `Constraints` and `System Revisions` are SuperMod [215, 214] and DarwinSPL [170]. Tools that support variability in space and time via `Feature Revisions` instead of `System Revisions` are DeltaEcore [219, 220], ECCO [75, 73, 139] and VaVe [10]. Again, combinations of tools (e.g., FeatureIDE and Git) are not considered as they do not provide dedicated functionality for dealing with both variability dimensions simultaneously. A more detailed description of the tools for variability in space is provided in Section 2.2.2, of the tools for variability in time in Section 2.3.2, and of tools coping with both dimensions in Section 2.4.2.

Table 6.1.: Distinguishing concepts of the selected tools [6, Tab. 1].

Tool \ Concept	Feature	Constraint	System Revision	Feature Revision
FeatureIDE	●	●	–	–
VTS	●	–	–	–
SiPL	●	●	–	–
SVN	–	–	●	–
Git	–	–	●	–
SuperMod	●	●	●	–
DarwinSPL	●	●	●	–
DeltaEcore	●	●	–	●
ECCO	●	–	–	●
VaVe	●	●	–	●

Expert Survey

In Step ②, I conducted an expert survey based on questionnaires comprising an initial set of use cases. Experts were invited that are among the most knowledgeable people of an elicited tool, all being researchers from academia. Analogously to the unification process of the unified conceptual model (see Figure 5.1), experts for `SVN` or `Git` were not invited as both tools are widely used with profound documentation available. Per tool, one questionnaire was completed by an expert. The goal was to obtain a mapping between a set of use cases presented in the questionnaire and the functionality (i.e., operations) provided by a respective tool. For each use case, the questionnaire therefore asked for its input and output, pre and post-condition and a description of its semantics. Furthermore, the questionnaire asked for further functionality related to the management of variability in space, time, or both that was not covered by the initial set of use cases.

Use cases were formulated on the abstraction level of the unified conceptual model, which excludes tool-specific operations related to, for example, delta modules or feature models. Moreover, use cases were considered that modify a unified system and produce non-trivial, mutable output from a `Unified System`. This excludes operations for variability analysis that compute a Boolean or integer value as well as run-time variability which operates on a running product.

Figure 6.2 depicts the set of selected use cases. User-facing user-goal use cases (white) and sub-function use cases (grey) are differentiated. In the following, non-trivial use cases are briefly described. Every use case is always executed in the context of a `Unified System`.

- *Add Unified System*: Adds another `Unified System` to this `Unified System`. This entails adding new `Features`, `Revisions`, `Mappings`, `Fragments` and/or `Constraints`. For the sake of clarity, all include-relationships between this use case and included add/update/delete use cases are omitted.
- *Add Product*: Adds a `Product` to this `Unified System`. This entails adding new `Features` of the `Product` and/or new `Revisions` of changed `Features` of the `Product` and/or new `Fragments` in the `Product` to this `Unified System`, including the corresponding addition or update of `Mappings`. For the sake of clarity, include-relationships between this use case and included add/update/delete use cases are omitted.
- *Derive Product*: Derives a `Product` from this `Unified System` based on a valid and complete `Configuration`.
- *Derive Unified System*: Derives a `Unified System` from this `Unified System`. `Features`, `Fragments`, `Mappings`, and `Constraints` of the derived `Unified System` are a subset of the ones of this `Unified System`.
- *Check Well-formedness*: Checks the well-formedness of the given `Fragments`. The precise realization of this use case depends on what the `Fragment` represent. For example, in the case of Java source code, this use case checks for syntactical validity.
- *Check Expression Validity*: Checks the validity of a given expression with respect to the `Constraints` in the `Unified System`. If the expression represents a `Configuration`, this is a special case of this use case and equivalent to the use case *Check Configuration Validity*
- *Check Configuration Completeness*: Checks the completeness of a given `Configuration` considering its selected or deselected `Options`.
- *Select Mappings*: Selects a subset of the `Mappings` in the `Unified System` given an expression against which the expression in each `Mapping` is evaluated.

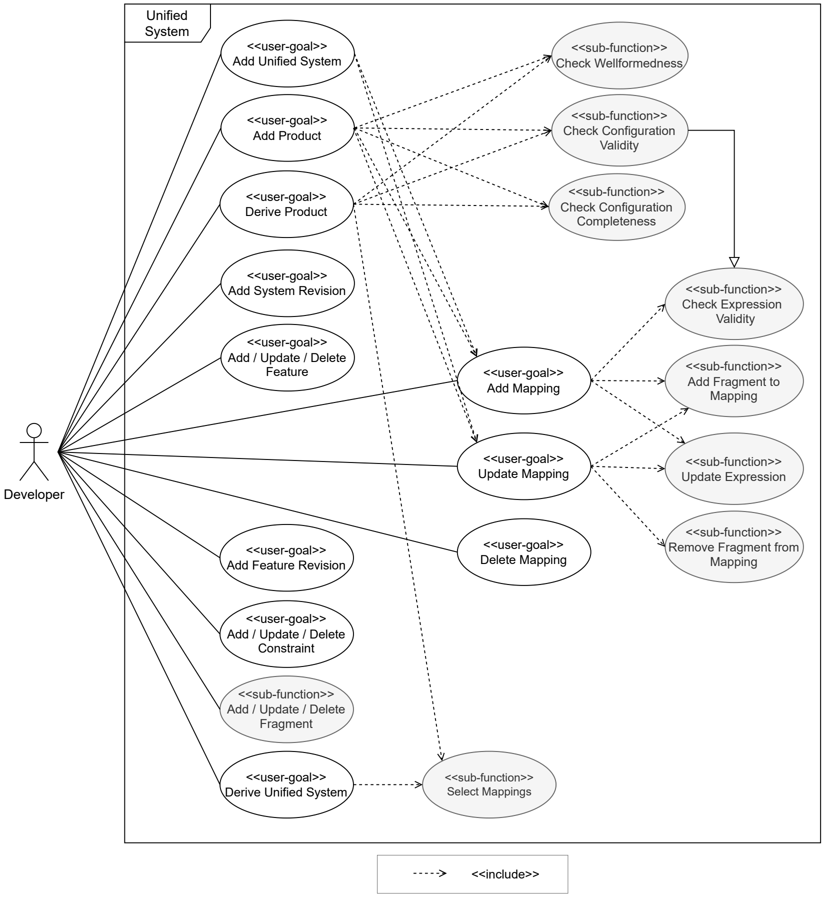

Unified System
Developer
<<user-goal>> Add Unified System
<<user-goal>> Add Product
<<user-goal>> Derive Product
<<user-goal>> Add System Revision
<<user-goal>> Add / Update / Delete Feature
<<user-goal>> Add Feature Revision
<<user-goal>> Add / Update / Delete Constraint
<<sub-function>> Add / Update / Delete Fragment
<<user-goal>> Derive Unified System
<<sub-function>> Select Mappings
<<user-goal>> Add Mapping
<<user-goal>> Update Mapping
<<user-goal>> Delete Mapping
<<sub-function>> Check Wellformedness
<<sub-function>> Check Configuration Validity
<<sub-function>> Check Configuration Completeness
<<sub-function>> Check Expression Validity
<<sub-function>> Add Fragment to Mapping
<<sub-function>> Update Expression
<<sub-function>> Remove Fragment from Mapping
<<include>>

Figure 6.2.: Use cases for expert survey [6].

Use Case Mappings

The use case mappings (one per tool) provided insights regarding each tool, its operations, their semantics and used terminology. The mapping was created by the tool experts based on semantic equivalence of the described use cases and the individual tool operations. The completed use case mappings showed that every use case is covered by at least one tool. Furthermore, depending on the tool, use cases could be user-facing (e.g., *Add Mapping* is user-facing in DeltaEcore, but not in ECCO), or may include other use cases as depicted in Figure 6.2 (e.g., in VTS, the use case *Add Fragment to Mapping* includes the use case *Add Fragment*, since `Mappings` contain the `Fragments`. Moreover, no use cases were missing that would have met the scope criteria. Finally, the used terminology in the use cases introduced ambiguities. For instance, *Add Product* could either be interpreted in a product-oriented way in the sense of clone-and-own (as it is in case of ECCO) or in a platform-oriented way (as it is in case of SuperMod, where the user provides an *ambition* to explicitly map changes to a `Feature` expression).

Operation Identification

The completed use case mappings were input to the identification of operations that deal with variability in space, time, or both (Step ③). Specifically, I evaluated the mappings and mapped the inputs and outputs of the individual tool operations to the concepts of the unified conceptual model (C1). Then, I classified each operation according to common and distinguishing properties and categorized the tool's operations based on a clear and self-contained concern to avoid redundancies and ambiguities and discussed the results with the tool experts. For instance, one category was defined for the integration of changes performed on a product while another category was defined for the integration of changes performed on an instance of a unified system (i.e., a partial product line). *Predicates*, that are used in pre and post-conditions of tool operations, were identified by the same means.

Operation Unification

The identification of each individual tool's operations along with the specified categories were input to the unification (Step ④). For each category, we specified its operation signature based on the name, inputs and outputs. Furthermore, the behavior of operations was unified such that the functionality of studied tools was preserved. Moreover, each operation was extended to cope with both variability dimensions if only one dimension was supported. The same unification procedure was applied for predicates. Afterwards, I presented the resulting unified operations to the tool experts to avoid misunderstandings and make sure the individual tools are covered correctly.

6.2. Identification of Operations

This section presents the identified predicates and operations (Step ③ in Figure 6.1). It starts with a classification to distinguish commonalities and differences of operations.

6.2.1. Classification

When examining the individual tool operations based on the completed use case mappings, commonalities and differences among the tool's functionality became obvious. While, for instance, the derivation of a `Product` is supported by all tools in a user-facing way, some tools employ functionality that is non user-facing but triggered through user-facing actions. For example, the addition of a `Feature` and its `Fragments` in ECCO would require a user to perform changes on a `Product` and commit them, while the `Mapping` is generated fully automatically without the user being able to modify it directly. Contrary, in DeltaEcore, the user would need to manually provide the `Mapping` of the new `Feature` (i.e., by directly modifying the `Mapping Model` of DeltaEcore). Based on insights and discussions with the tool experts, the identified operations could be classified by means of two aspects: the *edit modality* and the *development paradigm*. The edit modality is realized by either direct or view-based editing and describes the way in which a `Unified System` can be modified. Direct editing allows the user to directly modify any part of the system, while view-based editing only allows modifications to the system through views [22]. The development paradigm is realized by either product-oriented or platform-oriented development and describes the way in which a `Unified System` is developed. In platform-oriented development, the user needs to be aware of the entire platform when evolving the system. Contrary, in product-oriented development, the user evolves the system based on a single `Product` without considering the entire platform. While the former represents traditional SPLE, the latter is closer to clone-and-own [73, 198].

6.2.2. Predicates

Predicates are used in pre and post-conditions that evaluate to true or to false. Table 6.2 presents a categorization of the four identified predicates.

Table 6.2.: Categorization of predicates [6, Tab. 2].

Predicate \ Tool	Feature-IDE	VTS	SiPL	SVN	Git	Super-Mod	Darwin-SPL	Delta-Ecore	ECCO	VaVe
Complete Configuration	●	–	●	●	●	●	●	●	–	●
Valid Configuration	●	–	●	●	●	●	●	●	–	●
Well-formed Product	–	–	–	–	–	●	●	●	–	–
Valid Expression[1]	–	–	–	–	–	–	–	–	–	–

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

The predicate *Complete Configuration* is supported by all tools except for VTS and ECCO. It checks whether all `Options` employed by a tool (e.g., `Features` in FeatureIDE or `Features` and `Feature Revisions` in DeltaEcore) are bound in a `Configuration`. The predicate *Valid Configuration* is supported by the same tools and checks whether a `Configuration` violates any `Constraints`. The predicate *Well-formed Product* is supported by SuperMod, DarwinSPL and `DeltaEcore`. It checks whether a set of `Fragments` satisfies a given set of rules that specify the well-formedness of certain types of `Fragments`. For instance, the syntactical validity of Java source code or the conformance between a model and metamodel. The predicate *Valid Expression* checks if a propositional expression over `Feature Options` contradicts any `Constraints`. Note that this predicate is not employed by any tool, but was defined retroactively as it is required for the unified operations, as explained later.

6.2.3. Direct Editing Operations

Table 6.3 presents a categorization of the 21 identified direct editing operations. For every concept contained in a `Unified System`, there is an add, an update, and a delete operation. Since these operations could be found only in tools that employ platform-oriented development (i.e., FeatureIDE, VTS, SiPL, DarwinSPL and DeltaEcore), all direct editing operations are classified as platform-oriented. Tools that only deal with variability in space (i.e., FeatureIDE, VTS, SiPL) allow the user to add, update, and delete instances of their concepts for variability in space. Tools that only deal with variability in time (i.e., SVN, Git) do not support direct editing at all. Tools that deal with both variability dimensions support direct editing in varying degrees: SuperMod, ECCO, and VaVe do not allow direct editing at all. DarwinSPL allows to directly modify only concepts for

Table 6.3.: Categorization of direct editing operations [6, Tab. 3].

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

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

variability in space and only permits to directly add a `Feature` and `Constraint` but not to update or delete it in order to guarantee a reproducible past. Finally, `DeltaEcore`, as the only tool, permits the direct editing of concepts of both dimensions. Note that direct editing operations can be considered atomic operations without complex behavior and thus do not need unification.

6.2.4. View-Based Operations

Table 6.4 presents a categorization of seven view-based operations (one operation per category) that are classified according to the supported development paradigms. While *Externalize* operations create an editable view of the `Unified System`, *Internalize* operations are performed on a view and modify the `Unified System`. Note that names common for operations in SPLE and SCM (e.g., *derivation* or *checkout*) were intentionally not used, but instead the terminology proposed in a recent survey of VarCS [141] (see Definition 2.2). In contrast to direct editing operations, view-based operations offer a higher degree of automation by essentially executing predefined sequences of direct-editing operations to ensure the system's integrity. For instance, adding a `Feature` leads to the automated creation of a new `Mapping` and `System Revision`. View-based operations are used in tools that employ platform-oriented development, product-oriented development, or both. The operation

Externalize Domain derives a set of `Feature Options` and `Constraints` from the `Unified System`. It is supported by SuperMod and DarwinSPL for retrieving a feature model. A changed domain can be integrated into the system via *Internalize Domain*, which is only supported via *commit* in SuperMod. Both operations are part of platform-oriented development. The operation *Externalize Product* retrieves a `Product` from the `Unified System` and is the only operation supported across all tools. A modified `Product` can be integrated either via the product-oriented operation *Internalize Product* (e.g., *commit* in Git or ECCO), or the platform-oriented operation *Internalize Changes* (e.g., *commit* in SuperMod, that additionally requires an *ambition* which maps `Features` and changed `Fragments`). The operations *Externalize Unified System* and *Internalize Unified System* are employed by Git and ECCO and support distributed development. While the first operation creates a new instance of a `Unified System` (e.g., *clone* in Git or ECCO), the second operation integrates another instance of a `Unified System` (e.g, *pull/push* in Git or ECCO). Both operations are part of both development paradigms.
Note that the *branch* and *merge* operations of the tool Git are not in scope, as they do not satisfy the scope criteria. The *merge* operation only modifies the `Product` (i.e., working copy in Git) and not the `Unified System` (i.e., repository in Git), as it does not create a merge point in a revision graph. Instead, the *commit* operation, that follows a *merge* operation, creates the actual merge point in a revision graph. The *branch* operation neither modifies the `Product` nor the `Unified System`, as it only assigns a label to a `commit` and does not create a branch point in a revision graph. Instead, again the following *commit* operation creates the actual branch point.

For each of the seven view-based operations, there is at least one tool that supports it. One view-based operation (*Externalize Product*) is supported by all tools. None of the tools support all view-based operations or implements any view-based operation for both `System Revisions` and `Feature Revisions`. Furthermore, some tools provide a single operation to address multiple concerns. For instance, in VTS, the operation *get* can be used to derive a `Product` or a product line depending on whether the provided `Configuration` is *complete* or *partial*. Following the design principle of separation of concerns (i.e., one operation per concern), the individual tool operations were factorized based on their concerns.

Table 6.4.: Categorization of view-based operations [6, Tab. 4].

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

6.3. Unification of Operations

This section presents the unified predicates and operations for managing variability in space and time. The main commonalities and differences among the tools are discussed and main insights and design decisions of the unification explained (Step ④ in Figure 6.1).

6.3.1. Predicates

Predicates are Boolean properties of concepts. They can be used on their own or in pre and post-conditions of operations. Figure 6.3 provides definitions of the unified predicates.

Predicate: Complete Configuration. Evaluates for a given `Configuration`, whether all `Options` (i.e., `Features`, `Feature Revisions`, `System Revisions`) of a `Unified System` are bound. The definition of a *Complete* (or full) *Configuration* differs based on whether the tool supports variability in space, variability in time, or both. While the semantics of a complete configuration are well understood for variability in space and uniformly used in SPLE, the semantics of completeness of a configuration over time (including `System Revisions` or `Feature Revisions`) is not obvious. A configuration that is not complete is also referred to as a *partial* configuration.

Comparison: All tools evaluate the completeness of a `Configuration` except for VTS and ECCO, which simply assume any not explicitly selected `Feature` or `Feature Revision` as deselected. In tools coping with variability in space (i.e., FeatureIDE, SiPL), a configuration is complete if it either selects or deselects every `Feature` in a `Unified System`. In tools coping with variability in time (i.e, SVN, Git), a `Configuration` (a `revision number` or a `commit hash`) is complete if it selects exactly one `System Revision`. In tools coping with both variability dimensions via `Feature Revisions` (i.e., DeltaEcore, ECCO, VaVe), a configuration is complete if it either selects or deselects every `Feature` and, for every selected `Feature`, it selects exactly one `Feature Revision`. ECCO is an exception in this regard, as it allows a complete configuration to select multiple `Feature Revisions` for every selected `Feature` for merging. In tools coping with both variability dimensions via `System Revisions` and `Features` (i.e., SuperMod, DarwinSPL), a configuration is complete if it selects exactly one `System Revision` and all of its enabled `Features` are either selected or deselected.

complete

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

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

valid

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

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

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

validExpr

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

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

wellformed

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

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

Figure 6.3.: Overview of the predicates [6, Fig. 3].

Unification: The semantics of a complete `Configuration` in either space, time, or both do not contradict each other and can be unified straightforward as presented in Figure 6.3. Note that the condition of selecting *exactly* one `Revision` per `Feature` to *at least* one `Revision` is relaxed to allow merging of `Feature Revisions`.

Predicate: Valid Configuration. Evaluates whether a `Configuration` contradicts any `Constraints` in a `Unified System`. A *Valid Configuration* neither contradicts any explicitly specified `Constraints` (e.g., via a variability model) nor any implicit `Constraints` (e.g., different `System Revision` exclude each other). Analogously to a *Complete Configuration*, the semantics of a *Valid Configuration* are well-understood for variability in space in the area of SPLE, but not so obvious with variability in time.

Comparison: Except for `VTS` and `ECCO`, which do not employ `Constraints`, all considered tools evaluate the validity of a `Configuration`. In the analyzed tools coping with variability in time, a `Configuration` is valid if the `Unified System` contains the selected `System Revision`. In tools coping with variability in space or with both variability dimensions via `Feature Revisions`, a `Configuration` is valid if it does not contradict any of the `Constraints` in the `Unified System`. In tools coping with both variability dimensions via `Features` and `System Revisions`, a `Configuration` is valid with respect to a selected `System Revision`, if all selected `Features` are enabled, and none of the enabled `Constraints` are contradicted.

Unification: The above definitions of a *Valid Configuration* align well and can be combined in a unified definition considering variability in space and time via `Features`, `Feature Revisions`, and `System Revisions`, as presented in Figure 6.3. Note that the unified definition of a *Valid Configuration* can be applied to complete or to partial `Configurations`.

Predicate: Valid Expression. Evaluates whether an expression over `Feature Options` in a `Unified System` contradicts any `Constraints`. A *Valid Expression* generalizes the *Valid Configuration* predicate from a `Configuration` (i.e., conjunction of `Options`) to arbitrary expressions over `Feature Options`.

Comparison and Unification: This predicate is not used by any of the analyzed tools, but is required for the unified operation *iC*. Figure 6.3 presents the definition of the *Valid Expression* predicate.

Predicate: Well-Formed Product. Evaluates whether the implementation of a `Product` (i.e., a set of `Fragments`) is well-formed based on a set of rules that are specific to the `Fragments`. For instance, this involves the validation of OCL constraints, the conformance of a UML model with the corresponding metamodel or the syntactic validity of Java code.

Comparison: SuperMod, DarwinSPL, and DeltaEcore perform well-formedness checks of `Products`, such as checking the conformance of a model to its corresponding metamodel.

Unification: For the unification, well-formedness is defined generically (i.e., without considering specifics of certain types of `Fragments`) and on the level of abstraction of the unified conceptual model, where the only available information in this regard is the set of `Fragments` and how they reference each other. Figure 6.3 shows the unified definition of the *Well-Formed Product* predicate.

6.3.2. Operations

Figure 6.4 depicts possible execution sequences of the unified view-based operations in the form of a UML state chart diagram. States represent currently available concept instances. Transitions represent operation executions on the instances. Operations are highlighted based on the supported development paradigm: Operations for product-oriented development are highlighted in orange, while operations for platform-oriented development are highlighted in yellow. Finally, operations are highlighted that are used in both paradigms in blue. A remote `Unified System` can be used to externalize a local instance of a `Unified System` via *eUS*. To modify the domain of the variable system, `Feature Options` (i.e., `Features` and `Feature Revisions`) and `Constraints` must first be obtained via *eD*, which provides a clean (i.e., unmodified) view on the domain. While the view remains clean, the *eD* operation can be used to switch to a view on the domain at another point in time. Once the view has been modified (e.g., by adding a new *optional* `Feature`) it is marked as dirty (i.e., modified). Changes to the domain view are integrated back into the local `Unified System` via *iD*. Analogously, to modify the implementation of the local `Unified System`, a `Product` is first externalized via *eP*. Again, *eP* can be repeatedly performed to switch between different `Products` as long as it remains unmodified (i.e., clean). Editing a `Product` marks it as dirty. There are two options for integrating a changed `Product` into the local `Unified System`. In product-oriented development, *iP* can be used to internalize an entire `Product`, and in platform-oriented development, *iC* can be used to internalize the changes performed in a `Product`. In contrast to *iP*, *iC* requires an expression provided by the user to integrate changes in a fine-grained manner, leading again to a clean product. Note that an *iC* operation must

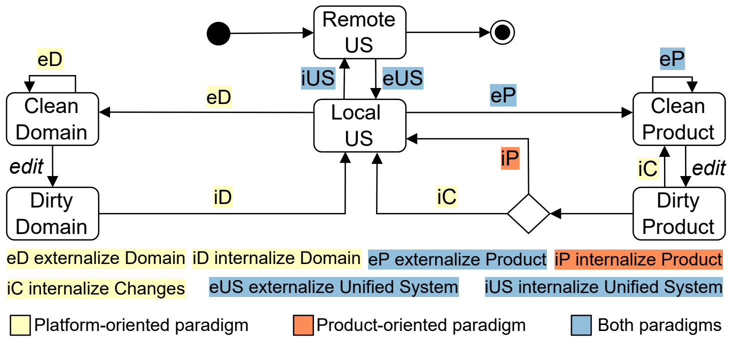


Figure 6.4.: Execution sequences of unified operations [6, Fig. 4].

eD

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

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

Figure 6.5.: Definition of Externalize Domain operation [6, Fig. 5].

be performed once per feature or feature interaction. Consequently, the edit and *iC* cycle can be performed repeatably as long as changes affect the same `Product`. Finally, either the entire local `Unified System` or parts of it (e.g., a new `Feature`) can be integrated into the remote `Unified System` via *iUS*.

In the following, each operation is discussed.

Operation: Externalize Domain (*eD*). This platform-oriented operation creates a view on the domain of a `Unified System`, for instance, in the form of a variability model. The view comprises the `Feature Options` and `Constraints` of the `Unified System` that are visible at one or more specified points in time, i.e., that are enabled by at least one of potentially multiple given `System Revisions`. The operation can be used to edit the domain, merge domains at different points in time or to create configurations.

Comparison: This operation is supported by SuperMod, where it is part of the *checkout* operation, and by DarwinSPL via its operation *getCopyOfValidModel*.

iD

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

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

Figure 6.6.: Definition of Internalize Domain operation [6, Fig. 5].

Both tools equally consider both variability dimensions via `Features` and `System Revisions` and behave coinciding.

Unification: Consequently, the unification of *eD* is the behavior of the tools as defined in Figure 6.5. Note that in the unification, the produced domain view also comprises `Feature Revisions` in addition to `Features`. Moreover, multiple `System Revisions` can be specified as input in order to be able to merge domain views and create merge points in the `System Revision` graph.

Operation: Internalize Domain (*iD*). This platform-oriented operation integrates edits on a domain view (created via *eD*) into the `Unified System`.

Comparison: This operation is supported by SuperMod via its *commit* operation. Although DarwinSPL supports the generation of domain views, it does not offer this view-based operation for internalizing a changed domain view. Instead, DarwinSPL offers direct editing operations for the domain where the user must specify the enabling `System Revision` for every `Feature` and `Constraint` manually.

Unification: For the unification, the behavior is extended to consider `Feature Revisions` by generalizing from `Features` to `Feature Options` (that also include `Feature Revisions`), as defined in Figure 6.6. However, the user is intentionally not allowed to add new `Feature Revisions` to a domain view, only new `Features`. Thus, `Constraints` may only be formulated using `Features` or the `Feature Revisions` that are visible in the externalized domain view. Furthermore, the behavior is extended to support distributed development: The *iD* operation creates a new `System Revision` sr' that becomes the successor eP

Operation:	**Externalize Product**
Input:	`Unified System` US, `Configuration` c
Pre-condition:	valid(US, c) $\wedge$ complete(US,c)
Output:	`Product` p
Post-condition:	wellformed(p)

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

Figure 6.7.: Definition of Externalize Product operation [6, Fig. 5].

of all `System Revisions` specified for eD. If a `System Revision` sr was specified for eD that already has a successor, adding the new `System Revision` sr' as another successor makes sr a branch point in the revision graph. If multiple `System Revisions` were specified for eD, sr' becomes successor to all of them and thus a merge point in the revision graph.

Operation: Externalize Product (*eP*). This operation creates a `Product` that consists of `Fragments` contained in the `Unified System`. A `Product` is externalized based on a complete and valid `Configuration`. The operation *eP* is part of both development paradigms.

Comparison: This operation is supported by *all* of the studied tools. Furthermore, its behavior is essentially the same across all tools. All `Mappings` of the `Unified System` are evaluated and, in case their expression is satisfied by the `Configuration`, selected. Based on the selected `Mappings`, the respective `Fragments` are used to generate the `Product`. The individual tool operations differ with respect to the `Options` that are used in `Configurations` and `Mappings`, as the available `Options` depend on the supported variability dimensions. Except for SVN, Git, ECCO and VTS, which do not include `Constraints`, all considered tools evaluate the completeness and validity of a `Configuration` as a pre-condition. Tools coping with variability in space often refer to this operation as *product derivation*. In tools coping with variability in time, it is referred to as *checkout*. In SVN or Git, that support product-oriented development and solely the time dimension, only previously internalized `Products` can be externalized, as `Options` are realized only by `System Revisions`, of which only one (or multiple for merging) can be specified in a valid configuration. This is referred to as *extensional versioning* [48]). Tools iP

Operation: **Internalize Product**
Input: `Unified System` US, `Product` p
Pre-condition: $\text{wellformed}(p) \wedge \text{valid}(US, c_p)$

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

Figure 6.8.: Definition of Internalize Product operation [6, Fig. 5].

that support platform-oriented development and variability in space (e.g., via direct editing in FeatureIDE or the *iC* operation such as the *commit* operation in SuperMod), can also externalize `Products` with `Configurations` that have not previously been internalized, as `Options` are at least realized by `Features`, of which multiple can be combined in valid and complete `Configurations`. This is referred to as intensional versioning [48]). An exception is ECCO, which neither supports direct editing nor the *iC* operation, but still supports intensional versioning. ECCO performs *feature location* [160] to compute `Mappings` and locate relevant `Fragments` from previously internalized `Products` when externalizing a `Product`.

Unification: Figure 6.7 shows the definition of the unified operation *eP*. The feature location technique of ECCO does not conflict with the behavior of the other tools, and can therefore be optionally included in the *eP* operation. This would enable intensional versioning for product-oriented development.

Operation: Internalize Product (*iP*). This product-oriented operation supports a development process similar to clone-and-own for integrating a `Product` into the `Unified System`. Specifically, edits are performed on an externalized `Product`.

Comparison: This operation is realized by all tools that support product-oriented development, i.e., SVN, Git, and ECCO. As common behavior among the tools, a new `System Revision` sr' is created and mapped to the `Fragments` of the internalized Product. Among the three tools, ECCO supports both variability dimensions via Features and Feature Revisions, which extends its behavior compared to Git and SVN. To indicate the modified Features for which a new Feature Revision must be created, they must be manually marked in a Product's Configuration.

Unification: The unified operation combines the tool behaviors in a fairly straightforward manner, as defined in Figure 6.8. In all cases, a new System Revision is created and mapped to all Fragments of the Product. The expression in the Mapping is automatically set to *true*. Additionally, the new System Revision enables all Feature Options that were selected in the Configuration of the internalized Product. Consequently, *iP* explicitly tracks the Feature Revisions. This is not the case in SVN and Git, where Feature Options are not tracked. Therefore, in practice, other methods are often used in conjunction, such as the use of a preprocessor to mark Features, the manual documentation of modified Features in the commit message, or approaches for retroactively mining variability information. Furthermore, *iP* can create branch and merge points in revision graphs. If the Configuration of an externalized Product contains multiple Revision of a Feature for which a new Feature Revision was created, the new Feature Revision becomes a successor to all of them and thus a merge point in its revision graph. In case the System Revision or a Feature Revision of a modified Feature in the Product's Configuration already have a successor, these Revisions receive yet another successor and thus become branch points in their respective revision graphs.

Operation: Internalize Changes (*iC*). This platform-oriented operation integrates changes applied to an externalized Product back into the Unified System based on an expression over Feature Options that a user provides manually to indicate what Feature Options should be affected by the changes. In practice, this operation is performed *in place*, i.e., the current instance of the Unified System is updated instead of creating a new instance. Its behavior is closer to SPLE (i.e., support for modifying the product-line platform based on individual features) than it is to clone-and-own (see *iP*).

Comparison: This operation is supported by the tools SuperMod, VTS, and VaVe that each require the user to provide an expression over Features that is referred to as *ambition* by SuperMod and VTS. The expression can be a partial Configuration (i.e., a conjunction of selected and deselected Features) iC

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

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

Figure 6.9.: Definition of Internalize Changes operation [6, Fig. 5].

like in SuperMod, an arbitrary expression over `Features` like in VTS, or consist of a single `Feature` like in VaVe. SuperMod and VaVe employ the *Valid Configuration* predicate to ensure that the expression does not violate any `Constraints`. In VTS, the validity of the expression is not evaluated as it does not support `Constraints`. Another interesting case related to pre-conditions is the relation between the configuration of the provided product p and the provided expression e. While in VTS, the expression e given to *iC* must imply the `Configuration` c_p of `Product` p ($e \Rightarrow c_p$), this is exactly the opposite in SuperMod ($c_p \Rightarrow e$). The rationale for the latter is that changes applied to a `Product` must be visible *at least* in the `Configuration` of the `Product` on which they were performed. This ensures that, for example, users cannot specify an expression such that changes performed in a `Product` would affect a `Feature` that is not even part of that `Product`. In VTS, the rationale of the pre-condition is that changes must not affect `Configurations` other than the one of the `Product` on which the changes were performed, thus following consistency principles derived from the *lens laws* [76]. This condition is only sensible if the view is not a `Product` based on a complete `Configuration` (as is the case for this operation), but a view on a partial `Configuration`.

Unification: Figure 6.9 shows the definition of the unified operation iC. Since eP is defined such that a view is a `Product` based on a complete `Configuration` (and not a subset `Unified System`), the pre-condition of VTS can be reduced to $e \Leftrightarrow c$. This strongly limits the effect of an edit, as it could only affect the exact view (i.e., `Product`) in which it was performed. Therefore, this pre-condition is excluded from the unification of this operation. The pre-condition of SuperMod ($c_p \Rightarrow e$) is used in the unification as is. Finally, to be able to combine the pre-condition of SuperMod and VaVe (where the expression is a partial `Configuration` that must not violate any `Constraints`) with the behavior of VTS (where the expression can be of arbitrary form), the new *Valid Expression* predicate is used. Note that the expression is extended from being formulated only on `Features` to `Feature Options` to let the user decide whether or not new `Feature Revisions` shall be created when performing iC. In case no `Feature Revision` of a `Feature` appearing in the expression is provided, a new `Feature Revision` is created for that `Feature`. In case the user provides the same `Feature Revision` as is selected in the configuration of the externalized `Product`, no new `Feature Revision` is created for the respective `Feature`. Other feature revisions cannot be specified in the expression. This addresses interesting corner cases, such as the creation of a new `Feature Revision` only for one of the `Features` of a feature interaction. Beyond that, the intention and behavior is essentially the same across the three tools. Each tool first computes the `Fragments` FT^+ that were added, the `Fragments` FT^o that remained unchanged, and the `Fragments` FT^- that were removed from the `Product`. Then, the `Mappings` for the added, unmodified and removed `Fragments` FT^+, FT^o and FT^- are computed by SuperMod, VTS and VaVe. Analogously to iP, a new `Feature Revision` becomes a merge point in its revision graph if more than one `Feature Revision` of the same `Feature` were selected in the `Configuration` of the externalized `Product`, as it becomes the successor to all of them. A `Feature Revision` in the `Product`'s `Configuration` becomes a branch point if it already has a successor when receiving a new `Feature Revision` as successor. A `System Revision` in the `Product`'s `Configuration` becomes a branch point if it already has a successor.

Operation: Externalize Unified System (*eUS*). This operation derives a `Unified System` US' that is a subset of the original `Unified System` US. The eUS

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

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

Figure 6.10.: Definition of Externalize Unified System operation [6, Fig. 5].

derived `Unified System` US' is a full clone of the `Unified System` US if the given `Configuration` c is empty, and a partial clone otherwise.

Comparison: This operation is supported by FeatureIDE, VTS, Git, and ECCO. The behavior of tools dealing with variability in space (FeatureIDE, VTS, ECCO) coincides: `Feature Options` that are neither selected nor deselected in the partial configuration (and, thus, have no value assigned) remain variable and are essentially copied to the new `Unified System`. Selected `Feature Options` are retained and set to *true*, so all `Mappings` and corresponding `Fragments` where the `Feature Option` appears negated are not retained, and the positive `Features` and `Feature Revisions` are substituted by *true* in `Mapping` expressions and `Constraints`. Deselected `Feature Options` are removed and replaced by *false* in expressions of `Mappings` and `Constraints`, together with all elements (`Fragments` or `Mappings`) that require the `Feature` to be selected. `Mappings` (including the corresponding `Fragments`) whose expression cannot be satisfied anymore (i.e., contradicts configuration c) are removed. As the only tool for variability in time that supports eUS, Git applies additional behavior regarding ancestors of `Revisions` selected in the `Configuration`. It creates a full clone of a `Unified System` if no `System Revision` is selected, and a partial clone otherwise, by only keeping selected `System Revisions` along with their ancestors.

Unification: Figure 6.10 shows the definition of the unified operation eUS. All tools exhibit essentially the same behavior when it comes to `Features`, `Mappings` and `Fragments`. While only FeatureIDE supports `Constraints`, its behavior is consistent with the desired semantics of this operation (i.e., to obtain a subset of a `Unified System` by filtering unneeded `Options` based on a partial `Configuration`) and can thus also be transferred directly to the unification.

iUS

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

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

Figure 6.11.: Definition of Internalize Unified System operation [6, Fig. 5].

Unifying the tools' behavior of dealing with `Feature Revisions` (ECCO) and `System Revisions` (Git) is less straightforward, as it is not yet supported by any of the existing tools. Specifically, when it comes to variability in time, Git exhibits behavior regarding the ancestors of selected `System Revisions` that is not present in how ECCO copes with `Feature Revisions`. However, there is also no contradicting behavior. Consequently, the behavior of Git is applied to `Feature Revisions` and `System Revisions` of the unified definition, i.e., only the selected `System Revisions` and `Feature Revisions` and their ancestors are retained during this operation. Consequently, this unification provides additional semantics for uniformly dealing with variability in space and time beyond the behavior of the analyzed tools.

Operation: Internalize Unified System (*iUS*). This operation combines two `Unified Systems` by essentially merging their contents and creating their union. In practice, this operation is performed *in place*. It updates a `Unified System` US by integrating the contents of another `Unified System` US'.

Comparison: This operation is supported by Git and ECCO (i.e., *pull/push*), and VTS (i.e., *put* given a partial `Configuration`). While Git merges the `System Revisions`, ECCO behaves analogously by merging the `Features` and `Feature Revisions`. In both cases, each respective revision graph is merged by merging the predecessors and successors of each individual `Revision`. Additionally, both tools merge `Fragments` and `Mappings`. VTS supports this operation via its *put* operation given an empty (or *true*) expression as parameter if it follows a *get* operation with a partial `Configuration`. Since VTS does not support variability in time and thus has no `Revisions`, its *put* operation essentially replaces the contents of the `Unified System` US with the contents of the `Unified System` US'.

Unification: The unification of this operation is described in Figure 6.11 and considers an aspect from each of the three tools: `System Revisions` from Git, Features from VTS and ECCO, and Feature Revisions from ECCO; and applies the underlying intention to each dimension, namely to combine two Unified Systems in a single Unified System. Specifically, the Fragments, Features, Revisions (including predecessors, successors, and enables relations), and Mappings are combined into a single Unified System.

6.4. Expected Benefits

So far, the unified operations, their unification process as well as main design decisions have been described. This section comprises a discussion of the expected benefits of the unified operations.

Likewise to the expected benefits of the unified conceptual model (see Section 5.3), the unified operations can be used in a *descriptive manner*. By systematically building on the functionality of contemporary and diverse tools from both the SPLE and SCM research area, they extend the common foundation for unified management of variability in space, broadening the body of knowledge and advancing state of the art. Researchers and practitioners can use the operations for understanding and getting an overview of recent research and practices, and for comparing functionalities of existing tools. Beyond that, compatibility of tools can be analyzed by means of their supported operations and expected inputs and provided outputs.

Moreover, the unified operations can be used in a *prescriptive manner*. A tool that supports the unified operations could provide benefits in several ways: Since it manages both variability dimensions simultaneously, it liberates from the necessity of employing and maintaining particular tools for either version control or product line engineering that need to be compatible while requiring developers to often switch context. Thus, no heterogeneous tool landscape is required, which reduces maintenance costs and development time, since data does not need to be imported and exported between tools, which may also lead to loss of information. Furthermore, by addressing gaps in state of the art (such as dealing with variability in space and time while explicitly managing Feature Revisions and System Revisions simultaneously) and not losing functionality of the analyzed tools, the unified operations propose novel ways for uniformly operating on concepts for variability in space and time. Upon every modification of the Unified System, either of the problem space (i.e., Features and Constraints) or the solution space (i.e., Fragments), the

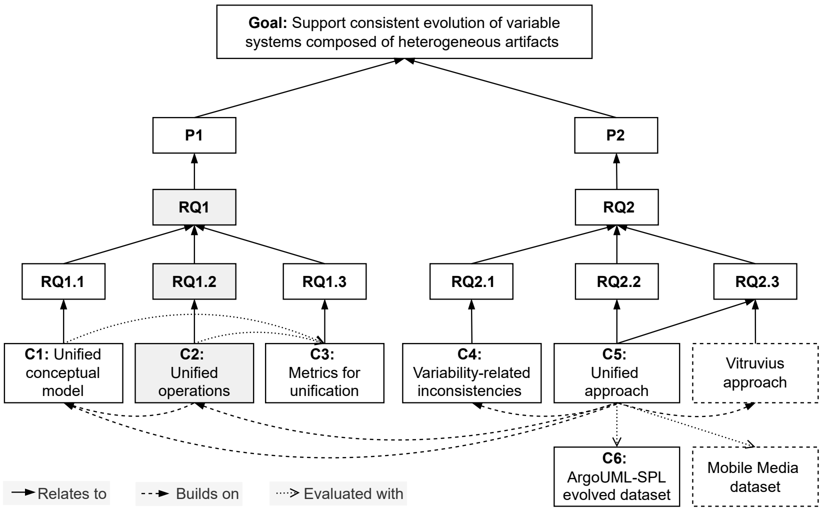

Goal: Support consistent evolution of variable systems composed of heterogeneous artifacts
P1
P2
RQ1
RQ2
RQ1.1
RQ1.2
RQ1.3
RQ2.1
RQ2.2
RQ2.3
C1: Unified conceptual model
C2: Unified operations
C3: Metrics for unification
C4: Variability-related inconsistencies
C5: Unified approach
Vitruvius approach
C6: ArgoUML-SPL evolved dataset
Mobile Media dataset
Relates to
Builds on
Evaluated with

Figure 6.12.: Contribution of Chapter 6 of the thesis.

computation of a new `System Revision`, `Feature Revisions` and `Mappings` happens in a fully automated manner. While direct editing operations require significant manual effort and are thus error-prone and costly, the high degree of automation of view-based editing significantly reduces the manual effort and the cognitive complexity of evolving a variable system. Consequently, unified view-based operations support the conception of novel techniques with a higher degree of automation compared to direct editing operations.

6.5. Summary

This chapter presented the unified operations (C2). The goal was to specify the operational management of variability in space and time while considering `Feature Revisions` and `System Revisions` simultaneously, which is beyond the behavior of the analyzed tools. Inputs and outputs of the unified operations were defined based on the concepts of the unified conceptual model (see Figure 5.3) to lift the operations to the same level of abstraction. To identify relevant tool operations, an expert survey was conducted based on use case questionnaires which were completed by one expert per tool, revealing the individual tool operations. Tool operations were considered that i) deal with variability in space and time, ii) operate on the abstraction level of the unified conceptual model and, iii) modify the `Unified System` or produce non-trivial, mutable output from a `Unified System`. Operations were categorized based on a clear and self-contained concern to avoid redundancies and ambiguities, for instance, one operation for externalizing a `Product` and another operation for externalizing a `Unified System`. Moreover, the identified operations were classified according to the edit modality (direct or view-based editing) and development paradigm (platform or product-oriented development). Finally, the unified operations were specified based on inputs, outputs, and semantics, such that the capabilities of the studied tools are preserved while extending the unified operations to support both variability dimensions. The same process was applied to identify and unify predicates that are used in pre and post-conditions of operations. As result of the identification and unification process, four predicates, 21 direct editing operations, and seven view-based operations are provided. None of the tools support all operations along with all pre and post-conditions for both variability dimensions, which we consider a gap in current tool support for managing variability in space and time simultaneously. Consequently, the unified operations provide means for researchers and practitioners to clarify, communicate and compare their work based on their common and distinguishing operations. Moreover, the unified operations provide guidance for the design of novel unified variability management techniques while liberating from the burden of employing a heterogeneous tool landscape to deal with both variability dimensions.

Thus, this contribution addresses RQ 1.2. Figure 6.12 shows an overview of all contributions and highlights the contribution of this chapter in grey.

7. Variability-Related Inconsistencies

This chapter builds on a publication at SPLC [3].

This chapter presents a classification of variability-related inconsistencies that may occur during the evolution of a variable system including causes and repair options.

The detection and repair of inconsistencies during the evolution of a variable system is an open research problem that has been addressed by numerous works in the SPLE community. Various types of inconsistencies can occur that differ in their causes, effects and possible repair options. Thus, notions of consistency in SPLE are manifold and vary for the problem space and the solution space. Developing an understanding of when, why and in which artifacts inconsistencies occur helps to support the evolution of a variable system in a consistency-aware manner. To organize the research landscape in this field, a literature survey has been performed, and its results have been generalized and mapped to a classification schema. Moreover, gaps in the schema have been filled while ensuring that there is no overlap among the types of inconsistencies (i.e., they are disjoint). The goal was to obtain a generalized, complete, and disjoint classification of variability-related inconsistency types.

Referring to problem statement P2, the following research question is asked:

RQ 2.2 What types of inconsistencies can occur in variable systems?

Section 7.1 presents the literature survey. Section 7.2 introduces a classification of variability-related inconsistency types. In Section 7.3, the identified types are discussed with respect to their completeness and symmetry. Finally, Section 7.4 concludes this chapter with the expected benefits. The results are summarized in Section 7.5.

This chapter thus constitutes the contribution C4.

```
("Software product line engineering" OR "Product line" OR "Variability" OR "
    Variant") AND
("Consistency" OR "Inconsistency" OR "Inconsistent" OR "Well-formedness" OR "
    Well-formed" OR "Repair" OR "Fix") AND ("Evolution" OR "Evolved" OR "Co-
    evolution")
```

Listing 7.1: Search string of the literature survey.

7.1. Literature Survey

I followed the methodology proposed by Kitchenham [113] to systematically search the literature for identifying types of inconsistencies. The literature survey focused on variability-related inconsistencies, i.e., any inconsistencies that can occur during the evolution of a variable system and therefore affect artifacts of the problem space or the solution space. Consequently, the main objective of a publication should be related to providing support for consistent variability evolution.

After careful consideration, the search string presented in Listing 7.1 was used in different scientific databases to retrieve relevant publications.

The following exclusion criteria was specified and applied while inspecting the title and abstract of the publications:

1. The publication is not written in English.
2. The publication originates from outside the software engineering area.
3. The year of the publication is before 2000.
4. The publication is not peer-reviewed.

Figure 7.1 shows the results of the literature search process. Step ① encompassed a digital libraries search. Table 7.1 shows the findings based on IEEE Xplore, the ACM Digital Library and SpringerLink. In total, 406 publications were retrieved from which 25 papers were elicited based on the exclusion criteria. Step ② built on a systematic mapping study by Santos et al. [202] that provides an overview of strategies for consistency checking on SPLs. Based on this mapping study, snowballing was performed that provided 16 relevant publications until 2015, out of which 9 were already retrieved in

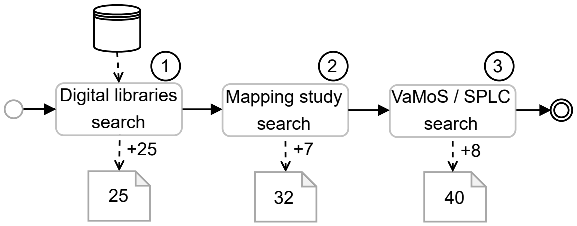


Figure 7.1.: Results of the literature search process.

Table 7.1.: Overview on the literature survey of scientific databases.

Database	Results	Excluded	Offtopic	Remaining
IEEE Xplore	116	92	15	9
ACM Digital Library	24	10 (+ 2 duplicates)	7	5
SpringerLink	266	143 (+ 7 duplicates)	105	11

Step ① and thus represented duplicates. Finally, Step ③ involved an inspection of the last six instances of conference proceedings known to be key venues of publication of the SPLE community: the Systems and Software Product Line Conference (SPLC) and the Variability Modelling of Software-Intensive Systems (VaMoS) Working Conference, revealing 8 further relevant publications. Based on a total of 40 elicited publications between the years 2000 and 2021, a classification of variability-related inconsistency is proposed in the following.

7.2. Types of Inconsistencies

This section describes the identified variability-related inconsistency types that may occur during evolution of a variable system. Moreover, their causes, effects as well as repair options are discussed.

The results of the literature survey were generalized and mapped onto four discrete areas in which inconsistencies can be caused or repaired. Figure 7.2 shows these areas and the identified types of inconsistencies. The region outside the square represents a variable system and the region inside the square represents a `Product`. Separated by a dashed line on the left hand side, a variability model and variable implementation of another variable system indicate distributed development. The upper area outside the square is the *problem space of the variable system*, which comprises a variability model, i.e., `Features` and `Constraints`. The lower area outside the square is the *solution space of the variable system*, which comprises the variable implementation, i.e., `Fragments` and `Mappings`. The upper area inside the square is the *problem space of a product*, which comprises its `Configuration`. The lower area inside the square is the *solution space of a product*, which comprises its non-variable implementation, i.e., only `Fragments`. An inconsistency can be caused or repaired in any of the four areas. Each type of inconsistency is depicted as an arrow from cause (left side) to repair (right side). The Inconsistency Types 1 – 6 are either caused or repaired in a `Product`, Types 7 – 12 are caused or repaired in the variable system, and Types 13 – 14 occur due to distributed development across variable systems.

Table 7.2 shows an overview of the 40 publications collected in the literature survey distinguished between those only considering consistency checking and those additionally including consistency preservation. A publication is arranged according to the addressed Inconsistency Type (rows), and whether it only considers consistency checking (first column) or additionally includes consistency preservation (second column). Note that some publications handle multiple inconsistency types to support round-trip consistency preservation (e.g., Types 2 and 5 or Types 11 and 12).

7.2.1. Product-Level Inconsistencies

Type 1 is caused in the problem space of the system if features are removed or dependencies are introduced between features by adding constraints to the variability model, which decreases the configurable space (in case of feature modeling, this is commonly referred to as *specialization* [236]). This might invalidate previously valid configurations that can be repaired by being transitioned to a valid configuration [25, 78, 174, 175]. All contributions propose semi-automated repair options. Nieke et al. [174, 175] propose an approach for guiding the evolution of configurations. For example, in cases where features are merged, a configuration is computed that maintains the product behavior by automatically transitioning it to a configuration with the same set of artifacts based on mappings. Barreiros et al. [25] focus on the repair of invalid configurations based on partioning and analysis of the feature model while identifying the minimal number of changes to feature (de)selection

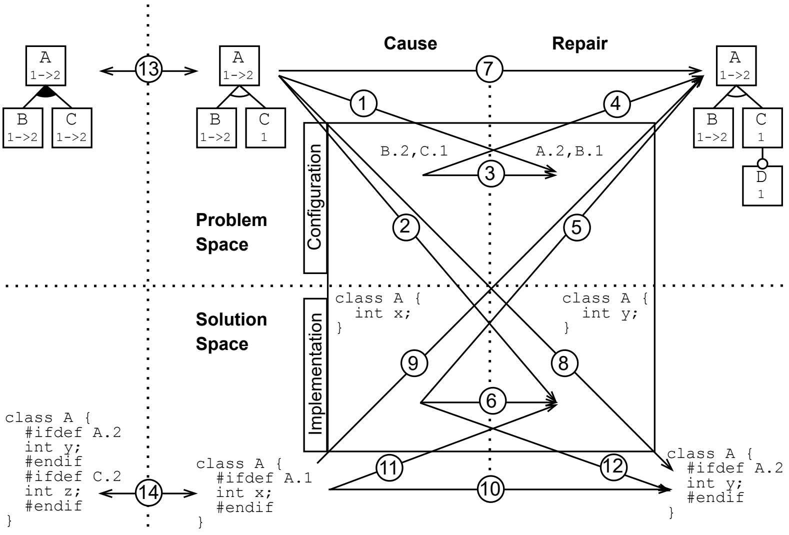


Figure 7.2.: Variability-related inconsistency types. Adapted from [3, Fig. 4].

Table 7.2.: Overview of the 40 elicited publications of variability-related inconsistencies distinguished between those only considering consistency checking and those additionally including consistency preservation.

Type	Checking	Checking & Preservation
1	-	[78, 25, 175, 174]
2	[85, 203, 57, 237, 2]	[149, 217]
3	-	-
4	-	-
5	[204, 85, 57, 147, 237, 103, 234, 133]	[217, 65]
6	[85, 148, 64]	[238, 186, 39]
7	[209]	[29, 91, 192, 120, 18, 176, 17, 100, 94]
8	[54]	[58, 44]
9	-	[58]
10	[132]	-
11	[70]	[93, 213, 211, 81, 208, 214]
12	-	[211, 81]
13	-	[177]
14	-	[87]

from the initial configuration. Gámez and Fuentes [78] additionally consider the automated update of configurations (by adding or removing features) based on changes of cardinality-based feature models.

Likewise, **Type 2** is caused in the problem space of the system if new features are added or dependencies between features are removed by deleting constraints from the variability model, which increases the configurable space (in case of feature modeling, this is commonly referred to as *generalization* [236]). This can validate previously invalid configurations and thus enable new combinations of features. However, deriving a product based on such new configuration might lead to an inconsistent implementation, for instance, because a new combination of features requires additional implementation [85, 203, 57, 237, 2, 149, 217]. A repair would require to modify the implementation of affected products, respectively. This type of inconsistency cannot be fully automated since manual inspection is required in case of new feature combinations that may lead to inconsistencies in the solution space. For instance, if features F_1 and F_2 can be selected in the same configuration after changing the variability model, features F_1 may delete methods required by F_2. Lopez-Herrejon and Egyed [149] propose hints to support the user in fixing such inconsistencies. The authors combine model-driven development with feature-oriented modeling and propose a heuristic based on a feature model analysis to identify the features (and, consequently, the feature modules) where a fix (i.e., required model elements) should be placed. Seidl et al. [217] focus on evolution support due to changes of either the variability model or the non-variable implementation. To this end, the authors propose remapping operations that perform consistency preserving updates of mappings between features and implementation artifacts.

In contrast, **Type 3** and **4** occur in the problem space of a product if an invalid configuration has been selected. For Type 3, a repair requires the transition of the configuration to a valid one. In that case, repair options for Inconsistency Type 1 could be applied. For Type 4, a repair requires the modification of the variability model, such that the desired configuration becomes valid. Similar techniques as for Type 7 could be leveraged to update the variability model in order to support a desired configuration.

Type 5 is caused in the solution space of a product if a change is performed in a product's implementation. This may affect other products, such that their implementation becomes inconsistent [204, 85, 57, 147, 237, 103, 234, 217, 133, 65]. For instance, if a feature F_1 (that is part of several products) is modified in one product p to reference an element introduced by another feature F_2 that is part of p, but not part of all the other products with feature F_1, this would lead to an inconsistent implementation for all products with feature F_1 but without feature F_2. A repair would require to invalidate configurations of the now inconsistent products by lifting the dependencies between features on the solution space to the variability model in the problem space (e.g., $F_1 \Rightarrow F_2$). While most identified publications focus on consistency checks, Seidl et al. [217] propose consistency preserving updates of mappings between features and implementation artifacts due to changes either in the solution space of a product or the variability model (as described above for Type 2). Feichtinger et al. [65] propose an approach that performs a static code analysis of individual products to determine all dependencies and visualizes these as links between features in the feature model to guide the developer in repairing inconsistencies.

Likewise, **Type 6** is caused by changing the solution space of a product. While different artifact types within one product of the variable system describe partially overlapping information that must be kept consistent [116, 62, 56, 242], products also share partially overlapping information in the form of features. Thus, if a feature changes in one product, all other products with the same feature must be changed accordingly [39, 186, 85, 148, 238, 64]. Across all identified publications in the research area of SPLE, consistency between heterogeneous artifacts of a particular product is supported by means of consistency checking and error messages without repair options. For instance, Vierhäuser et al. [238] propose an approach where manually defined consistency constraints between the involved artifacts are incrementally evaluated. In case of violated constraints, error messages are provided to support repairs. To deal with heterogeneous artifacts, the approach requires a specific facade for each artifact type in order to convert the elements contained in an artifact into a generic and artifact-agnostic representation. To ensure consistency between products in case a feature has been changed, Pfofe et al. [186] introduce the rather recent approach *VariantSync* that employs *feature trace recording* [39] to automatically synchronize products.

7.2.2. System-Level Inconsistencies

Type 7 is caused in the problem space of the system if dependencies are introduced between features by adding constraints to the variability model, like for Type 1. This can lead to inconsistencies in the variability model, often referred to as *defects* or *anomalies* [120, 94, 29, 100, 176, 91, 18, 17, 209, 192], such as dead features (i.e., features cannot be selected in any valid configuration of the product line) or redundant constraints (i.e., the removal of a constraint does not change the configurable space). A repair would require to fix the variability model in order to resolve the inconsistencies. Several works go beyond the mere detection of variability model inconsistencies (mostly using SAT) and additionally provide explanations to guide the user in fixing [29, 120, 94, 192], or avoiding them [176, 100]. Guo et al. [91] provide an overview of semantic and syntactic consistency constraints of feature models and propose an ontology-based approach to identify feature model inconsistencies and restore them by means of additionally executed operations (e.g., the removal of a feature invalidates its child features which thus would be removed to maintain the consistency of the feature model). Arcaini et al. [18, 17] propose a rather recent approach to support the automated evolution of feature models upon change requirements, i.e., based on a desired set of features and configurations, an evolutionary algorithm aims at obtaining a feature model that satisfies the specified requirements.

Similarly, **Type 8** is caused in the problem space of the system by adding features or removing constraints. Analogously, a repair would require to modify the solution space [54, 58, 44]. However, in contrast to Type 2, the repair is not performed on a product view, but directly on the variable implementation of the system. Buchmann et al [44] propose an annotative approach for mapping constraints of the feature model to model elements. The approach preserves consistency per construction (e.g., it is not allowed to annotate model elements with undeclared features) or by performing repairs (feature annotations are automatically propagated to dependent model elements).

Type 9 is caused in the solution space of the system if the variable implementation is changed such that dependencies between feature implementations are added that are not reflected in the variability model. In such a case, the implementation and the variability model have become inconsistent. This would require the modification of the variability model (which could be the same repair as for Type 5). To support the co-evolution of the variable implementation with the variability model, Dhungana et al. [58] propose a model-driven approach that identifies changes in the variable implementation using a state-based comparison and suggests actions for repairing the variability model by adding or deleting model elements.

Type 10 is caused in the solution space of the system if a change in the variable implementation leads to an inconsistency in the implementation of products, e.g., syntactically malformed products in case of undisciplined annotations [135] or dangling references in some products. Lauenroth and Pohl [132] propose a formal framework to automatically check the consistency of the requirements specification of the product line and thereby support the detection of inconsistencies prior to the derivation of individual products.

Likewise to Type 10, **Type 11** is caused in the solution space of the system if the variable implementation has been changed. This can lead to a non-viable implementation of products [93, 70, 213, 211, 81, 208, 214]. A repair is performed on the externalized products. Several works propose approaches to preserve consistency for this inconsistency type. Heider et al. [93] research the impact of changes to the variable implementation on derived products, classify types of changes and propose conflict resolutions during updates of products (e.g., by replaying configuration decisions in case only non-variable parts of the variable implementation have been changed). Schulze et al. [211] present an approach that identifies and repairs inconsistencies between an evolved variable implementation and an evolved product based on a *three-way comparison* (i.e., a product that constitutes an evolved working copy, a newly derived product based on the evolved variable implementation, and the product before the evolution serving as common base). The approach provides merging techniques ranging from fully automated to manual inconsistency repair. Gerling [81] proposes to support the co-evolution of the solution space of the system and product by automatically merging *semantic units* (i.e., semantically related lines of code for a particular purpose). For instance, if a feature is added or changed in a product, the artifacts from which the product was derived must be changed accordingly. In the tool SuperMod [213, 208, 214], which is also studied in this thesis, well-formedness consistency rules are defined and evaluated after a product's externalization. Note that when externalizing a product in SuperMod, the workspace is populated upfront with a feature model which is used to configure the product. In case of violated consistency rules (e.g., an object with multiple container objects), default resolution strategies (e.g., the most recent change to add a containment reference value with respect to the object) are applied to the respective product.

Type 12 is caused by changing the implementation of a product, for instance, to perform improvements or fix bugs. Consequently, the product and the variable implementation differ. The desired repair integrates the improvements of the product into the variable implementation [211, 81]. While in case of direct editing this is an entirely manual task, view-based operations (such as the ones provided in this thesis, e.g., *iC*) automate this task to a point where the user only needs to specify an expression manually. Similar as for Type 11, the approaches proposed by Schulze et al. [211] and Gerling [81] for supporting the co-evolution between the evolved variable implementation and an evolved product also support Type 12.

7.2.3. Cross-System Inconsistencies

Type 13 and **Type 14** cover inconsistencies due to distributed development.

In **Type 13**, the problem space of two instances of the unified system (e.g., created via a distributed operation such as *eUS*) evolve independently. For instance, an alternative-group of features is changed to an or-group in another instance of the unified system. When the two instances shall be integrated again (e.g., via the distributed operation *iUS*), these changes must be consolidated. Depending on how far the two problem spaces have diverged from each other, the effort for the consolidation can range from trivial (fully automated) to very complex manual merges. Niu et al. [177] propose resolution strategies for integrating variability models, e.g., in case a feature is mandatory in the variability model of one instance of the unified system but optional in another, it becomes mandatory in the other instance of the unified system.

Likewise, in **Type 14**, the variable implementation of two instances of a unified system evolve independently. For example, the mappings (e.g., in the form of annotations such as if-defs) have been changed in one instance of the unified system and shall be integrated into the instance it has been derived from (e.g., via the distributed operations *eUS* followed by *iUS*). To this end, Greiner et al. [87] propose an approach to automatically propagate annotations from one variable system to another.

7.3. Discussion

The literature survey provides evidence that, for most inconsistency types, numerous works exist that target the detection of such inconsistencies. Some approaches additionally provide explanations on the cause of the inconsistency (particularly of inconsistencies that are caused in the problem space), while only very few works exist that deal with the repair of inconsistencies. Consequently, consistency preservation ranges from providing recommendations for repair to actually performing them fully automatically. To this end, several relevant scientific databases and diverse publication venues were considered to improve the reliability of the findings and the generalizability of the results.

As an inconsistency can be caused or repaired in any of the four areas (i.e., the problem and solution space of the variable system and of the product), there are theoretically $4^2 = 16$ possible types of inconsistencies (excluding the two cross-system inconsistency types). However, not all of those combinations of cause and repair are sensible, covered by the definitions of consistency of this thesis (see Section 2.7), or can be found in the literature. Therefore, this work only considers 12 of the 16 possible combinations of cause and repair as types of variability-related inconsistencies.

The literature survey did not yield research on Type 3 and Type 4 inconsistencies. However, according to the definitions of consistency of this thesis (see Section 2.7) reasonable scenarios could be constructed with corresponding causes and repairs. For example, in case of Type 4, such a scenario would be reactive SPLE [12], where the variable system shall be extended with a new (not yet valid) configuration by accordingly adapting the variability model. Therefore, these two types of inconsistencies were deemed relevant and included to obtain a complete picture and ensure that the identified variability-related inconsistency types are general enough to also remain relevant in the future.

Two of the combinations that were not considered as types of inconsistencies are caused in the solution space of the system (i.e., the variable implementation) and repaired in the problem space of a product (i.e., its configuration), and vice versa. Both cases are covered by a sequence of other types of inconsistencies with an intermediate step via the problem space of the system (i.e., the variability model). In the former case, if the implementation of a system is modified such that the configuration of a product must be changed, this is covered by Type 9 (which is caused in the implementation of a system) followed by Type 1 (which is repaired in the configuration of a product). In the latter case, if the configuration of a product is modified such that the implementation of the system must be changed, this is covered by Type 4 (which is caused in the configuration of a product) followed by Type 8 (which is repaired in the implementation of the system).

The remaining two combinations that were not considered as types of inconsistencies are caused and repaired in the solution space of the product (i.e., its implementation) and the problem space of the product (i.e., its configuration). The literature survey did not reveal work where a modification of the implementation of a product requires to repair the configuration of the product and it is questionable whether sensible scenarios can be found. Conversely, a modification of the configuration of a product that requires a repair in the implementation of the product was also not evident by the literature survey. In this case, it can even be questioned, whether the transition of a product to another configuration is sensible, as the identity of a product is given by its configuration. Therefore, in a case where another configuration is desired, instead of changing the configuration of an existing product, simply a new product can be derived from the system.

There is a symmetry between causes and repairs over all inconsistency types, as for every type of consistency there is another type with the same, but reversed, areas of cause and repair. For example, Type 2 is symmetric to Type 5. Such symmetry does not exist between problem space and solution space. For example, Type 2 is caused in the problem space of the system (i.e., variability model) and repaired in the solution space of the product, but there is no type of inconsistency that is caused in the solution space of the system and repaired in the problem space of the product. Similarly, there is no symmetry between the system and the product.

7.4. Expected Benefits

In total, 14 types of variability-related inconsistencies were identified and classified according to their cause and repair in either the problem space or the solution space. This section comprises a discussion of the expected benefits.

The performed synthesis of the existing body of knowledge on variability-related inconsistency types maps and organizes the corresponding research landscape. It enables researchers to classify, communicate, and scope their work. Furthermore, the literature survey revealed inconsistency types that are less considered and thus might be candidates for further research. Moreover, beyond the results of the literature survey, further inconsistency types were identified. Due to the generality and disjointedness of the inconsistency types, future work on variability-related inconsistencies is expected to be classifiable according to the proposed types. The product line community is invited to classify their research according to the proposed classification scheme.

7.5. Summary

This chapter presented the identified variability-related inconsistencies (C4). Based on a literature survey [113], 40 relevant publications were collected, generalized, mapped to a schema, and gaps in the schema have been filled while ensuring that there is no overlap among the types of inconsistencies (i.e., they are disjoint). The goal was to obtain a generalized, complete, and disjoint classification of variability-related inconsistency types. Ultimately, 14 different inconsistency types were identified and classified according to their cause and repair in either the problem space or the solution space of either the variable system or a product. This chapter comprised a description of each inconsistency type, introduced relevant works in that field and explained repair options. Interestingly, it became obvious that while some inconsistency types are often subject to research in SPLE (i.e., Type 1, 2, 5, 7, and 11), the

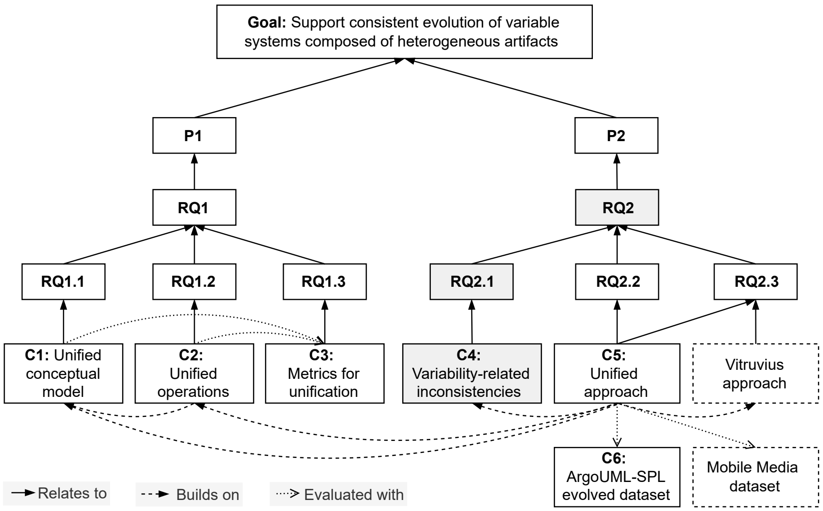

Goal: Support consistent evolution of variable systems composed of heterogeneous artifacts
P1
P2
RQ1
RQ2
RQ1.1
RQ1.2
RQ1.3
RQ2.1
RQ2.2
RQ2.3
C1: Unified conceptual model
C2: Unified operations
C3: Metrics for unification
C4: Variability-related inconsistencies
C5: Unified approach
Vitruvius approach
C6: ArgoUML-SPL evolved dataset
Mobile Media dataset
Relates to
Builds on
Evaluated with

Figure 7.3.: Contribution of Chapter 7 of the thesis.

remaining inconsistency types are researched less. While several approaches propose explanations of inconsistencies or fully-automated repair options, this particularly addresses the problem space. Consistency preservation involving the solution space is addressed significantly less, especially when it comes to heterogeneous artifacts and distributed development.

Thus, this contribution addresses RQ 2.1. Figure 7.3 shows an overview of all contributions and highlights the contribution of this chapter in grey.

8. Unified Approach

This chapter builds on a publication at VaMoS [10] and SPLC [3].

This chapter presents the unified view-based approach to support consistent management of variable systems. Since the unified approach builds upon all prior contributions (C1–C4), it benefits from the insights of unifying concepts, their relations, and operations to deal with variability in space and time simultaneously as well as of variability-related inconsistencies, their causes and possible repairs. The goal is to provide an approach that deals with variability in space and time involving both `Feature Revisions` *and* `System Revisions` and that is also capable of handling variability-related inconsistencies during the evolution of a system.

Referring to problem statement P2, two research questions are asked:

RQ 2.2 How can unified operations be combined with consistency preservation of variability-related inconsistency types?

RQ 2.3 How can the VITRUVIUS approach be leveraged to support variability in space and time and preserve consistency in variable systems comprised of heterogeneous artifacts?

Section 8.1 introduces the construction process of the unified approach. Section 8.2 gives an overview of the conceptual architecture of the unified approach. The proposed workflow of the unified approach when evolving a variable system is presented in Section 8.3. Section 8.4 comprises an augmentation of unified operations with consistency preservation, while Section 8.5 encompasses a demonstrating application. Finally, Section 8.6 provides details on how the unified approach deals with a selected subset of variability-related inconsistencies caused or repaired in the solution space. Section 8.7 presents the prototypical implementation of the unified approach before expected benefits in Section 8.8 and a summary of the results in Section 8.9 conclude this chapter.

This chapter thus constitutes the contribution C5.

8.1. Construction Process

Figure 8.1 shows the process for constructing the unified approach.
In Step ①, I identified types of inconsistencies that occur during the evolution of a variable system and possible repair options (see Chapter 7). In Step ②, I refined the unified elements (i.e., the unified conceptual model and unified operations) into a concrete metamodel and corresponding operations of the unified approach. Finally, in Step ③, I augmented the unified operations with capabilities of consistency preservation, ranging from automated suggestions to restore consistency to fully-automated repair. In the following, necessary refinements to the unified conceptual model and operations are explained.

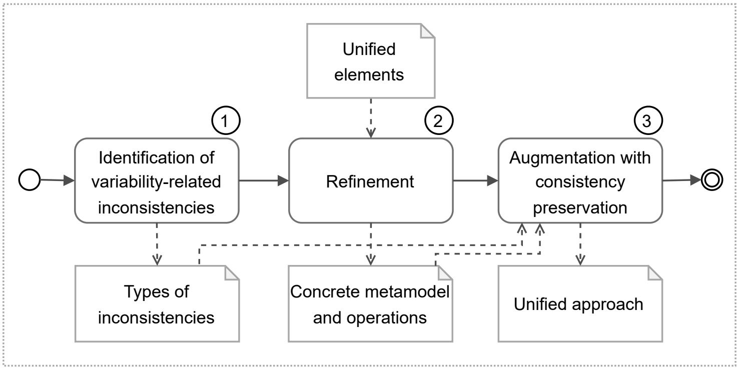

Unified elements
1
2
3
Identification of variability-related inconsistencies
Refinement
Augmentation with consistency preservation
Types of inconsistencies
Concrete metamodel and operations
Unified approach

Figure 8.1.: Unified approach construction process.

8.2. Conceptual Architecture

This section presents the conceptual architecture of the unified approach. Main design ideas of the approach are explained and key mechanisms described. The section starts with refinements made to the unified conceptual model, and continues with the integration with the Vitruvius approach and refinements of the employed unified view-based operations.

```
context CrossTreeConstraint
inv:
  Set{self.expression} -> closure(e:Expression |
    if e.oclIsKindOf(Operator) then e.expr else Set{})
  -> select(e:Expression | e.oclIsKindOf(Variable))
  -> forAll(v:Variable | v.option.oclIsKindOf(FeatureOption))
```

Listing 8.1: Well-formedness of a `Cross-tree Constraint`.

8.2.1. Concrete Metamodel

Figure 8.2 shows the concrete metamodel of the unified approach as a refinement of the unified conceptual model. Thereby, metaclasses are created for each concept and new metaclasses are introduced where necessary. Equivalent to the conceptual model, metaclasses for variability in space are colored green, for variability in time orange, for variability in both dimensions purple, and metaclasses for unified concepts (that cope with either one variability dimension or both) are white. Relations are colored analogously. Lighter colors and italic fonts represent abstract metaclasses. The concrete metamodel employs the metaclasses `Unified System`, `Feature`, `Feature Revision`, `System Revision`, `Mapping`, and `Configuration`, and refines `Constraint` and `Fragment`. A `Feature Model` is used as variability model, which is the de-facto standard of variability modeling in research and industry [50, 107, 117, 194]. Both `Tree Constraint` and `Cross-tree Constraint` specialize `Constraint`. While `Tree Constraints` only refer to `Features` (that can be decomposed into optional `Features`, mandatory `Features`, or-groups, and alternative-groups), `Cross-tree Constraints` can be formulated on `Features` and `Feature Revisions` via a Boolean `Expression`. The `Unified System` contains `Delta Modules` that specialize `Fragments` and represent the variability mechanism for composing `Products` based on a `Configuration`. A `Delta Module` comprises `Deltas` that for now are omitted from the figure. A `Mapping` relates `Delta Modules` and `Options` via a Boolean `Expression`. The expression is represented by an expression tree where inner nodes represent operators, such as *Implication* or *Conjunction*, and leafs represent `Options`. An additional OCL constraint in Listing 8.1 ensures that expressions contained in `Cross-tree Constraints` only refer to `Feature Options` instead of any `Option`, as is the case for expressions contained in `Mappings`.

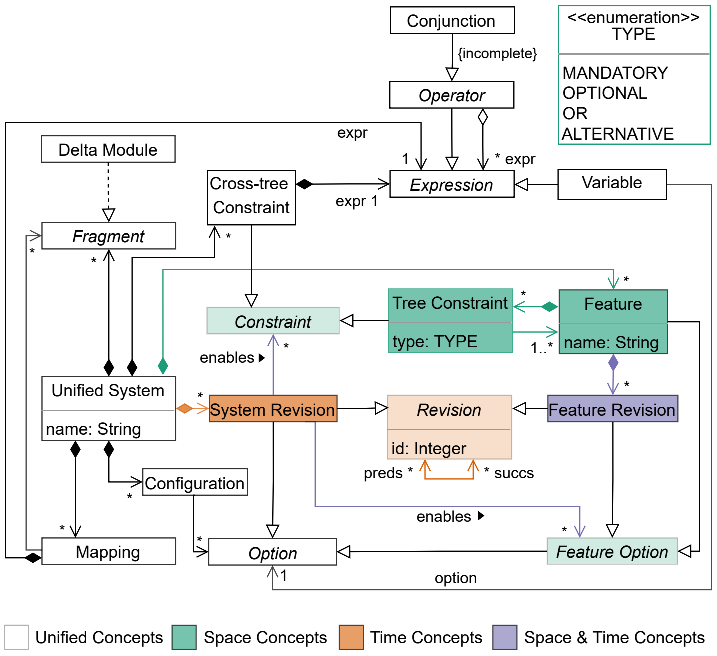


Figure 8.2.: Concrete metamodel of the unified approach. Adapted from [3, Fig. 3].

8.2.2. Integration with Vitruvius

To leverage Vitruvius' consistency preservation mechanisms between different artifact types (see Section 2.6.2), its integration with the unified approach is explained in the following. Figure 8.3 shows a conceptual model that describes the connection between the unified approach and Vitruvius, highlighting concepts that belong to the solution space in orange and concepts that belong to the problem space in blue. Note that only the connecting concepts are depicted.

Most depicted concepts are part of the solution space. The `Delta Module` of the unified approach refines the `Fragment` and comprises `Changes` (i.e., `Deltas`) from Vitruvius. A `Change` can be atomic, such as an additive or subtractive `Change`, or be a compound `Change`. Vitruvius uses a dedicated metamodel to describe possible change types that are omitted for the sake of simplicity. An

Artifact Model is derived by applying Changes and represents an engineering artifact of a particular type, e.g., an artifact model for Java source code or an artifact model for a UML diagram. An Artifact Model conforms to an Artifact Metamodel, e.g., Java metamodel or UML metamodel. A V-SUM is comprised of Artifact Models and conforms to the V-SUM Metamodel that, in turn, contains the Artifact Metamodels. Moreover, the V-SUM Metamodel comprises Consistency Preservation Rules (CPRs) that are specified between two Artifact Metamodels. The V-SUM Product specializes the V-SUM of Vitruvius and the Product of the unified approach. Specifically, the V-SUM Product temporarily stores Changes (i.e., *original changes* performed by the developer during the modification of an Artifact Model) until they are integrated into the Unified System. Moreover, the V-SUM Product contains its *valid* and *complete* Configuration (see Figure 6.3), that connects concepts of the problem space (i.e., Options) with concepts of the solution space (i.e., Product). Based on the System Revision and Feature Revisions in this Configuration, the newly created System Revision and Feature Revisions can be related correctly in the revision graphs of the Unified System upon the integration of Changes. The Unified System contains Configuration and Mappings. All three concepts connect the problem space and the solution space (i.e., the Unified System contains concepts that are part of both spaces, while the Mapping relates Options with Fragments). Note that *consequential changes* are not stored since they may vary based on the context, i.e., the Configuration of the Product, in which the causing original changes are applied.

In sum, the unified approach leverages the consistency preserving mechanisms of Vitruvius to preserve consistency between artifact models in the solution space. Specifically, by employing the Deltas used in Vitruvius and by specializing its V-SUM as Product. In turn, the unified approach extends Vitruvius with concepts of the problem space to enable unified variability management.

8.2.3. Concrete Operations

The inputs and outputs of the view-based unified operations are refined accordingly to the concrete metamodel. The *eD* operation produces a feature model as output and the *iD* operation takes a feature model as input instead of sets of Features and Constraints. The *eP* operation utilizes Deltas as transformational variability mechanism [206]. The *iC* operation internalizes

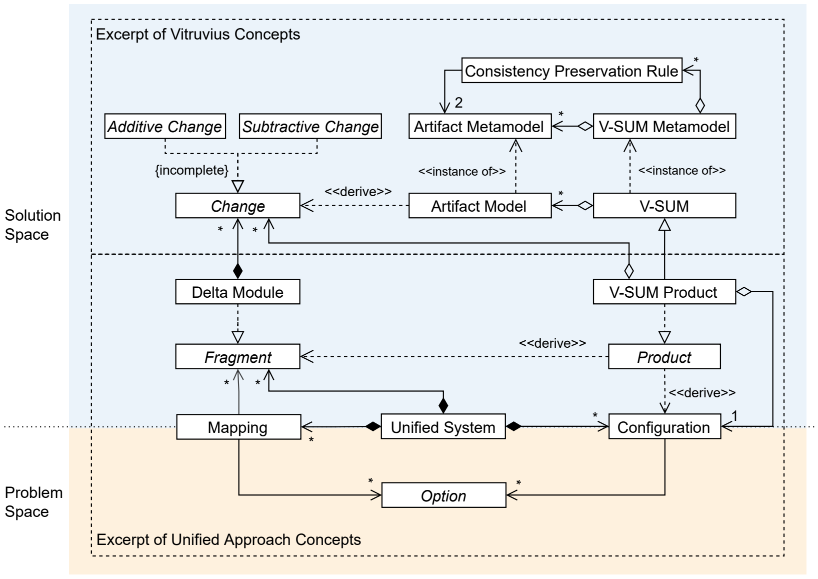


Figure 8.3.: A conceptual model of the relevant concepts for describing the connection between the unified approach and Vitruvius.

the manually performed changes on a product view by propagating recorded `Deltas` to the `Unified System`. In the following, the algorithms for each operation are presented using the notation introduced in Section 5.2.3.

The *eD* operation is shown in Algorithm 8.1. It takes as input a set of `System Revisions` SR. The set of `Feature Options` FO', `Tree Constraints` TC' and `Cross-tree Constraints` CTC', that are enabled by any of the given `System Revisions` $sr \in SR$, are obtained and used to construct the feature model FM. Finally, the feature model FM is returned.

The *iD* operation is shown in Algorithm 8.2. It takes as input a feature model FM. First, a new `System Revision` sr' is created. All `System Revisions` from which the feature model FM was created via *eD* are set as predecessors of the new `System Revision` sr' and receive it as successor. Next, the new `System Revision` sr' is added to the `Unified System`. It enables the `Feature Options`, `Tree Constraints` and `Cross-tree Constraints` in the feature model FM. Existing `Mappings` M', that contain any of the `System Revisions` in SR_{FM}, are obtained and copied to M'', and all occurrences of any `System Revision` in their expression e_m of a `Mapping` $m \in M''$ are replaced by the new `System Revision` sr'. The copied and updated `Mappings` are added to the `Unified System`.

The *eP* operation is shown in Algorithm 8.3. It takes as input a `Configuration` c. First, it is checked whether c is a valid and complete `Configuration`. Then, all `Mappings` $M' \subseteq M$ in the `Unified System` whose expression is satisfied by the given `Configuration` c are obtained in the same order in which they were added to the `Unified System` via *iC* (shown in Algorithm 8.4). These `Mappings` M' are used to construct the `Product` p. Specifically, each `Fragment` $ft \in FT_m$ to which a `Mapping` $m \in M'$ refers represents a `Change`. These `Changes` are applied in sequence to the initially empty model I. Note that no additional re-ordering of `Changes` is needed, since the deltas are recorded on product views and not created manually. This guarantees that required deltas are already contained in the `Unified System` as they are needed for the externalization of the `Product` in the first place. Finally, the `Product` p is constructed and returned.

The *iC* operation is shown in Algorithm 8.4. It takes as input a `Product` p and a conjunction of `Feature Options` e. First, a new `System Revision` sr' is created. All `System Revisions` of the `Configuration` c_p of the given `Product` p are set as direct predecessors. The new `System Revision` sr' is set as successor for all its predecessors and added to the `Unified System` us. Next, the *enables* relations of all predecessor `System Revisions` are copied to the new `System Revision`. For every `Feature` f in the expression e, a new `Feature Revision` fr'_f is created and added to that `Feature`'s set of `Feature Revisions` $f.FR$. Each new `Feature Revision` is set as direct successor for all `Feature Revisions` of the same `Feature` f in the `Configuration` c_p of the

Algorithm 8.1 Externalize Domain (eD) operation

```
1: function eD($SR$)
2:     $FO' \leftarrow \bigcup_{sr \in SR} FO_{sr}$                ▷ Obtain enabled features
3:     $TC' \leftarrow \bigcup_{sr \in SR} TC_{sr}$                ▷ Obtain enabled tree constraints
4:     $CTC' \leftarrow \bigcup_{sr \in SR} CTC_{sr}$              ▷ Obtain enabled cross-tree constraints
5:     $FM \leftarrow (SR, FO', TC', CTC')$             ▷ Construct feature model
6:     return $FM$
7: end function
```

Algorithm 8.2 Internalize Domain (iD) operation

```
function iD($FM$)
    $sr' \leftarrow ()$    ▷ Create new system revision
    $sr'$.succ $\leftarrow \{\}$
    $sr'$.pred $\leftarrow \{sr \mid sr \in SR \wedge sr \in SR_{FM}\}$    ▷ Set predecessors of new system revision
    for each $sr \in sr'$.pred do
        $sr$.succ $\leftarrow sr$.succ $\cup \{sr'\}$    ▷ Add new system revision as successor
    end for
    $SR \leftarrow SR \cup \{sr'\}$    ▷ Add new system revision to unified system
    $FO_{sr'} \leftarrow FO_{FM}$    ▷ New system revision enables feature options of FM
    $TC_{sr'} \leftarrow TC_{FM}$    ▷ New system revision enables tree constraints of FM
    $CTC_{sr'} \leftarrow CTC_{FM}$    ▷ New system revision enables cross-tree constraints of FM
    $M' \leftarrow \{m \mid m \in M \wedge sr'.\text{pred} \cap e_m \neq \emptyset\}$    ▷ Obtain mappings that contain current system revision(s)
    $M'' \leftarrow$ <Copy each mapping $m \in M'$ and replace all occurrences of any $sr \in SR_{FM}$ in the expression of any $m \in M''$ by $sr'$>
    $M \leftarrow M \cup M''$    ▷ Add copied and updated mappings to unified system
end function
```

Algorithm 8.3 Externalize Product (eP) operation

```
function eP($c$)
    if $\neg$valid($c$) $\vee$ $\neg$complete($c$) then    ▷ Check if configuration is valid and complete
        return ERROR
    end if
    $M' \leftarrow \{m \mid m \in M \wedge SAT(c \wedge e_m\}$    ▷ Obtain mappings whose expression is satisifed by the configuration
    $I \leftarrow ()$    ▷ Create empty model
    for each $m \in M'$ do
        for each $ft \in FT_m$ do
            $I \leftarrow$ apply$(I, ft)$    ▷ Apply delta module to model
        end for
    end for
    $p \leftarrow (c, I)$    ▷ Construct product
    return $p$
end function
```

given Product p. The new Fragments FT^+ (which constitute the recorded Changes that were applied to Product p) are obtained. Existing Mappings M', that contain any of the System Revisions in c_p, are obtained and copied to M'', and all occurrences of any System Revision in their expression e_m of a Mapping $m \in M''$ are replaced by the new System Revision sr'. The copied and updated Mappings are added to the Unified System. Finally, a new Mapping is created for the new Fragments FT^+ with the expression $sr' \wedge e$ and also added to the Unified System.

8.3. Proposed Workflow

Figure 8.4 depicts a UML activity diagram of the general workflow with involved roles of the unified approach. Traditional proactive SPLE distinguishes between domain engineering and application engineering [191]. Respectively, the domain engineer defines the variability at an appropriate level of abstraction (for developers as well as customers) by defining the variability model and establishing the reusable platform. The application engineer builds customer-specific applications by deriving and completing Products based on the platform established in domain engineering. While the unified approach proposed in this thesis relies on the main principles of SPLE, such as the reusable platform and feature-oriented development, it represents a VarCS (see Definition 2.2) and thus encourages product line development based on product views to overcome limitations of existing variability management practices (see Section 2.2.2). Consequently, the roles of the unified approach deviate from the traditional roles in SPLE.

Analogously to the domain engineer in SPLE, the problem space engineer defines the variability and commonality of the product line by means of a variability model (i.e., a feature model). Based on a particular System Revision, the operation *eD* (see Algorithm 8.1) provides a feature model to the problem space engineer. Due to changed requirements, the problem space engineer evolves the feature model, for instance, by adding new Features. To integrate the performed changes back into the Unified System, the evolved feature model is input to the operation *iD* (see Algorithm 8.2) and leads to an updated instance of the Unified System. Analogously to the application engineer in SPLE, the solution space engineer externalizes a Product based on a Configuration via *eP* (see Algorithm 8.3) and develops product-specific

Algorithm 8.4 Internalize Changes (iC) operation

```
 1: function iC($p$, $e$)
 2:     $sr' \leftarrow ()$                                    ▷ Create new system revision
 3:     $sr'.\text{succ} \leftarrow \{\}$
 4:     $sr'.\text{pred} \leftarrow \{sr \mid sr \in us.SR \land sr \in c_p\}$          ▷ Set predecessors of new system revision
 5:     for each $sr \in sr'.\text{pred}$ do
 6:         $sr.\text{succ} \leftarrow sr.\text{succ} \cup \{sr'\}$      ▷ Set new system revision as successor
 7:     end for
 8:     $us.SR \leftarrow us.SR \cup \{sr'\}$   ▷ Add new system revision to unified system
 9:     $sr'.\text{enablesF} \leftarrow \bigcup_{sr \in sr'.\text{pred}} sr.\text{enablesF}$
10:     $sr'.\text{enablesTC} \leftarrow \bigcup_{sr \in sr'.\text{pred}} sr.\text{enablesTC}$
11:     $sr'.\text{enablesCTC} \leftarrow \bigcup_{sr \in sr'.\text{pred}} sr.\text{enablesCTC}$
12:     for each $f \in e$ do
13:         $fr'_f \leftarrow ()$                                ▷ Create new feature revision
14:         $fr'_f.f \leftarrow f$                               ▷ Set feature of feature revision
15:         $f.FR \leftarrow f.frFR \cup \{fr'_f\}$              ▷ Add feature revision to feature
16:         $fr'_f.\text{succ} \leftarrow \{\}$
17:         $fr'_f.\text{pred} \leftarrow \{fr_f \mid fr \in FR_f \land c_p\}$ ▷ Set predecessor of new feature revision
18:         for each $fr_f \in fr'_f.\text{pred}$ do
19:             $fr_f.\text{succ} \leftarrow fr_f.\text{succ} \cup \{fr'_f\}$        ▷ Set new feature revision as successor
20:         end for
21:     end for
22:     $FT^+ \leftarrow p.\text{recordedChanges}$                  ▷ Get recorded changes
23:     $M' \leftarrow \{m \mid m \in M \land sr'.\text{pred} \cap e_m \neq \emptyset\}$        ▷ Obtain mappings that contain current system revision(s)
24:     $M'' \leftarrow$ <Copy each mapping $m \in M'$ and replace all occurrences of any $sr \in c_p$ in the expression of any $m \in M''$ by $sr'$>
25:     $us.M \leftarrow us.M \cup M''$   ▷ Add copied and updated mappings to unified system
26:     $m' \leftarrow (sr' \land e, FT^+)$ ▷ Create mapping for new fragments (i.e., deltas)
27:     $us.M \leftarrow us.M \cup \{m'\}$                  ▷ Add mapping to unified system
28: end function
```

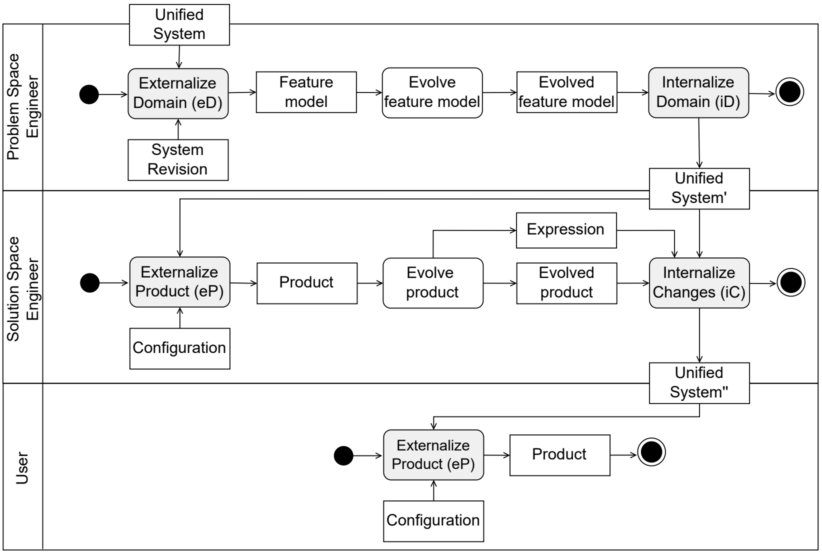


Figure 8.4.: Proposed workflow of the unified approach with roles.

artifacts. Moreover, the solution space engineer is also responsible for developing reusable artifacts of the platform. Respective `Changes` to the `Product` are recorded by a change monitor, and integrated into the `Unified System` in a fine-grained feature-oriented manner via *iC* based on an expression (see Algorithm 8.4), which further updates the `Unified System`. Finally, a user can externalize a `Product` based on a `Configuration` via *eP*.

To sum up, instead of domain and application engineering, the unified approach classifies activities into problem space and solution space engineering. By integrating changes performed on a product in a fine-grained manner into the reusable platform, other products can benefit from these changes. Additionally, documenting changes in a feature-aware manner allows for intensional versioning (i.e., the externalization of `Products` that have never been internalized before). While the described workflow supports traditional proactive development of an SPL, it particularly encourages reactive development.

8.4. Augmentation of Operations with Consistency Preservation

Table 8.1 maps the product-level types of inconsistencies to unified view-based operations. For every inconsistency type, the causing operation, the affected artifact, respective repair operations (denoted as regular expressions) and the artifact in which the repair is performed are identified. While the unified approach offers consistency preservation for inconsistency types that are either caused or repaired in the solution space (i.e., Type 2, 5, and 6), the augmentation of unified view-based operations with consistency preservation for *every* product-level type of inconsistency is described in the following to answer RQ 2.2.

Producing a view on the feature model via *eD* and integrating modifications, such as the addition of a constraint, via *iD*, may lead to Type 1 or 2 inconsistencies. Removing features or adding constraints between features (Type 1) reduces the configurable space, which might invalidate configurations currently in use. After *iD*, a repair automatically suggests an alternative configuration according to one of several possible strategies. Such strategies could be based on already existing ideas in literature (see Section 7.2.1). In addition, update strategies can be conceived by analyzing and quantifying differences in features as well as in implementation for the selection of adjacent valid configurations: i) the highest number of common features, i.e., the lowest number of removed features; ii) the lowest number of feature differences, i.e., the lowest number of added and removed features; iii) the highest overlap of implementation, i.e., the lowest number of deletions in the implementation; iv) the smallest difference in implementation, i.e., the lowest number of insertions and deletions in the implementation. Furthermore, v) already performed changes on a product implementation and yet another update strategy that minimizes the affect of the transition on the already changed parts of the implementation could be considered. Optionally, the developer may intervene and manually adapt the computed configuration, or simply confirm it as input to *eP* to transition to the new product.

In contrast, adding features or removing constraints between them (Type 2) increases the configurable space and enables new configurations that may not (yet) lead to consistent product implementations due to missing implementation of new feature(s) or feature interaction(s). Thus, after *iD*, an enumeration of new features and feature combinations is provided. Henceforth, in case the developer externalizes an affected configuration via *eP*, a hint is issued that the product implementation produced by *eP* may be missing implementation for new features or feature combinations and thus be inconsistent. While the approach provides suggestions for the developer, the actual fix must be implemented manually and internalized via (*iC*)+.

Producing a product via *eP* based on an invalid configuration leads to Type 3 and 4 inconsistencies. A repair requires either to externalize another product via *eP* with a valid configuration (semi-automatically by selecting a transition strategy, as mentioned for Type 1) or by adapting the feature model (via approaches such as *minimal unsatisfiable subsets* [136], which could be used to suggest possible repair options in the feature model). In the latter case, a view on the feature model corresponding to the causing configuration can be generated automatically via *eD*, and possible repairs to the feature model can be applied to the externalized view automatically. Again, the developer may intervene and make further modifications, or simply confirm and internalize the repaired feature model via *iD*.

Producing the product via *eP* and integrating changes via *iC* may lead to Type 5 and 6 inconsistencies. In both cases, the implementation may have become inconsistent. If a feature dependency has been added on implementation level that is not captured by the feature model (Type 5), products with an inconsistent implementation may be externalized. A repair requires to add the respective implementation level dependency between features in the feature model. The view on the domain can be created via *eD* and the corresponding feature dependency can be added as a cross-tree constraint to the feature model automatically to invalidate configurations of inconsistent products. Again, the developer can make further modifications to the feature model view, e.g., perform a more impactful restructuring of the feature model, and then confirm the repair by performing *iD*.

Changing redundant or dependent information across heterogeneous artifacts of the same product, e.g., by modifying the Java view of a product produced via *eP* and applying it to the system via *iC*, can quickly lead to inconsistencies in other artifact types and products due to redundancies or dependencies (Type 6). Based on consistency preserving mechanisms provided by Vitruvius (see Section 2.6.2), this type of inconsistency can be detected and repaired (semi-) automatically by applying *consequential changes* as reaction to the manually performed *original changes* of the developer. The consequential changes can perform repairs in the same type of artifacts and other types of artifacts, or other products entirely. In the latter case, these consequential changes may differ based on the product in which they are applied. If the abstraction level of the corresponding artifacts differs, several valid repair options can be possible. Thus, the automatically applied consequential changes may require user interaction, e.g., an added Java class could represent a UML component or not be propagated at all in case it is only supposed to represent a class on implementation level.

In conclusion, inconsistencies can be caused by modifications to different artifacts of a variable system and require different kinds of repairs that may vary in their potential for automation. While in some cases, e.g., Types 3 and 4, several semi-automated repair options exist, there are cases that require manual inspection, such as Type 2.

8.5. Demonstrative Application

This section illustrates the above described unified consistency preservation based on the running example (Section 2.1). We start at system revision one where feature `Dist` does not exist yet and all other features have exactly one feature revision.

Assume that we externalize the domain (i.e., feature model) in the first system revision via *eD*. We add feature `Dist` to the feature model and internalize it via *iD*, leading to the second system revision. The newly supported products are initially inconsistent as they are missing the implementation for the new feature (Type 2). Therefore, we externalize a product via *eP* with the configuration $\{\texttt{Car}_1, \texttt{ET}, \texttt{Gas}_1, \texttt{Dist}\}$. We add Lines 7 and 11 to it and internalize the changes via *iC* with the expression *Dist*, leading to the first feature revision for `Dist` and the third system revision. Additionally, we add the interaction between features `Gas` and `Dist` by adding Line 8. We internalize it with expression *Gas && Dist*, leading to the second revision of `Gas` and `Dist`, and the fourth system revision. In the process, other artifact types and products may have become inconsistent (Type 6). To restore consistency in the SysML model, changes performed in Java are automatically propagated to the SysML model in Line 5. To restore consistency in other affected products that involve the added feature `Dist`, such as $\{\texttt{Car}_1, \texttt{ET}, \texttt{Ele}_1, \texttt{Dist}_2\}$, we externalize it via *eP*. This product is inconsistent as it lacks a return statement in method *getDistanceLeft().* Thus, Line 9 is added and internalized with the expression

Table 8.1.: Mapping between inconsistency types and unified operations.

Type	Causing operation(s)	Causing artifact	Repair operation(s)	Repaired artifact
1	*eD*, *iD* (configurable space reduction)	Variability model	*eP*	Configuration
2	*eD*, *iD* (configurable space increase)	Variability model	(*eP*, (*iC*)+)+	Implementation
3	*eP* (invalid configuration)	Configuration	*eP*	Configuration
4	*eP* (invalid configuration)	Configuration	*eD*, *iD*	Variability model
5	*eP*, *iC* (feature dependency addition)	Implementation	*eD*, *iD*	Variability model
6	*eP*, *iC* (redundant information change)	Implementation	*eP*	Implementation

Ele && Dist, leading to the second revision for `Ele`, the third revision for `Dist` and the fifth system revision. Since electric cars need to constantly monitor the remaining distance, Line 5 is added and internalized with the expression *Ele*. While this leads to the third revision for `Ele` and the sixth system revision, it causes a dependency between $\texttt{Ele}_3$ and $\texttt{Dist}_3$ that is not covered by the feature model (Type 5), and thus allows for externalizing inconsistent products (e.g., $\{\texttt{Car}_1, \texttt{ET}, \texttt{Ele}_3)$. Specifically, the mapping of Line 5 ($\texttt{Ele}_3$) in conjunction with the feature model does not imply the mapping the required Line 7 ($\texttt{Dist}_3$). Therefore, the feature model is externalized via *eD*, the cross-tree constraint $\texttt{Ele}_3 \Rightarrow \texttt{Dist3}_3$ is added as a constraint to the feature model, and the repaired feature model is internalized again via *iD*, which this leads to the seventh system revision. As a consequence, the product $\{\texttt{Car}_1, \texttt{ET}, \texttt{Ele}_3\}$ is not supported anymore (Type 1). Transitioning the configuration to $\{\texttt{Car}_1, \texttt{ET}, \texttt{Ele}_3, \texttt{Dist}_3\}$—with the highest number of common features and lowest number of feature differences—validates the configuration. Finally, assume that we strive for a hybrid car and create a configuration $\{\texttt{Car}_1, \texttt{ET}, \texttt{Ele}_3, \texttt{Gas}_2, \texttt{Dist}_3\}$, which, however, is not supported by the feature model. Since we do not want to change our configuration (Type 3), we externalize the domain (in the latest system revision) and change the *alternative* group of the features `Gas` and `Ele` to an *or* group (Type 4), leading to the eighth system revision. Now, we just need to add the missing interactions between $\texttt{Gas}_2$, $\texttt{Ele}_3$ and $\texttt{Dist}_3$ in the hybrid car product (Type 2). Thus, we remove Lines 8 and 9 from the product, add Line 10 and internalize the changes with *Gas && Ele*, leading to the third revision for `Gas`, the fourth revision for `Ele`, and the ninth system revision. Note that the mappings of the deleted lines are appended with the negation of the provided features.

8.6. Preservation of Solution Space Inconsistencies

While the unified operations have been augmented with consistency preservation capabilities for all product-level types of inconsistencies, the unified approach integrates consistency preservation for a selected subset of these. Specifically, it addresses inconsistency types that are either caused or repaired in the solution space of a product, i.e., Type 2 (feature model to product consistency), Type 5 (product to feature model consistency), and Type 6 (product consistency). In the following, the workflow of the unified approach for dealing with each of these inconsistency types is depicted.

8.6.1. Feature Model to Product Consistency

Figure 8.5 shows a UML sequence diagram, and Figure 8.6 an illustrative depiction, of the workflow of the proposed unified approach for Type 2 inconsistencies. First, the developer externalizes the domain via eD (e.g., at the second `System Revision` via $eD(SR.2)$) that returns a feature model at that point in time. The developer edits the feature model by removing constraints (e.g., by removing the *excludes* constraint $\neg A \vee \neg B$ between features A and B). This increases the configurable space, since features or feature revisions may now be combined in a valid configuration that previously could not. Internalizing the changed feature model via iD triggers an analysis of the configuration space based on a SAT solver. For each individual feature and feature combination, the unified approach checks whether it is supported by the unchanged feature model and the changed feature model, respectively. Specifically, the set H of new features and feature combinations is computed as follows:

$$H = \{h \mid \text{SAT}(FM_{\text{new}} \wedge \neg FM_{\text{old}} \wedge h)\}$$

In case a valid assignment can be found for all features and feature combinations in both feature models, the configurable space has not changed. In case a valid assignment can be found for a particular feature or feature combination only in the unchanged feature model, the configurable space has decreased. Finally, in case a valid assignment can be found for a feature or feature combination only in the changed feature model, the configurable space has increased. In this case, all newly valid features and feature combinations are stored.

Upon eP, newly valid and not yet dealt with features or feature combinations may lead to an inconsistent product. On the one hand, desired feature interactions may be missing and need to be implemented. On the other hand, undesired static or dynamic feature interactions may occur. In the former case, the implementation of one feature may conflict with the implementation of another, e.g., one feature may delete a method that is required by the other feature. The latter is the case when the behaviors of two features interfere, which may lead to undesired behavior of the system. When the developer externalizes a product with a configuration with new features or combinations thereof (e.g., via $eP(SR.1, A.1, B.2, C.1)$), the approach provides a set of respective hints H to the developer in the form of a list of not yet dealt with features and feature combinations that could affect the product:

$$H = \{h \mid h \in H \land \mathrm{SAT}(h \land c)\}$$

The developer becomes aware of the missing feature or possible interaction and may resolve it if necessary, e.g., by providing an implementation particularly for the interaction of features. If the developer internalizes changes via iC with an expression that addresses one of the hinted at features or feature combinations (e.g., via $iC(p1, A \land B)$), the respective hint (e.g., $A \land B$) is removed from the list of hints and will not be provided anymore for the feature or combination of the features that have been specified in the expression e of iC:

$$H' = H \setminus \{e\}$$

Example: In the demonstrative application of the `Car` system (see Section 8.5), inconsistencies of this type occur twice. First, when the feature `Dist` is added to the feature model, as it is missing implementation. Although this does not lead to an inconsistency with respect to syntactic well-formedness, the implementation of the feature should be added to ensure problem space–solution space consistency (see Section 2.7). Second, this inconsistency occurs when the *alternative* group of the features `Gas` and `Ele` is changed to an *or* group in the feature model, which requires to add the pair-wise interaction between $\texttt{Gas}_2$ and $\texttt{Ele}_3$ to realize the `Car` product with a hybrid engine.

8.6.2. Product to Feature Model Consistency

Figure 8.7 shows a UML sequence diagram, and Figure 8.8 an illustrative depiction, of the workflow of the proposed unified approach for Type 5 inconsistencies. First, the developer externalizes a product (e.g., via $eP(SR.1, A.1, B.2)$) and adds a dependency in the solution space between elements of two feature revisions (e.g., an element of $A.1$ refers to an element of $B.2$) that were independent before on both the problem space as well as the solution space. Next, the developer internalizes the performed changes on the product and specifies the feature to which the change applies (e.g., feature A via $iC(p1, A)$). The iC operation triggers a dependency analysis.

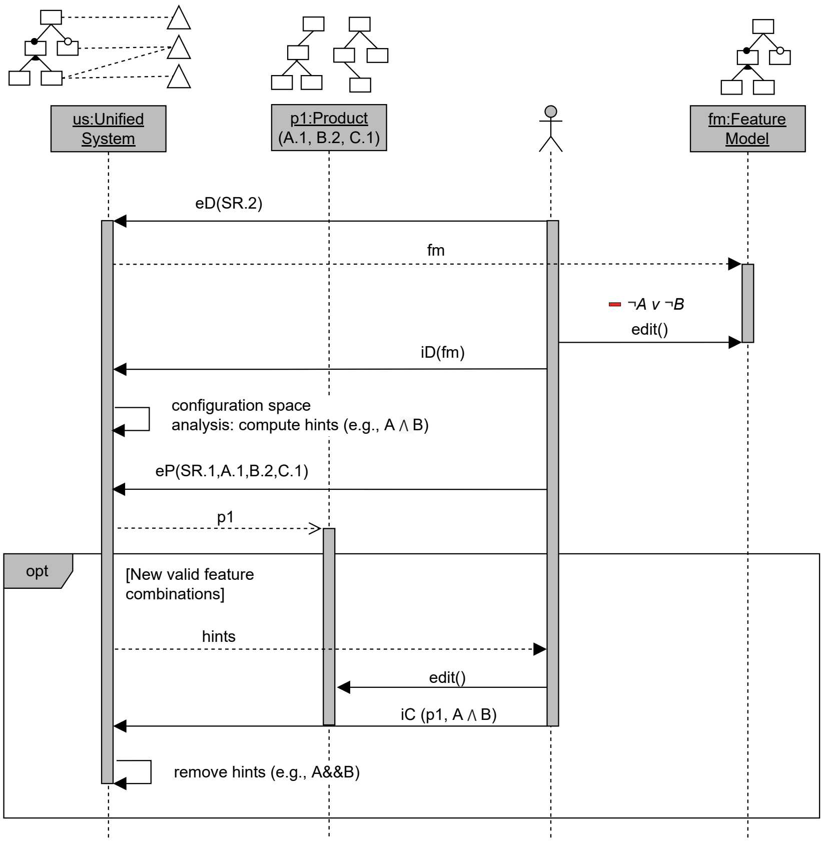

us:Unified System
p1:Product (A.1, B.2, C.1)
fm:Feature Model
eD(SR.2)
fm
¬A v ¬B
edit()
iD(fm)
configuration space analysis: compute hints (e.g., A ∧ B)
eP(SR.1,A.1,B.2,C.1)
p1
opt
[New valid feature combinations]
hints
edit()
iC (p1, A ∧ B)
remove hints (e.g., A&&B)

Figure 8.5.: Sequence diagram of the feature model to product consistency.

In the first step of the analysis, all pairs of deltas are identified that either require or exclude each other based on the elements they affect (e.g., a delta that adds a reference to an element excludes a delta that deletes that element, and requires a delta that creates the element). Then, dependencies are lifted from the solution space (deltas) to the problem space (features) via the mappings between them. Based on the requiring and excluding deltas, the respective mappings that refer to the deltas are identified. Consequently, mappings may require or exclude other mappings. Thus, the expression e_1 (e.g., $A.1$) of a requiring mapping must imply the expression e_2 (e.g., $B.2$) of

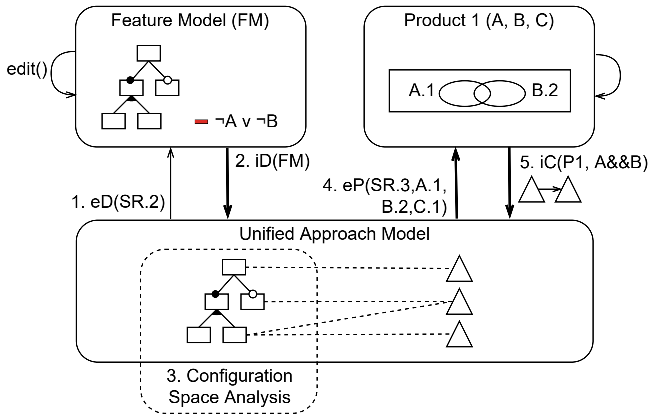


Figure 8.6.: Feature model to product consistency overview.

the required mapping ($e_1 \Rightarrow e_2$), and at least one of the expressions e_3 and e_4 of two exclusive mappings must be false ($\neg e_3 \vee \neg e_4$). A SAT solver is used to check whether the above conditions hold for a feature model FM. Specifically, for a requires relationship to be violated, all clauses of a feature model FM together with requiring expression e_1 and negated required expression $\neg e_2$ must be satisfiable:

$$SAT(FM \wedge e_1 \wedge \neg e_2)$$

If this is the case, there is at least one configuration where the requiring delta is applied without the required delta and the dependency of the solution space is not represented by the problem space. The current feature model is externalized via eD (e.g., at system revision $SR.2$ via $eD(SR.2)$) and the constraint $e_1 \Rightarrow e_2$ is automatically added to the feature model (e.g., $A.1 \Rightarrow B.2$). For an excludes relationship to be violated, all clauses of a feature model together with the two excluding expressions e_3 and e_4 must be satisfiable:

$$SAT(FM \wedge e_3 \wedge e_4)$$

If this is the case, there is at least one configuration where both excluding deltas are applied and the dependency of the solution space is automatically added as constraint to the feature model. The developer may investigate the

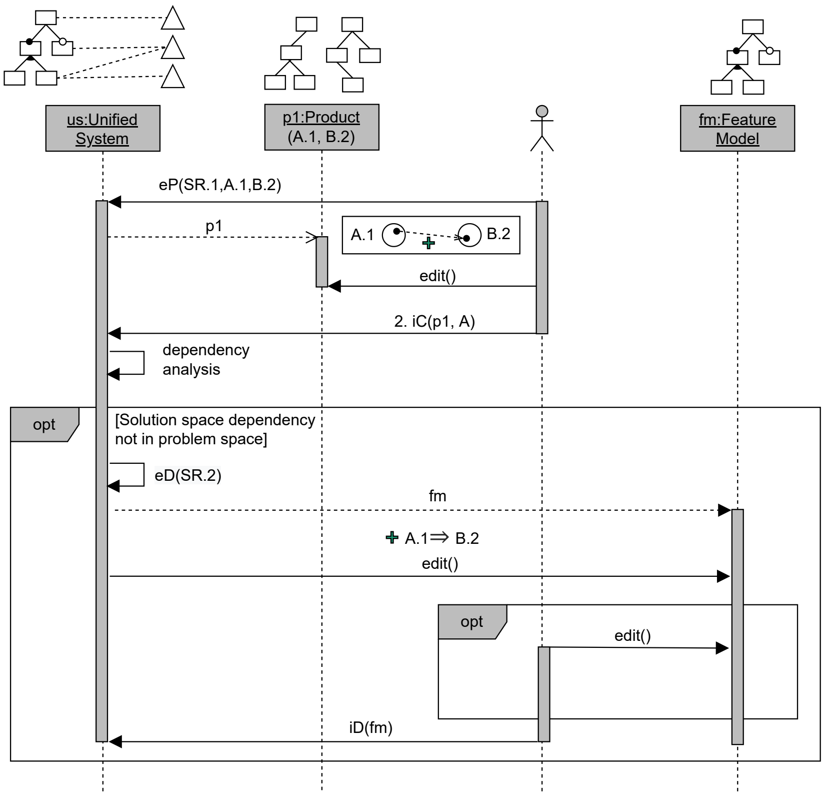


Figure 8.7.: Sequence diagram of the product to feature model consistency.

feature model with the performed repair and perform further modifications before internalizing them into the unified system via *iD*.

Example: In the demonstrative application of the `Car` system (see Section 8.5), inconsistencies of this type occur once. Adding Line 5 and internalizing it with the expression *Ele* leads to a dependency between $\texttt{Ele}_3$ and $\texttt{Dist}_3$ that is not covered by the feature model. The dependency analysis is performed, which leads to the cross-tree constraint $\texttt{Ele}_3 \implies \texttt{Dist}_3$. The constraint is automatically added by the approach to the feature model to resolve the inconsistency between the product and the feature model.

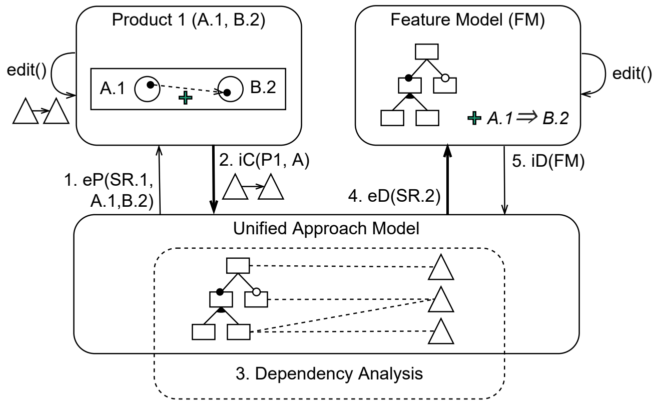


Figure 8.8.: Product to feature model consistency overview.

8.6.3. Product Consistency

Figure 8.9 shows a UML sequence diagram, and Figure 8.10 an illustrative depiction, of the workflow of the unified approach for Type 6 inconsistencies. First, the developer externalizes a product via $eP(SR.1, A.1, B.2)$ and edits it. In case redundant or dependent information across heterogeneous artifact models were modified, e.g., by modifying the Java view of the product, this may lead to inconsistencies of other artifact models. Based on the *Consistency Preservation Rules (CPRs)* of Vitruvius (see Section 2.6.2), changes are propagated to dependent artifact models by (semi-) automatically applying *consequential changes* as reactions to *original changes* performed by the developer.

Next, the developer integrates the performed changes into the system by providing the changed product and specifying the affected feature(s) in the respective expression via $iC(p1, A)$). In this way, a change applied in one product may affect other products. On the one hand, this illustrates an advantage of the unified approach that allows to propagate improvements or fixes made to a feature implementation in one product to all other affected products. Consequently, other products benefit from the changes performed in one particular product. On the other hand, this may lead to inconsistencies in the other products, as the context (i.e., other present features) is different in each product. To summarize, changes are propagated between heterogeneous artifact mod-

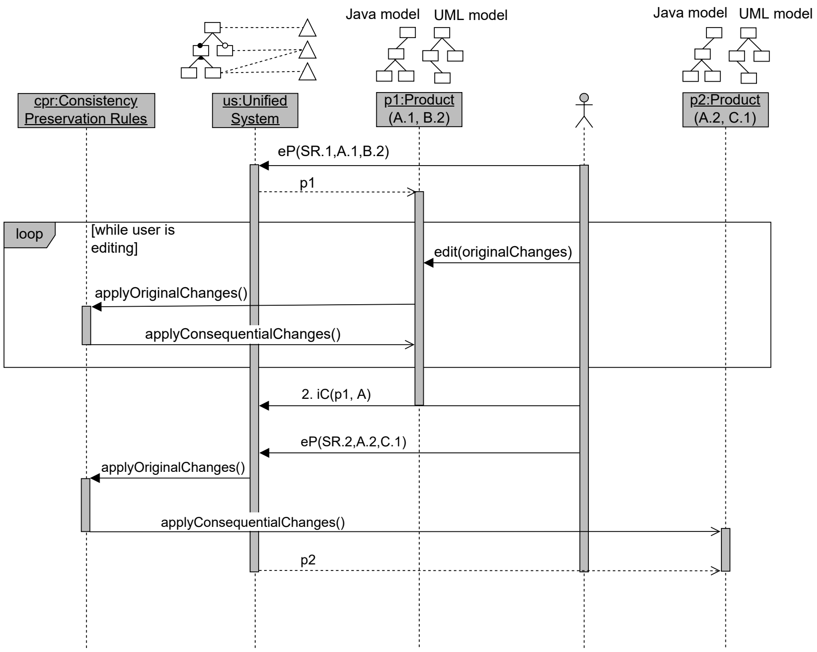


Figure 8.9.: Sequence diagram of the product consistency.

els when one type of artifact is modified via a product view, or when a new product is externalized and changes are propagated that were originally performed on another product.

Example: In the demonstrative application of the `Car` system (see Section 8.5), inconsistencies of this type occur several times. Changes to the Java view that realize the feature `Dist` (Lines 7 and 11) and the interaction between `Gas` and `Dist` (Line 8) are automatically propagated to the SysML view (Line 5). To restore consistency in other affected products that involve the added feature `Dist`, such as $\{\texttt{Car}_1, \texttt{ET}, \texttt{Ele}_1, \texttt{Dist}_2\}$, it is externalized via *eP*. This product is inconsistent, as it lacks a return statement in method *getDistanceLeft()*. Thus, Line 9 is added and internalized with the expression *Ele && Dist*.

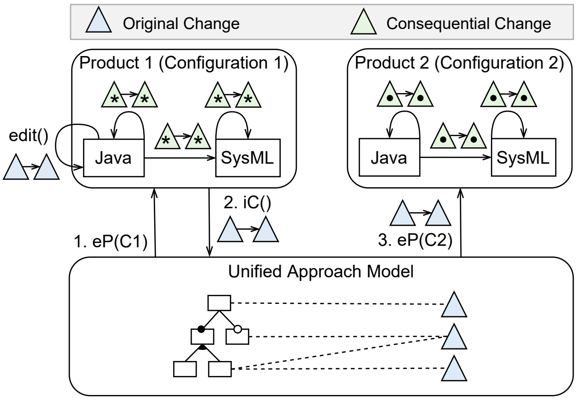


Figure 8.10.: Product consistency overview.

8.7. Prototypical Implementation as VaVe 2.0 Tool

The presented unified approach for supporting the consistent evolution of variable systems composed of heterogeneous artifacts has been prototypically implemented in the `VaVe 2.0` (unified *Va*riants and *Ve*rsions management) tool. Note that `VaVe 2.0` is an extension of the `VaVe` tool (see Section 2.4.2) and realizes the unified approach. VaVe 2.0 has been implemented in Java using the Eclipse Modeling Framework (EMF) [1] (see Section 2.5.2).

The metamodel shown in Figure 8.2 has been implemented as Ecore metamodel. VaVe 2.0 makes use of the Vitruvius framework (see Section 2.6.2) for product derivation and consistency preservation. Specifically, the `Delta Module` in the `VaVe 2.0` metamodel refers to the `EChanges` in the change descriptions metamodel of Vitruvius, which defines a metaclass for each type of change that is possible in Ecore models. Furthermore, the `V-SUM Product` (which represents the `Product` concept of the unified conceptual model) has been implemented as a specialization of the V-SUM of Vitruvius and thus inherits support for multiple heterogeneous artifact models and consistency preservation among them (see Section 8.2.2). For the latter, the unified approach makes use of the Reactions language of Vitruvius (see Section 2.6.2)

[1] `https://www.eclipse.org/modeling/emf/`

that is used for defining unidirectional transformations that preserve consistency between elements of the same or different models. The unified view-based operations were implemented as methods of the `VaVe 2.0 Unified System` according to the specifications given in Listings 8.1, 8.2, 8.3 and 8.4.

8.8. Expected Benefits

The proposed unified approach, its architecture, integration with the Vitruvius approach, workflow and consistency preservation has been explained. This section comprises a discussion on the expected benefits of the unified approach.

Krüger and Berger [127] explain how missing proactive tracking of variability evolution may lead to additional costs. To this end, the unified approach addresses this shortcoming by tracking the revision history of the variable system as well as of its comprised features while explicitly relating both. Consequently, costs can be reduced, the need for retrospective information mining is eliminated, and immediate analyses of evolution history is enabled, i.e., how often a feature changes in the course of a particular time period.

By following product line development based on product views, as encouraged by the upcoming research area of VarCS (see Definition 2.2), limitations of existing variability management practices, such as manually integrating changes into the reusable platform, can be overcome [141, 214]. The user develops the variable system based on a product where variability is fully bound (i.e., each feature is either selected or deselected) while variability is managed fully automatically by employing a variability mechanism internally and hidden from the user. Moreover, other products benefit from improvements performed in one particular product, since these changes are integrated into the reusable platform and, thus, can be propagated to other products. This not only eases the transition from tedious clone-and-own development to a systematic evolution process for variable systems, it also reduces *cognitive complexity* [212, pp. 128] since several tasks are performed fully automatically (e.g., adding a feature automatically creates a new system revision, a new feature revision, links them respectively and computes a corresponding mapping). Thus, the unified approach is expected to reduce the complexity as well as manual effort when managing a variable system.

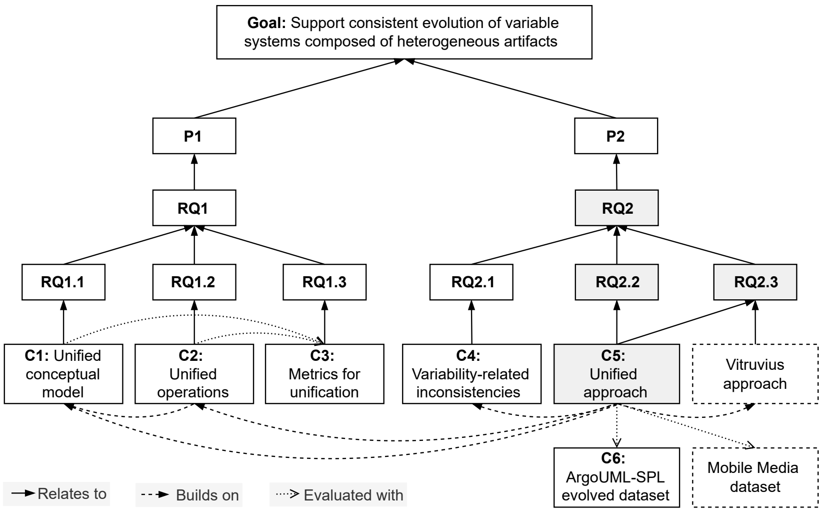

Goal: Support consistent evolution of variable systems composed of heterogeneous artifacts
P1
P2
RQ1
RQ2
RQ1.1
RQ1.2
RQ1.3
RQ2.1
RQ2.2
RQ2.3
C1: Unified conceptual model
C2: Unified operations
C3: Metrics for unification
C4: Variability-related inconsistencies
C5: Unified approach
Vitruvius approach
C6: ArgoUML-SPL evolved dataset
Mobile Media dataset
Relates to
Builds on
Evaluated with

Figure 8.11.: Contribution of Chapter 8 of the thesis.

Finally, the unified approach helps to detect and repair the variability-related inconsistencies caused or repaired in the solution space (i..e, Types 2, 5, and 6), ranging from hints to developers for Type 2 to fully automated repairs for Types 5 and 6. As a consequence, the unified approach is therefore expected to reduce costs and improve the system's quality.

8.9. Summary and Conclusion

This chapter presented the unified approach to support consistent management of variable systems (C5). The unified approach refines the unified conceptual model (C1) and unified operations (C2)), augmenting the unified operations with consistency preservation. `Fragments` of the unified conceptual model are refined using `Deltas` and feature modeling is enabled through refining `Constraints` by `Tree Constraints` and `Cross-tree Constraints`. The evolution of a variable system is supported based on a product or domain view (i.e., the feature model). Consequently, no specific variability mechanisms for different types of artifacts are required anymore since variability is already fully bound in the product. Moreover, multiple tasks are performed fully automatically to support the evolution of a variable system, such as the automated addition of system revisions and feature revisions upon the integration of changes, while also reducing *cognitive complexity* for the developer [212, pp. 128]. Since the unified approach builds on insights from the unification (i.e., C1 and C2), it combines the advantages of the analyzed tools. Additionally, it closes the gap of handling `Feature Revisions` and `System Revisions` simultaneously, which additionally enables it to support systems that vary in space and time holistically.

Moreover, the unified approach supports the consistent evolution of a variable system. It offers consistency preservation for dealing with variability-related inconsistency types that are either caused or repaired in the solution space (i.e., Type 2, 5, and 6).
Adding features or removing constraints between them increases the configurable space and enables new configurations that may not (yet) lead to valid product implementations due to missing implementation of new feature(s) or feature interaction(s) (Type 2). While the unified approach cannot repair such inconsistencies fully automatically, it supports the user in increasing the awareness of potential inconsistencies by providing hints in the form of all new features or new valid feature combinations for which no implementation has been provided yet. If a feature dependency has been added on implementation level that is not captured by the feature model, products with an inconsistent implementation may be externalized (Type 5). The unified approach performs a dependency analysis between deltas of the `Unified System` and, if necessary, lifts the dependencies to the feature model by automatically adding missing constraints. Finally, changing redundant or dependent information across heterogeneous artifacts of the same product can lead to inconsistencies in other artifact types and products due to redundancies or dependencies (Type 6). To this end, the unified approach integrates the consistency preserving mechanism of the Vitruvius approach to propagate changes between dependent artifact models as well as when deriving a product to ensure consistency among all artifact models of the respective product. Particularly, the unified approach employs the `Deltas` used in Vitruvius and specializes its `V-SUM` as `Product`. Thus, consistency preserving mechanisms of Vitruvius are leveraged while it is extended with concepts of the problem space to enable unified variability management.

This contribution addresses RQ 2.2 and RQ 2.3. Figure 6.12 shows an overview of all contributions of the thesis and highlights the contribution of this chapter in grey.

Part III.

Evaluation and Discussion

9. Overview

Chapter 5-Chapter 8 described main contributions of this thesis to support consistent management of variable systems composed of heterogeneous artifacts. This encompassed unified concepts to cope with variability in space and time (C1), unified operations (C2), variability-related inconsistencies that may occur during the evolution of a system (C4) and, finally, the unified approach building upon the preceding contributions (C5).

Part III presents an empirical evaluation of the proposed contributions, structured according to the GQM method [27]. First, the general evaluation process of the unification results is explained in Chapter 10. To this end, metrics for unification are proposed (C3). Subsequently, the conducted evaluation is presented for the unified conceptual model in Chapter 11, for the unified operations in Chapter 12, and for the unified approach in Chapter 13. For each contribution, the evaluation goal, questions, evaluation process and results are discussed.

10. Evaluation Process and Metrics

This chapter builds on publications at SPLC [7] and Empirical Software Engineering [5].

This chapter describes the performed evaluation process in a generalized way for the unified conceptual model (C1) and unified operations (C2) with the purpose of reproducability. The goal of the evaluation is to make reliable statements about whether individual tool elements have been appropriately unified as well as about their applicability.

Referring to problem statement P1, the following research question is asked:

RQ 1.3 How can the appropriateness of a unification with respect to the studied approaches be quantified?

Section 10.1 describes the general evaluation process for the unified conceptual model (C1) and unified operations (C2). Section 10.2 introduces and illustrates the conceived metrics for unification (C3). A summary in Section 10.3 closes this chapter.

This chapter thus constitutes the contribution C3.

10.1. General Evaluation Process

Figure 10.1 shows the general two-step evaluation process of the unification. Based on the construction mappings (between the unified elements and each tool's elements), I conceived metrics for unification. The metrics are based on properties for the evaluation of modeling languages proposed by Guizzardi et al. [90] (Step ⑤). The metric results indicate the appropriateness of the unified elements regarding granularity and coverage with respect to the individual tool elements. Moreover, the application of unified elements is exemplary demonstrated (Step ⑥), which concludes the evaluation process.

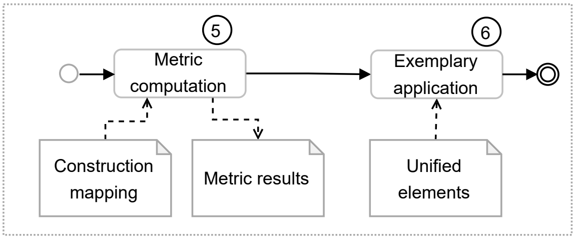
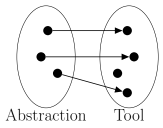


Figure 10.1.: General evaluation process.

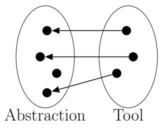
10.2. Metrics for Unification

Evaluating the appropriateness of an abstraction for a diverse set of tools is difficult as the abstraction shall describe a common mental model that suites each analyzed tool. Unfortunately, no techniques are provided in the literature yet to quantify and assess the appropriateness of an abstraction. The closest work is proposed by Guizzardi et al. [90]. The authors introduce a framework to evaluate the appropriateness of modeling languages comprising the properties *laconic*, *lucid*, *complete*, and *sound*. This thesis contributes metrics for evaluating the appropriateness of a unification (C3) based on this framework. Although the same properties are used, their framework is augmented in three ways: First, instead of a tool's language, its abstraction in the form of the tool's structure and operations is considered. Second, metrics are introduced that range from 0.0 to 1.0 to measure to what extent these properties hold for an abstraction (i.e., the unified model and operations) and a tool. Finally, metrics are added to compare an abstraction not just to a single tool but to a set of tools. The metrics *laconicity* and *lucidity* quantify the granularity of the abstraction. The metrics *completeness* and *soundness* quantify their coverage. Each metric is defined for a unification U and a tool $T \in \mathcal{T}$, where $\mathcal{T}$ is the set of studied tools. The unification U is a set of *unified elements* $u \in U$. A tool $T \in \mathcal{T}$ is a set of *tool elements* $t \in T$. $\mathbb{R}_T^U \subseteq U \times T$ is the set of *mappings* of unified elements in U onto tool elements in T. Figure 10.2 shows graphical illustrations of the four metrics. In the following, each metric is defined and an example is provided.

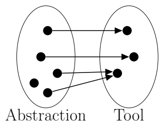
Definition 10.1 presents the metric *laconicity*. As an example, consider Figure 10.2a, where all four tool elements (right side) are laconic. Consequently, the laconicity of the abstraction (left side) with respect to the tool (right side) is 1. In Figure 10.2e, two of the three tool elements (right side) are laconic.



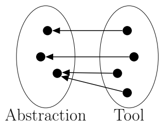
(a) Laconicity = 1.0



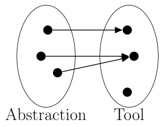
(b) Lucidity = 1.0



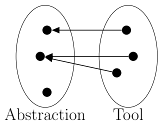
(c) Completeness = 1.0



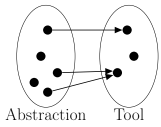
(d) Soundness = 1.0



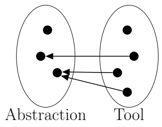
(e) Laconicity = $\frac{2}{3}$ = 0.67



(f) Lucidity = $\frac{2}{3}$ = 0.67



(g) Complete. = $\frac{2}{3}$ = 0.67



(h) Soundness = $\frac{2}{3}$ = 0.67

Figure 10.2.: Overview of the unification metrics. *Abstraction* refers to the unification elements and *tool* refers to a tool's elements [5, Fig. 6].

Consequently, the laconicity of the abstraction (left side) with respect to the tool (right side) is 0.67.

Definition 10.1 (Metric laconicity [5, p. 29]) *A tool's element t is* laconic, *iff it implements at most one unified element u of the unification U.* Laconicity ∈ [0..1] *(higher is better) is then the fraction of* laconic *tool elements:*

$$
\begin{aligned}
laconic(U,T,t) &= \begin{cases} 1 & \textbf{if } |\{u \mid (u,t) \in \mathbb{R}^U_T\}| \leq 1 \\ 0 & \textbf{otherwise} \end{cases} \\
laconicity(U,T) &= \frac{\sum_{t \in T} laconic(U,T,t)}{|T|}
\end{aligned}
$$

Low laconicity indicates that elements of the unification may be too fine-grained, i.e., there are redundant elements in the unification that should be merged.

Definition 10.2 presents the metric *lucidity.* As an example, consider Figure 10.2b, where all four unification elements (left side) are lucid. Consequently, the lucidity of the abstraction (left side) with respect to the tool (right side) is 1. In Figure 10.2f, two of the three unification elements (left side) are lucid. Consequently, the lucidity of the abstraction (left side) with respect to the tool (right side) is 0.67.

Definition 10.2 (Metric lucidity [5, p. 29]) *A unification's element u is* lucid, *iff it is implemented by at most one element t of a tool T.* Lucidity $\in [0..1]$ *(higher is better) is then the fraction of* lucid *unification elements:*

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

Low lucidity indicates that elements of the unification may be too coarse-grained, meaning that there are unspecific elements in the unification that should be split up.

Definition 10.3 presents the metric *completeness*. As an example, consider Figure 10.2c, where all three tool elements (right side) are complete. Consequently, the completeness of the abstraction (left side) with respect to the tool (right side) is 1. In Figure 10.2g, two of the three tool elements (right side) are complete. Thus, the completeness of the abstraction (left side) with respect to the tool (right side) is 0.67.

Definition 10.3 (Metric completeness [5, p. 30]) *A tool's element t is* complete, *iff it is represented by at least one element u in the unification U.* Completeness $\in [0..1]$ *(higher is better) is then the fraction of* complete *tool elements:*

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

Low completeness indicates that the unification may be missing concepts that should be added.

Definition 10.4 presents the *soundness* metric. As an example, consider Figure 10.2d, where all three unification elements (left side) are sound. Consequently, the soundness of the abstraction (left side) with respect to the tool (right side) is 1. In Figure 10.2h, two of the three unification elements (left side) are sound. Consequently, the soundness of the abstraction (left side) with respect to the tool (right side) is 0.67.

Definition 10.4 (Metric soundness [5, p. 30]) *A unification's element u is* sound, *iff it is implemented by at least one element t in the tool T.* Soundness $\in [0..1]$ *(higher is better) is then the fraction of* sound *unification elements:*

$$sound(U,T,u) = \begin{cases} 1 & \textbf{if } |\{t \mid (u,t) \in \mathbb{R}^{U}_{T}\}| \geq 1 \\ 0 & \textbf{otherwise} \end{cases}$$

$$soundness(U,T) = \frac{\sum_{u \in U} sound(U,T,u)}{|U|}$$

Low soundness indicates that the unification may include unused elements that should be removed.

Finally, in Definition 10.5, metrics are specified for a finite set of tools $\mathcal{T}$ to quantify whether the unification is of appropriate granularity and coverage with respect to all selected tools $\mathcal{T}$. Laconicity and completeness for a set of tools are defined such that the properties laconic and complete are evaluated for the union of all tools' constructs. Lucidity and soundness for a set of tools are defined such that respectively all tools (min) and at least one tool (max) must satisfy the corresponding property for each unified element. Note that tool elements that map to the same unification element are not considered as equivalent. Therefore, tool elements are unique (i.e., for all $T_1, T_2 \in \mathcal{T}$ with $T_1 \neq T_2$ it holds that $T_1 \cap T_2 = \emptyset$).

Definition 10.5 (Metrics over a set of tools [5, p. 31]) *A unification element u is* lucid, *if it is* lucid *in all tools $T \in \mathcal{T}$. A unification element u is* sound, *if it is* sound *in at least one tool $T \in \mathcal{T}$:*

$$\overline{laconicity}(U,\mathcal{T}) = laconicity\Big(U, \bigcup_{T \in \mathcal{T}} T\Big)$$

$$\overline{lucidity}(U,\mathcal{T}) = \frac{\sum_{u \in M} \left(\min_{T \in \mathcal{T}} lucid(U,T,u)\right)}{|U|}$$

$$\overline{completeness}(U,\mathcal{T}) = completeness\Big(U, \bigcup_{T \in \mathcal{T}} T\Big)$$

$$\overline{soundness}(U,\mathcal{T}) = \frac{\sum_{u \in U} \left(\max_{T \in \mathcal{T}} sound(U,T,u)\right)}{|U|}$$

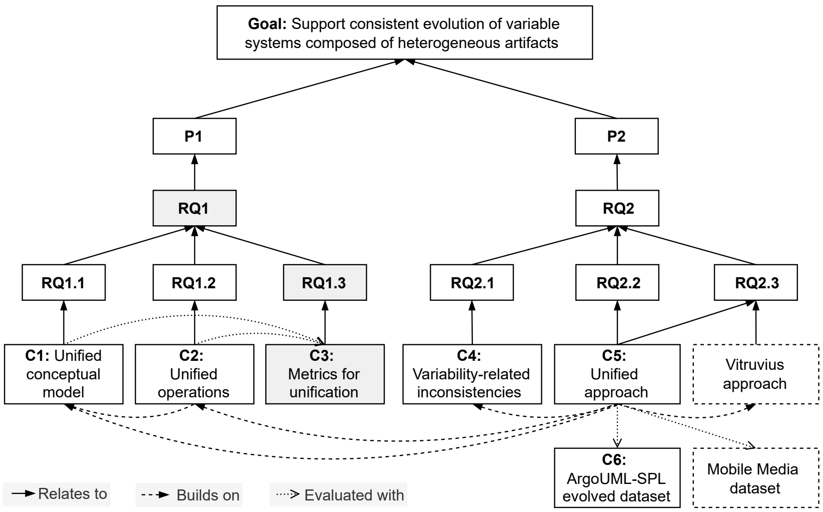

Goal: Support consistent evolution of variable systems composed of heterogeneous artifacts
P1
P2
RQ1
RQ2
RQ1.1
RQ1.2
RQ1.3
RQ2.1
RQ2.2
RQ2.3
C1: Unified conceptual model
C2: Unified operations
C3: Metrics for unification
C4: Variability-related inconsistencies
C5: Unified approach
Vitruvius approach
C6: ArgoUML-SPL evolved dataset
Mobile Media dataset
Relates to
Builds on
Evaluated with

Figure 10.3.: Contribution of Chapter 10 of the thesis.

10.3. Summary

This chapter presented the general evaluation process with respect to the appropriateness and applicability of a conceived unification (i.e., the unified conceptual model and unified operations) performed in this thesis to foster reproducability. The evaluation process comprises the computation of the metrics for unification (C3) as well as an exemplary application of the unification. The metrics have been conceived based on properties from the literature proposed by Guizzardi et al. [90] and quantify the appropriateness of a unification with respect to a set of tools based on the granularity and coverage of the unified elements.

Thus, this contribution addresses RQ 1.3. Figure 10.3 shows an overview of all contributions and highlights the contribution of this chapter in grey.

11. Evaluation of the Unified Conceptual Model

This chapter builds on publications at SPLC [7] and Empirical Software Engineering [5]. An open-access repository comprises all artifacts related to the construction and evaluation of the unified conceptual model.[1]

Chapter 5 presented the unified conceptual model for variability in space and time (C1). This chapter presents its evaluation based on the GQM method [27].

Section 11.1 introduces the goals and questions of the evaluation. Section 11.2 presents the specialized evaluation process of the unified conceptual model. Section 11.3 encompasses the first part of the evaluation which comprises a qualitative analysis based on an expert survey. A quantitative analysis follows in Section 11.4 and uses the unification metrics introduced in Section 10.2. The second part of the evaluation demonstrates an application of the unified conceptual model in Section 11.5. Degrees of freedom for refining the conceptual model are presented as well as the derivation of two exemplary tools. Additionally, Section 11.6 presents a formal concept analysis to provide additional insights of the unification. Section 11.7 offers answers to the posed questions while threats to validity are considered in Section 11.8. Section 11.9 comprises a discussion of the limitations and future work of the unified conceptual model. A summary of the main insights in Section 11.10 closes this chapter.

[1] `https://doi.org/10.5281/zenodo.5751916`

11.1. Goals and Questions

The unified conceptual model aims at appropriately unifying concepts and relations for variability in space, time, and both based on the selected tools. The following three goals regarding granularity, coverage and applicability shall be met by the conceptual model:

***Granularity*:** The unified conceptual model shall describe concepts with appropriate granularity that are neither unnecessarily fine-grained nor unnecessarily coarse-grained.

***Coverage*:** The unified conceptual model shall comprise all concepts and relations used by the tools selected for unification, but no more than necessary.

***Applicability*:** The unified conceptual model shall serve as foundation to derive new tools that are able to cope with variability in space and time.

Based on these goals, the following questions are asked:

Q 1.1 To what extent is the unified conceptual model of appropriate granularity?

Q 1.2 To what extent is the unified conceptual model of appropriate coverage?

Q 1.3 To what extent is the unified conceptual model applicable?

11.2. Specialized Evaluation Process

Figure 11.1 shows the evaluation process of the unified conceptual model. It follows the construction process shown in Figure 5.1 and extends the general evaluation process (see Section 10.1) by a qualitative analysis that encompasses an expert survey. Step ⑤ represents the expert survey based on questionnaires with the unified conceptual model (as input) which results in the validation mapping (as output). The mapping is input to the quantitative analysis in Step ⑥ where metrics for unification are computed (see Section 10.2). Finally, Step ⑦ comprises an exemplary application of the unified conceptual model based on two illustrating tools to demonstrate possible refinements as well as the computation of the unification metrics.

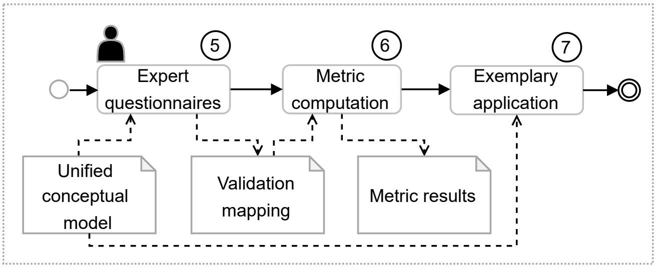


Figure 11.1.: Evaluation process of the unified conceptual model. Adapted from [5, Fig. 5].

Qualitative analysis: The qualitative analysis comprised an expert survey with one expert per tool. I conducted the expert survey based on questionnaires. Each expert received a questionnaire to create a mapping between tool constructs, their relations as well as well-formedness rules and the concepts, relations and well-formedness rules of the unified conceptual model. Additionally, experts were asked to document all tool constructs and relations that could not be mapped to the unified conceptual model.

Quantitative analysis: The quantitative analysis involved an application of the metrics for unification (see Section 10.2). I computed the metrics for the unified conceptual model based on the created mappings for each tool and, in addition, an aggregated metric value over all tools. The metrics *laconicity* and *lucidity* quantify the granularity of concepts (i.e., whether the concepts are as specific as possible while still being generic enough). The metrics *completeness* and *soundness* quantify the coverage of concepts and relations.

Exemplary Application: To demonstrate applicability, I created two fictive tools based on the unified conceptual model. Degrees of freedom for refining the unified conceptual model are explained as well as the design choices of each tool. Additionally, the metrics for unification are computed for each tool to illustrate them in greater detail.

11.3. Qualitative Analysis

In the following, Step ⑤ of Figure 11.1 is described encompassing the expert questionnaires and the resulting validation mappings.

11.3.1. Expert Questionnaire

For each tool, one expert completed a questionnaire for mapping the tool's constructs, relations, and well-formedness rules to the concepts, relations and well-formedness rules of the unified conceptual model. The questionnaire guide consisted of three parts. First, the unified conceptual model along with a definition of each concept and relation was introduced. Second, the guide asked for a mapping between each concept and relation of the conceptual model and a semantically equivalent construct and relation of the respective tool (also asking for constructs and relations that could not be mapped by the tool experts). Finally, well-formedness rules of the conceptual model were presented, asking for a mapping to well-formedness rules employed by the respective tool.

11.3.2. Validation Mapping

The validation mappings were derived from the completed expert questionnaires. To obtain an equivalent comparison between the tools and the model, the mappings were performed on the conceptual level of the tools. Since abstract concepts (such as `Option` and `Revision`) cannot be instantiated, they were not considered in a mapping.

11.3.3. Results

The validation mappings of concepts, relations, and well-formedness rules are shown in Table 11.1, Table 11.2, and Table 11.3.

Table 11.1 shows the mapping of unified model concepts (rows) to the constructs of each tool (columns). While all tools employ constructs for the five concepts `Unified System`, `Fragment`, `Mapping`, `Configuration`, and `Product`, the used terminology and their semantics differ considerably across the tools. For example, the concept `Fragment` is realized by the constructs `Blob` (file content) and `Tree Object` (directory) in the tool Git, while equivalent constructs in SVN are called `File Node` and `Directory Node`. SuperMod and ECCO refer to `Fragment` respectively as `Product Element` and `Artifact`. Moreover, delta-oriented tools (i.e., SiPL, DeltaEcore, DarwinSPL, VaVe) use the `Core Model` and `Delta Module` as `Fragments`. In some cases, the terminology used by the

Table 11.1.: Validation Mapping: Results of mapping the constructs of each tool to the concepts in the conceptual model [5, Tab. 1].

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*)	–	–	–	Revision	Commit	–	Revision	–	Temporal Validity	–
Feature Revision (*FR*)	–	–	–	–	–	Revision	–	Version	–	Version
Configuration (*C*)	Variant, Configuration	Configuration	Configuration	Revision	Commit	Configuration	Choice	Configuration	Configuration	Configuration[1]
Constraint (*CT*)	Constraint	Constraint, Relation	Constraint	–	–	–	Dependency	Constraint	Constraint	Constraint

[1]The concept exists at the conceptual level of the tool without an explicit construct in the implementation.

different tools is almost uniform. For instance, seven tools use the term `Configuration` and six tools use the term `Product`. However, across all tools, there are still five different terms for `Configuration` and three different terms for `Product`. This shows that there is quite some disparity in terminology even for the constructs with the highest consensus among the tools. Particularly interesting to note are the cases where different tools have constructs with the same name but with different semantics that map to different model concepts. For example, the tools VaVe, ECCO and FeatureIDE all have a construct named `Variant`. However, in each tool, it maps to another model concept, namely to `Feature`, `Product` and `Configuration`, respectively. Consequently, even within just the SPLE community there are overloaded terms that are used to refer to different concepts. Furthermore, one can see how the studied tools cover concepts for variability in space and/or time. Git and SVN use `System Revisions` for variability in time, while FeatureIDE, pure::variants, and SiPL use `Features` and `Constraints` for variability in space. All remaining tools (i.e., ECCO, SuperMod, DeltaEcore, DarwinSPL, VaVe) cope with both variability dimensions and involve the concept `Feature` in addition to `System Revision` or `Feature Revision`. None of the studied tools deals both variability dimensions while also considering `System Revision` and `Feature Revision`, as described in Section 5.2.2. Finally, a `Mapping` is understood equivalently across all tools by connecting `Fragments` and `Options`. While for tools coping with variability in time a `Mapping` simply links a `System Revision` to `Fragments`, the `Mapping` becomes more complex for tools that (additionally) consider variability in space, as `Fragments` can potentially be linked to any number of `Features`.

Table 11.2 shows whether relations among concepts of the unified conceptual model (rows) also exist among the respective constructs of each of the studied tools (columns). All tools employ the five relations: `Fragment has * Fragment`, `Mapping has * Fragment`, `Configuration has * Option`, `Unified System has * Fragment` and `Unified System has * Mapping`. Whether the remaining relations are supported by a tool or not depends on the supported variability dimension. Moreover, the unified conceptual model lacks the `remotes` relation that is used by the tools Git and ECCO by which repositories (i.e., `Unified Systems`) can refer to each other to support distributed development.

Table 11.3 shows the mapping of the well-formedness rules (see Section 5.2.4) of the unified conceptual model to each tool. A tool either satisfies a rule by construction (■), enforces it at all times (●), evaluates it but does not enforce

Table 11.2.: Validation Mapping: Results of mapping the relations in each tool to the relations in the conceptual model [5, Tab. 2].

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.

Table 11.3.: Validation Mapping: Results of mapping the well-formedness rules of the conceptual model to each tool [5, Tab. 3].

Tool / Rule	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.

it (◒), does not evaluate it (○), or the rule does not apply for the tool (–). If a tool *satisfies* a rule *by construction*, it is not necessary to enforce or evaluate it as the tool's structure makes it impossible to create a violating state. For instance, Rule 3 (see Listing 5.2) is satisfied by construction in SVN and Git, since they only employ `System Revisions` for which the `Repository` is the only container. If a tool *enforces* a rule *at all times*, it guarantees its fulfillment by means of checks. If a rule is violated, the causing change is prohibited upfront or the system's state is repaired. To this end, DeltaEcore enforces Rule 4 (see Listing 5.3) at all times by evaluating and ensuring the well-formedness of a configuration upon its creation. If a tool *evaluates* a rule but does *not enforce* it, it only checks if the rule is violated, but does not enforce it by additional actions if it is. For example, pure::variants evaluates Rule 5 (see Listing 5.3), but supports the import of external `Fragments`. Moreover, some tools *neither evaluate nor enforce* some rules even though they would be applicable. For instance, FeatureIDE neither evaluates nor enforces Rule 5 and therefore allows external `Fragments` to be used. Finally, a rule *does not apply*, if a tool neither employs concepts nor relations the rule refers to. For instance, Rules 1–3 and 7–10 cannot be applied to tools that do not deal with variability in time.

SuperMod, that supports System Revisions, satisfies most of the rules by construction. For instance, it ensures Rule 1 and Rule 2 by employing a linear sequence of revisions (instead of a revision graph) where every new revision is appended at the end of the sequence. DarwinSPL achieves similar mapping results while also using System Revisions, but enforces most rules. Furthermore, across all tools, either all or no well-formedness rules of the revision graph (i.e., Rules 1–3) are satisfied. Finally, Rule 9 is not applicable to any tool, since no tool supports both System Revisions and Feature Revisions.

11.4. Quantitative Analysis

In the following, Step ⑥ of Figure 11.1 is described. It encompasses an application of the metrics for unification (see Section 10.2).

11.4.1. Metrics

The metrics *laconicity* (see Definition 10.1) and *lucidity* (see Definition 10.2) quantify the granularity of concepts and relations of the unified conceptual model (Q 1.1) with respect to the elicited tools. The metrics *completeness* (see Definition 10.3) and *soundness* (see Definition 10.4) quantify their coverage (Q 1.2).

The unification U is the unified conceptual model M that is a set of *model concepts* $m \in M$. A tool $T \in \mathcal{T}$ is a set of *tool constructs* $t \in T$. Relations are considered as concepts and constructs, too. The *mappings* of unified model concepts and relations in M onto tool constructs and relations in T are displayed in Tables 11.1 and 11.2, respectively.

The conceptual model M 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) yields the set

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

In the following, an example for each metric is shown by applying it to DeltaEcore. For simplicity, in the example, we only consider unified conceptual model concepts and tool constructs and not their relations. The metrics are thus computed based on the mapping in Table 11.1. DeltaEcore implements nine constructs:

$$\begin{aligned} T_{\text{DeltaEcore}} = \{&\text{Core Model, Delta Module, Product, Product Line,} \\ &\text{Mapping Model, Feature, Version, Configuration, Constraint}\} \end{aligned}$$

Laconicity No tool construct in DeltaEcore maps to more than one model concept, which indicates that the unified conceptual model is not unnecessarily fine-grained. Thus, the laconicity with respect to DeltaEcore is ideal:

$$\text{laconicity}_{\text{DeltaEcore}}(M, T_{\text{DeltaEcore}}) = \tfrac{1+1+1+1+1+1+1+1+1}{9} = \tfrac{9}{9} = 1.0$$

Lucidity The model concept `Fragment` maps to the two constructs `Core Model` and `Delta Module` in DeltaEcore. All other model concepts are either not implemented by any construct or by exactly one construct in DeltaEcore and have no impact on the value of the metric. Thus, the unified conceptual model is slightly more coarse-grained and generic than DeltaEcore and lucidity is fairly high:

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

Completeness There are no constructs in DeltaEcore that do not map to any model concept, i.e., all its constructs correspond to at least one model concept. This indicates that the unified conceptual model is not missing any concepts to fully cover DeltaEcore. The completeness with respect to DeltaEcore is thus ideal:

$$\text{completeness}_{\text{DeltaEcore}}(M, T_{\text{DeltaEcore}}) = \tfrac{1+1+1+1+1+1+1+1+1}{9} = \tfrac{9}{9} = 1.0$$

Soudness The model concept `System Revision` cannot be mapped to any construct in DeltaEcore. All other model concepts are implemented by at least one construct in DeltaEcore. This indicates that there is one concept in the model that is not required by DeltaEcore. The soundness of the unified conceptual model with respect to DeltaEcore is thus fairly high, albeit not ideal:

$$\text{soundness}_{\text{DeltaEcore}}(M, T_{\text{DeltaEcore}}) = \tfrac{1+1+1+1+1+0+1+1+1}{9} = \tfrac{8}{9} = 0.889$$

11.4.2. Results

Table 11.4 shows the values of the four metrics (columns) for each tool (rows). The values are presented separately for concepts and relations of variability in space, time, both, and unified concepts and relations as well as in total. In case of lucidity and soundness, each row shows the percentage and the absolute number of conceptual model concepts and relations that satisfy each property. In case of laconicity and completeness, each row shows the percentage and the absolute number of tool constructs and relations that satisfy each property. A horizontal line indicates that a tool does not support a particular variability dimension or their combination. For instance, the tool SiPL supports variability in space but not variability in time (and, consequently, also no concepts and relations of both dimensions, such as the Feature Revision). The values for laconicity, lucidity, and completeness are between 92% and 100% for all analyzed tools. For example, lucidity of the conceptual model with respect to Git is 96%, as the concept Fragment maps to the two constructs Tree Object and Blob. As another example, the conceptual model is 93 % laconic with respect to SVN, because the Revision Number represents both the model concepts System Revision and Configuration. Since there is no tool that implements *all* concepts and relations of the unified conceptual model, the soundness values are generally lower. Thus, for tools that do not support one of the variability dimensions or their combination, the soundness value is even zero. This is, for example, the case for all tools that do not support variability in time (i.e., the concept System Revision and the two respective relations Unified System has * System Revisions and Revision has * Successor and * Predecessor). Vice versa, tools that only support variability in time (i.e, SVN, Git) have a soundness value of zero for variability in space. Moreover, tools that support variability in space and time via Feature Revisions but not via System Revisions reach a soundness value of 50% for variability in both dimensions, since they support the Feature Revision concept and the containment relation to the Feature, but not the two enables relations of the System Revision.

Table 11.5 shows the aggregated results over all tools. The four metrics are shown as columns for concepts/constructs and relations and concepts/constructs and relations for variability in space, time, both, and unified as rows. Metric values are at or close to 100%. The lower value for laconicity is due to the construct Commit in Git and the contruct Revision in SVN that each represent the two concepts Configuration and System Revision. Note that

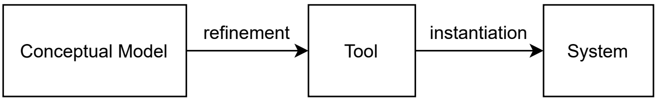


Figure 11.2.: Application stages of the conceptual model [5, Fig. 9].

the mapping to `Configuration` is debatable, since `Git` and `SVN` do not have an explicit construct for a `Configuration`, as it would simply consist of a single commit hash or revision number, respectively. The lower value for laconicity is due to several constructs representing the `Fragment` concept. For example, delta-oriented tools employ the constructs `Core Model` and `Delta Modules` to represent the `Fragment`. Considering completeness, self-relating repositories (such as in `Git` and `ECCO`) are not represented by the conceptual model. Finally, the conceptual model is entirely sound over all tools, as for every concept and relation there is at least one tool that implements it.

11.5. Exemplary Application

This section describes Step ⑦ shown in Figure 11.1 to demonstrate the applicability of the unified conceptual model. First, possible refinements of the conceptual model are explained when designing a conforming tool. Then, two exemplary tool metamodels are introduced by refining the unified conceptual model: the first tool uses a feature model and `Feature Revisions`, while the second tool employs both `System Revisions` and `Feature Revisions`. Finally, the computation of the unification metrics for both tools is demonstrated.

Figure 11.2 shows the two subsequent application stages for applying the conceptual model. In the first step, the conceptual model is refined by specifying abstract model concepts, such as `Fragments`, with concrete constructs, such as `Deltas`. In the second step, the resulting tool is instantiated for a system. While tool developers perform the refinement step, users implicitly perform the second step by applying the tool. UML class diagrams are used to illustrate the result of the first step while UML object diagrams are used to illustrate the result of the second step for both tools.

Table 11.4.: Metric Results for each tool individually [5, Tab. 4].

	for	laconicity	lucidity	completeness	soundness
Feature IDE	*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)
Super Mod	*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)
Delta Ecore	*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)
Darwin SPL	*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 11.5.: Metric results over all tools [5, Tab. 5].

Kind	for	laconicity	lucidity	completeness	soundness
Concept/ Construct	*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)
Relation	*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)

11.5.1. Refinement Process of the Conceptual Model

To develop a conforming tool, the developer has to decide between several degrees of freedom. Specifically, concrete subclasses must extend non-abstract concepts. For example, if a tool uses feature modeling, the `Constraint` concept may be refined by concrete subclasses to represent alternative or mandatory `Features`. Abstract concepts such as `Feature Option`, `Option`, and `Revision` are not supposed to be specialized when designing tools based on the unified conceptual model. In the following, the degrees of freedom for refining and applying the conceptual model are introduced.

Revision. The `Revision` concept can be refined by `Feature Revisions`, `System Revisions` or a combination of both revision types.

Constraint. The `Constraint` can be extended by subclasses and refined by attributes, e.g., to express optional `Features` or cross-tree constraints in a feature model.

Fragment. `Fragments` can be refined analogously to `Constraints`. For instance, by employing a `Core Model` and `Delta Modules` in case a transformational variability mechanism is used to derive a `Product`.

Mapping. This concept can be refined to express relations between `Options` and `Fragments`. For instance, by using Boolean expressions.

Configuration. The refinement of the `Configuration` concept is optional. Developers may use this concept directly as it is in the conceptual model (i.e., a set of references to `Options`) or refine it similar to a `Mapping`. Also, its containment in the `Unified System` is considered optional for representing non-persistent `Configurations`.

In the following, two exemplary tool refinements are introduced encompassing design choices, the resulting metamodels, a computation of the metrics for unification and an exemplary instantiation based on the running example of the `Car` system.

11.5.2. Feature-Revision / Transformational Tool T_T

Figure 11.3 shows the metamodel for the exemplary tool T_T resulting from refining the conceptual model. It employs a transformational variability mechanism.

Design Decisions. The tool T_T was created by extending and refining the unified conceptual model as described above. Added concepts are highlighted with a hatched area and are the `Change`, `Delta Module`, `Expression`, `Cross-tree Constraint`, and `Tree Constraint`). Unused concepts of the conceptual model are highlighted in red (i.e., `System Revision` and its relations). Identical concepts and relations to the conceptual model are depicted as they are within that model. Furthermore, attributes were added (e.g., *value*, or *id*) to several concepts. The following design decisions were incorporated to derive the tool's metamodel.

Revision: Feature Revisions. `Feature Revisions` are employed as the only type of revision. While incorporating `System Revisions` jointly with `Feature Revisions` offers many advantages, for this example, both types of revisions were not combined to reduce complexity.

Constraint: Feature Model. The `Constraint` concept is specialized by the two subclasses `Tree Constraint` and `Cross-tree Constraint`. `Tree Constraints` can only refer to `Features`. `Cross-tree Constraints` can refer to `Feature Options` (i.e., `Features` and `Feature Revisions`) via a Boolean expression represented as a tree where nodes are operators

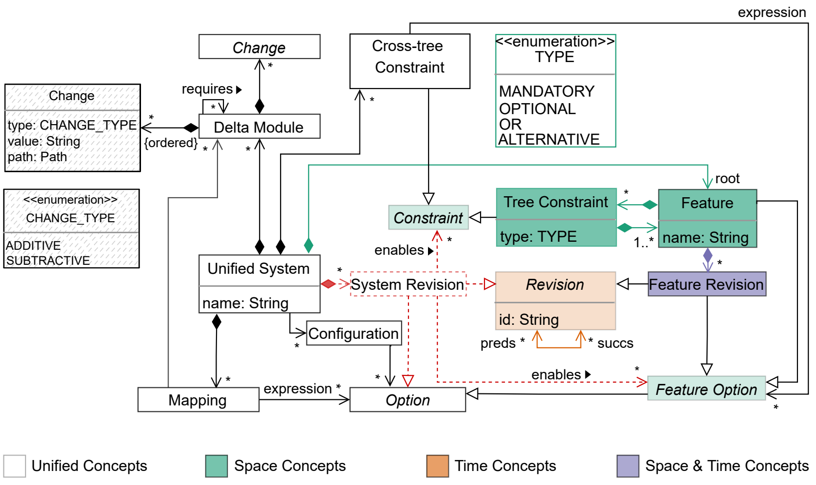


Figure 11.3.: Feature-revision / transformational tool model T_T [5, Fig. 10].

and leafs are Feature Options. For space reasons, these details are omitted from the metamodel.

Fragment: Delta Module, Change. In the exemplary instantiation, deltas are used to implement Fragments and to compose Products. Therefore, the new constructs Delta Module and Change are added to the metamodel. A delta module comprises an ordered cohesive set of changes and can require other delta modules. A change can be of additive or subtractive nature and thus be used to add or delete a Fragment (i.e., *value*) at a certain position (i.e., *path*). A String value can be used for textual Fragments, although the value is not limited to text.

Mapping: Boolean Expression. The relation between a Mapping and Options is represented with a Boolean expression.

Configuration: No Containment in Unified System. The Configuration is not considered as part of the Unified System and therefore not contained in the tool's metamodel.

Metrics Computation.

The metrics for unification (see Section 11.4.1) are computed for the unified conceptual model with respect to the constructs of T_T. It defines nine non-abstract constructs: T_T = {`Unified System`, `Feature`, `Tree Constraint`, `Cross-tree Constraint`, `Feature Revision`, `Configuration`, `Mapping`, `Delta Module`, `Change` } (note that enumeration types are not considered and thus ignored for computation).

Since there is not a single construct in T_T that can be mapped to more than one concept, the laconicity of the unified conceptual model with respect to T_T is:

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

The two constructs `Tree Constraint` and `Cross-tree Constraint` implement the model concept `Constraint`, while the remaining tool constructs correspond to at most one model concept. Thus, lucidity of the unified conceptual model with respect to T_T is:

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

The constructs `Unified System`, `Feature`, `Tree Constraint`, `Cross-tree Constraint`, `Feature Revision`, `Configuration`, `Mapping` and `Delta Module` map to at least one model concept. Note that the construct `Delta Module` can be considered to map to `Fragment` (as it represents a specialization). However, the construct `Change` of T_T does not implement any model concept. Consequently, the completeness of the unified conceptual model with respect to T_T is:

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

While T_T implements the seven model concepts `Unified System`, `Feature`, `Constraint`, `Feature Revision`, `Configuration`, `Mapping` and `Fragment`, there are no corresponding constructs that represent the two model concepts `System Revision` and `Product`. Consequently, the soundness of the unified conceptual model regarding the tool T_T is:

$$\text{soundness}_{T_T}(M, T_{T_T}) = \frac{7}{9} = 0.778$$

In sum, T_T refines model concepts, which leads to lower lucidity. Furthermore, it adds constructs, which lowers the completeness value. Moreover, not all model concepts are employed, which lowers soundness. Since every construct maps to at most one concept, the laconicity value remains ideal.

Instantiation.

Figure 11.4 depicts an exemplary instance of the tool T_T's metamodel for an excerpt of the `Car` example. The `Unified System` (named `Car`) contains a `Mapping`, `Cross-tree Constraint` and the root feature `Car`. The feature instances `EngineType` and `Distance` are children of the `Car` feature. `EngineType` is a mandatory feature (due to the `Tree Constraint` instance of type *mandatory*) and `Distance` an optional feature (due to the `Tree Constraint` instance *optional*). The feature instances `Gasoline` and `Electric` are children of the `EngineType` feature (due to a `Tree Constraint` instance of type *or*). While the features `Car` and `Gasoline` have one `Feature Revision`, the feature `Electric` and `Distance` exist in three subsequent `Feature Revisions`. Moreover, the `Unified System` comprises a `Cross-tree Constraint` with the expression $\neg Electric.3 \lor Distance.3$ that requires a valid `Configuration` with the feature `Electric` in its third revision to specify the `Distance` feature in its third revision. The `Mapping` comprises the expression $Car.1$, which refers to the first revision of feature *Car*, and a `Delta Module` instance which consists of two `Change` instances that are of type *additive*. The line "`class EngineController`" is added to Line 1 of the file `Car.java` by `Change c1`, while `Change c2` adds the line "`void doDriving()`" to Line 4 of the same file. A `Configuration` instance refers to the first revision of features `Car`, `Distance`, and `Gasoline`.

11.5.3. Both-Revisions / Compositional Tool T_C

For the second exemplary tool T_C, quite different design decisions were followed. The goal was to create T_C as close as possible to the conceptual model with minimal specialization. Moreover, the tool T_C employs a compositional variability mechanism and combines both `System Revisions` and `Feature Revisions`. Figure 11.5 depicts the tool's metamodel.

Design Decisions.

The tool's T_C metamodel was refined by the constructs `Mapping Expression` and `Constraint Expression`. The `Mapping Expression` is part of the `Mapping`

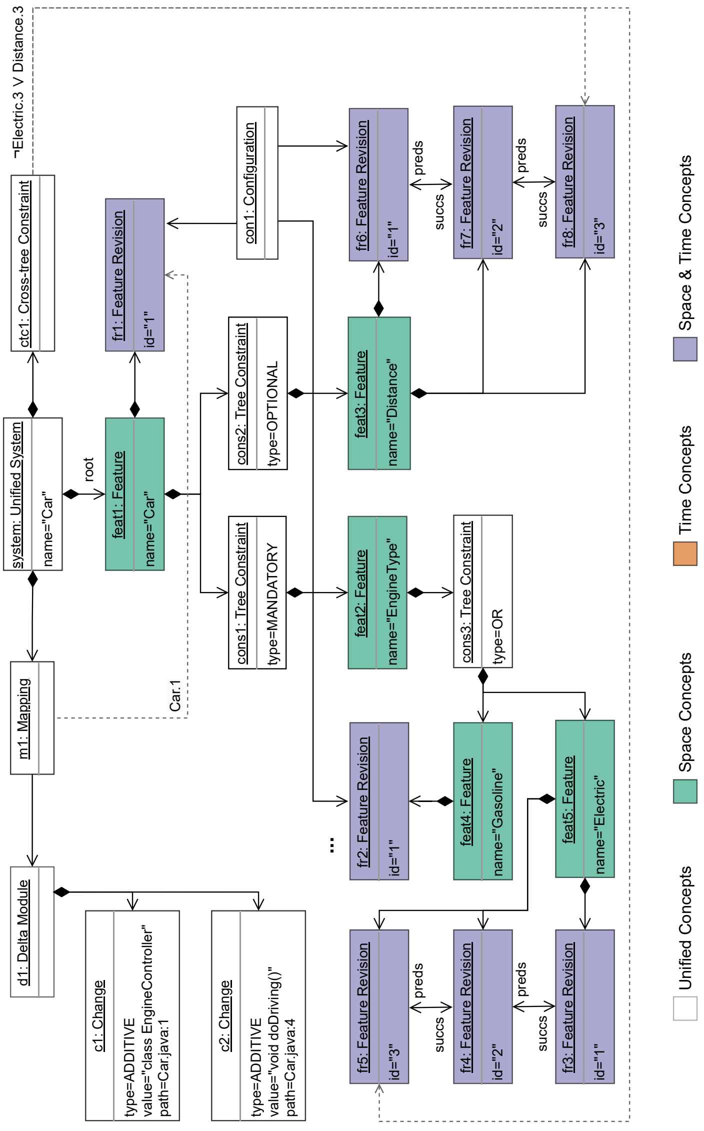


Figure 11.4.: Object diagram of tool T_T applied to the `Car` example [5, Fig. 11].

and refers to `Options`. The `Constraint Expression` is comprised by the `Constraint` and can be defined over `Feature Options`, respectively. Both expression types represent Boolean expressions in the form of expression trees, which are omitted for the sake of simplicity. Nodes represent Boolean operators (such as *implication* or *negation*). Leafs represent literals (i.e., `Options` in case of `Mappings` and `Feature Options` in case of `Constraints`). Furthermore, selected and deselected `Options` in `Configurations` are distinguished. In contrast to T_T, a `Product` directly contains the `Fragments` (instead of being derived by them, as is the case with `Delta Modules` as `Fragments`. The tool's metamodel is based on the following design decisions.

Revision: Feature Revisions and System Revisions. `System Revisions` and `Feature Revisions` are employed jointly to explicitly manage both revision types.

Constraint: Boolean Expression. To represent `Constraints`, Boolean expressions are used with `Feature Options`. Therefore, the new construct `Constraint Expression` relates `Constraints` and `Feature Options`.

Fragment: Implementation Artifacts. `Fragments` are represented by implementation artifacts (e.g., source code). Based on a compositional variability mechanism, implementation artifacts are added in an arbitrary order to obtain a valid `Product`. This is in contrast to the transformational variability mechanism of tool T_T using deltas, which must be applied in a specific order to construct a `Product` and that are not directly contained by the `Product`.

Mapping: Boolean Expression. To represent `Mappings`, Boolean expressions are used with `Options`. Thus, the new construct `Mapping Expression` relates `Mappings` and `Options`.

Configuration: Selected and Deselected Options. The concept `Configuration` is refined by means of distinguishing between explicitly *selected* and *deselected* `Options`. Also, it is possible to leave `Options` undecided for expressing partial `Configurations`.

Metrics Computation. The metrics for unification (see Section 11.4.1) are computed for the unified conceptual model with respect to the constructs of T_C. It defines eleven constructs: T_C = { `Unified System`, `Feature`, `Feature Revision`, `System Revision`, `Constraint`, `Constraint Expression`, `Mapping Expression`, `Mapping`, `Configuration`, `Fragment`, `Product` }.

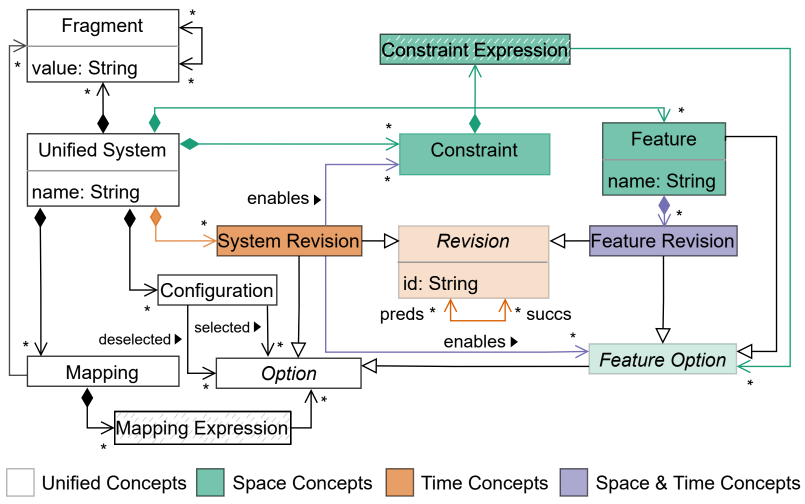


Figure 11.5.: Both-Revisions / Compositional tool model T_C [5, Fig. 12].

Since there is not a single construct in T_C that can be mapped to more than one concept, the laconicity of the unified conceptual model with respect to T_C is:

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

Moreover, there is not a single model concept that can be mapped to more than one construct in T_C. Thus, the lucidity of the unified conceptual model with respect to T_C is:

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

While the eight T_C constructs `Unified System`, `Feature`, `Feature Revision`, `System Revision`, `Constraint`, `Configuration`, `Mapping`, `Fragment`, and `Product` map to at least one model concept, the constructs `Constraint Expression` and `Mapping Expression` cannot be mapped to any model concept. Thus, completeness of the unified conceptual model with respect to T_C is:

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

Since all nine model concepts can be mapped to at least one construct in T_C, the soundness of the unified conceptual model regarding tool T_C is:

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

In summary, only the added construct `Expression` (for defining `Mappings` and `Constraints`) results in lower completeness, while the remaining metric values remain ideal.

Instantiation. Figure 11.6 shows an instance of the tool T_C's metamodel for an excerpt of the `Car` example. An instance of the `Unified System` named `Car` is located at its center and contains the features `Car`, `EngineType`, `Gasoline`, and `Electric`. The features contain instances of their `Feature Revisions`. Moreover, the `Unified System` contains one `System Revision` that enables `Feature Revision 1` of features `Car`, `Gasoline`, and `Electric`. Note that the `System Revision` also enables the abstract feature `EngineType`. While these three revisions result in a valid state of the `Car` system, other `Feature Revision` combinations may lead to inconsistencies. Additionally, the `Unified System` comprises two `Constraints`. `Constraint` *c1* contains the `Constraint Expression` *Car*, specifying that the *Car* feature must be always present in a `Product`, since it is the root feature. The second `Constraint` *c2* contains the `Constraint Expression` $Car \Leftrightarrow EngineType$. It expresses that both features `Car` and `EngineType` depend on each other and always appear together in a `Product`. Thus, `EngineType` is a mandatory feature. Instead of expressing `Constraint Expressions` as expression trees, they are shown in a simplified way as formulas placed on the respective references. Finally, two `Fragments` represent each a line of code of the `Car` system (according to the respective *value* attribute). Both `Fragments` are related to the first revision of the `Car` feature via the `Mapping Expression` *Car*.1. The `Configuration` *Customer1* yields a `Product` comprising both `Fragments`. Therefore, it selects `Feature Revisions` *Car*.1, *Gasoline*.1 and `EngineType` (represented by the "+" symbol), and deselects the feature `Electric` (represented by the "-" symbol).

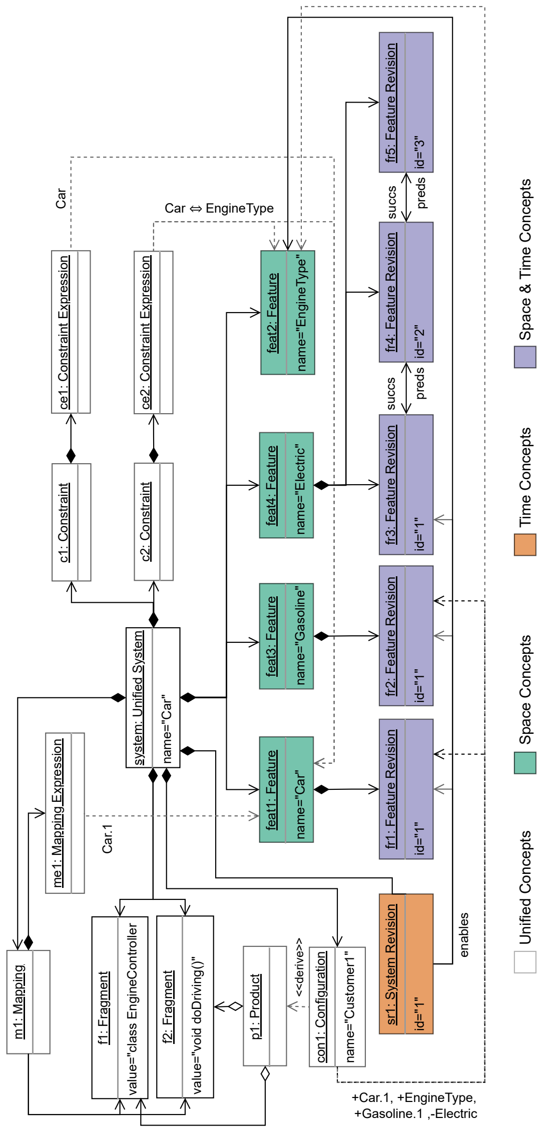


Figure 11.6.: Object diagram of tool T_C applied to the `Car` example [5, Fig. 13].

11.6. Formal Concept Analysis

To further analyze the commonalities and differences of the studied tools and the unified conceptual model, I performed a *formal concept analysis (FCA)* [79, 80]. FCA is an applied branch of lattice theory. It derives a hierarchy of concepts and their attributes by means of a *concept lattice*. For conducting the FCA, the same data (i.e., mappings) was used as for computing the metrics for unification. The FCA offers a comprehensible overview of the relationships among the tools as well as their relationships to the unified conceptual model.

Figure 11.7 depicts the concept lattice between objects (represented by the studied tools) and attributes (represented by concepts of the unified conceptual model). Every node associates a set of tools with a set of concepts that could be mapped to constructs of these tools. A blue filled upper semicircle indicates that there are one or more concepts attached to the node. A black filled lower semicircle of a node indicates that there are one or more tools attached to it. The color scheme of concepts of the unified conceptual model is reused to highlight the edges of the FCA. The concept lattice is inspected from top to bottom. The color of an edge stems from the concepts that are attached to the node it leads to, i.e., if an edge leads to a node with a concept attached for variability in time, such as `Feature`, the edge is colored green, respectively. In case green and orange edges are input to a node, the leafing edge is colored purple. Consequently, purple edges remain of the same color during their path from top to bottom. Inspecting the FCA reveals that nodes and edges top left are related to variability in time. Nodes and edges in the center and top right relate to variability in space. Those on the bottom and right side relate to both variability dimensions. Consequently, the color of edges remain separate until they eventually merge when approaching the unified conceptual model.

The top node depicts common concepts in all tools. The bottom node depicts the unified conceptual model itself. The FCA offers two insights by inspecting the concept lattice: On the one hand, the extent to which tools are related to the conceptual model based on their concepts. On the other hand, the extent to which tools are related to each other. The tools are grouped according to their supported concepts for variability in space, time, or both. For instance, Git and SVN use `System Revisions`, while SiPL, pure::variants and FeatureIDE employ `Constraints` and `Features`. Solely ECCO copes with variability in both dimensions, but does not employ `Constraints` and therefore no variability

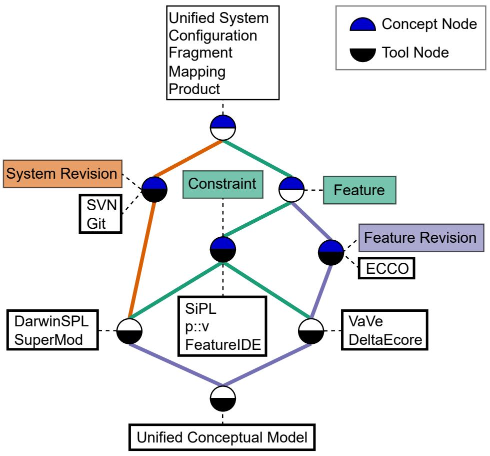


Figure 11.7.: FCA of tools based on unified conceptual model concepts [5, Fig. 7].

model. While DarwinSPL, SuperMod, VaVe, and DeltaEcore are closest to the unified conceptual model, there is no tool that involves all of its concepts.

Figure 11.8 shows the concept lattice between the tools and, in addition to the concepts, also the relations of the unified conceptual model. Concept labels are not depicted to reduce the visual overhead. This visualization allows to further differentiate the tools that employ the same concepts and support the comparison between tools and to the conceptual model. The top node represents six relations common to all tools. Edges on the left represent relations between concepts for variability in time and between unified concepts, while edges on the right represent relations between concepts for variability in space. Again, when getting closer to the conceptual model, edges merge as they represent relations related to both variability dimensions. Interestingly, only the two tools DarwinSPL and DeltaEcore are directly adjacent to the unified conceptual model. Moreover, tools that employ `System Revisions`, `Features`, and `Constraints`, i.e., DarwinSPL and SuperMod, also let `System Revisions` enable `Constraints` and `Feature Options`. Consequently, *enables*-relations are not employed in tools that deal with just one of the variability dimensions.

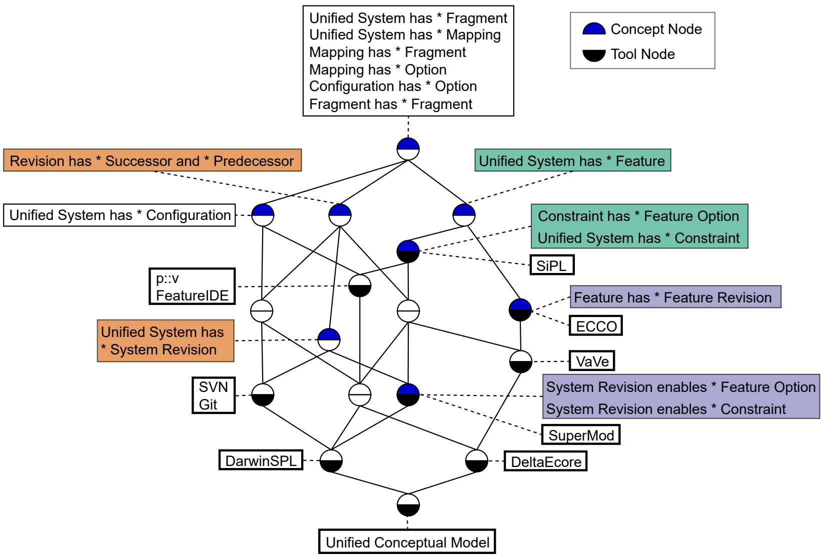

Unified System has * Fragment
Unified System has * Mapping
Mapping has * Fragment
Mapping has * Option
Configuration has * Option
Fragment has * Fragment
Concept Node
Tool Node
Revision has * Successor and * Predecessor
Unified System has * Feature
Unified System has * Configuration
Constraint has * Feature Option
Unified System has * Constraint
SiPL
p::v
FeatureIDE
Feature has * Feature Revision
ECCO
Unified System has * System Revision
VaVe
SVN
Git
System Revision enables * Feature Option
System Revision enables * Constraint
SuperMod
DarwinSPL
DeltaEcore
Unified Conceptual Model

Figure 11.8.: FCA of tools based on unified conceptual model concepts and relations [5, Fig. 8].

11.7. Discussion

From the qualitative and quantitative analyses as well as the illustrating applications, insights could be obtained to answer the questions and discuss the evaluation results.

Q 1.1: *To what extent is the unified conceptual model of appropriate granularity?*

Laconicity and lucidity indicate whether granularity is appropriate. The two tools Git and SVN represent both model concepts `System Revision` and `Configuration` by the construct `Configuration`, i.e., the `System Revision` is equivalent to the `Configuration`. Thus, the laconicity values indicate that both model concepts are unnecessarily fine-grained with respect to these tools and could be merged to increase the laconicity values for both tools. Nonetheless, for any tool that deals with variability in space, a `Configuration` is a set of `Features`. Merging both concepts would decrease the overall laconicity. The lucidity values of the six tools Git, SVN, DeltaEcore, SiPL, DarwinSPL, and VaVe indicate that the concept Fragment is too coarse-grained. The low values stem from different levels of abstraction: delta-oriented tools (i.e., DeltaEcore, SiPL, DarwinSPL, and VaVe) refine a Fragment into a Core Model and Delta Modules. In Git, a Fragment is represented by Blob and Tree Object, and in SVN by File Node and Directory Node. These cases lead to lower lucidity and could be split up. Since the unified conceptual model is intended to be tool-agnostic (which, otherwise, would cause lower laconicity), a reduction in lucidity is justified.

To summarize, the results provide evidence that the granularity of the unified conceptual model is appropriate. Concepts should neither be merged nor could be split up without lowering the abstraction to an undesired level.

Q 1.2: *To what extent is the unified conceptual model of appropriate coverage?*

Completeness and soundness indicate whether coverage is appropriate. The relation Remote of the two tools Git and ECCO is not represented by any relation of the conceptual model. Consequently, this lowers the completeness values. Moreover, the soundness values are rather low *per* tool. Since the unified conceptual model is intended to describe *all* concepts and relations of the elicited tools, those supporting only one variability dimension, such as SVN, Git or FeatureIDE (which are located on the upper half of the FCA in Figure 11.8 and, thus, further away from the unified conceptual model) lead to lower soundness. However, there are no unused concepts or relations in the conceptual model, which is evidenced by the aggregated values in Table 11.5. Consequently, every concept in the unified conceptual model is required by at least one of the studied tools.

To summarize, the results provide evidence that the coverage of the unified conceptual model is almost ideal. It neither employs unused concepts or relations, nor misses concepts. Solely, the unified conceptual model misses support for distributed development. Consequently, adding the relation Unified System refers to * Unified System is the only remaining change that would lead to an improvement of the model (i.e., completeness 100%). Figure 11.9 shows the final conceptual model with the missing added remotes relation.

Q 1.3: *To what extent is the unified conceptual model applicable?*

The illustrating examples presented in Section 11.5 demonstrate the applicability of the unified conceptual model. Based on several degrees of freedom, such as refining the Constraint to express feature modeling, or the Fragment depending on the employed variability mechanism, the unified conceptual model can be refined to apply it in practice. The refinement process was exemplary performed by deriving two tools. While the first tool T_T employs `Feature Revisions` and a feature model, the second tool T_C employs `System Revisions` and `Feature Revisions` simultaneously. Additionally, metrics for unification were demonstratively computed for both tools.

To summarize, the results provide evidence that the conceptual model can be applied to define new tools and to compare them against the unified conceptual model as well as against other tools based on the metrics for unification.

11.8. Threats to Validity

This section describes threats to the validity and how they were mitigated.

Construct Validity. The mapping of tool constructs to model concepts was performed on the conceptual level. Consequently, it was not always clear whether a tool construct constituted the conceptual level or the implementation level. For instance, the concept `Constraint` could be implemented in a tool by specific constructs to support feature modeling, e.g., an optional feature or alternative feature group. In such cases, the representative parent constructs was selected for the mapping, such as the `Constraint`. Interestingly, the tool experts provided answers usually on the same level of abstraction which reassured the mapping results. In most cases, there was no necessity to adjust the level of abstraction, and the answers were considered as literally as possible.

Internal Validity. Some tool experts were involved in the construction process of the unified conceptual model. This could have led to bias towards their tools, which is potential threat to internal validity. Involving further researchers and practitioners into the construction process, as even recommended [1], mitigated this threat.

External Validity. Whether tools can be mapped to the unified conceptual model is matter of external validity. It can be argued that the set of elicited tools is representative as it covers a broad body of existing contemporary tools from both, SPLE and SCM. Moreover, the selected tools are diverse by employing either or both variability dimensions by different means, i.e., via `System Revisions` or `Feature Revisions`). Consequently, bias and local optimizations towards particular tools were mitigated.

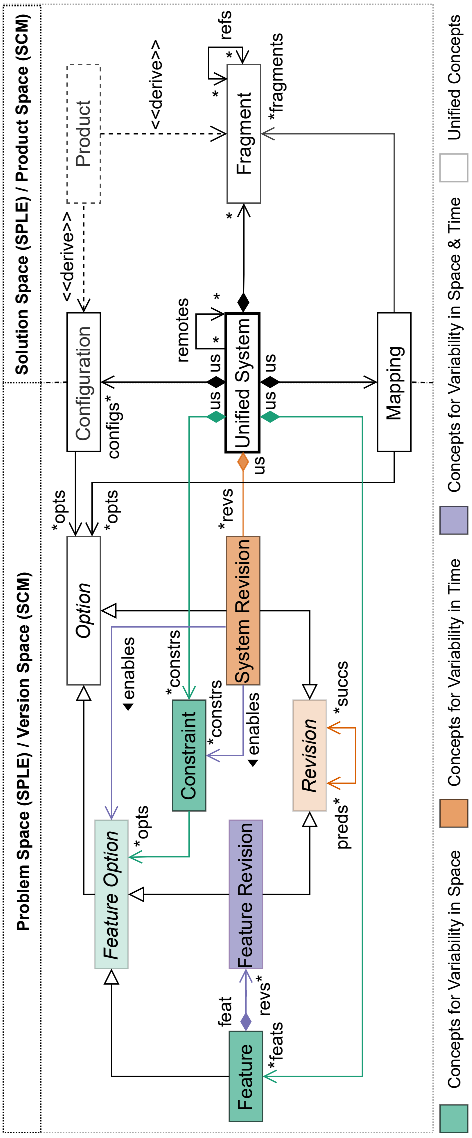


Figure 11.9.: Final unified conceptual model.

Conclusion Validity. Occasionally, the answers of tool experts left space for interpretation, or were incomplete, or posed questions. To mitigate this threat, I repeatedly conferred with the tool experts until consensus. To improve the conclusion validity, all data is published in an open-access repository, so other researchers are able to check the results.

11.9. Limitations and Future Work

Design decisions and evaluation results indicate limitations of the proposed unified conceptual model. In the following, the limitations as well as future work is discussed.

During the construction of the unified conceptual model, several design decisions were made that may be debatable. Versioning is applied to `Options` only but not to `Constraints`, `Configurations` or `Mappings`. In the selected tools, these concepts are not versioned either. Furthermore, it can be argued that, in contrast to `Options`, these concepts do not have their own identity but instead are comprised of concepts with identity (e.g., `Configurations` are comprised of `Options`, and `Constraints` are comprised of `Feature Options`).

Moreover, there may be other concepts or relations required by researchers or practitioners in the future not covered by the unified conceptual model yet. While there is no dedicated extension mechanism for the model, it relies on common object-oriented mechanisms for refinement. One idea for an extension would be to let `Constraints` be formulated over `System Revisions`. In the current state of the unified conceptual model, `Constraints` can only be formulated over `Feature Options` and not over `Options` in general. Nonetheless, `Constraints` are *enabled* by `System Revisions`. While this relation is not as powerful as an arbitrary expression in terms of expressiveness, it can be argued that practically relevant cases can be covered. For example, since different `System Revisions` (i.e., points in time) implicitly exclude each other, `Constraints` where two `System Revisions` exclude or require each other are either always or never satisfied, respectively. If this or any similar modification is still required in the future, the relevant model concepts can be refined accordingly (e.g., by creating a sub-class of `Constraint` that also refers to `System Revisions`).

For future work, it would be interesting to apply the conceptual model to a set of real-world case studies from disciplines other than SPLE and SCM to identify further limitations or shortcomings. This would also allow to investigate whether concepts or relations between them are missing to provide benefits beyond the current state of the art in SPLE and SCM.

11.10. Summary

This chapter presented an evaluation of the appropriateness and applicability of the unified conceptual model. The evaluation encompassed a qualitative analysis that involved a mapping between concepts and relations of the unified conceptual model to constructs and relations of selected tools based on an expert survey. Subsequently, a quantitative analysis comprised an application of the metrics for unification to quantify the appropriateness of granularity and coverage of the unified conceptual model with respect to the studied tools. The evaluation results showed that concepts should neither be merged (i.e., generalized) nor split up (i.e., made more specific). Moreover, the unified conceptual model neither misses concepts nor employs unused concepts or relations, it did solely not support a relation related to distributed development. Furthermore, the unified conceptual model can serve as base to derive novel tools. Therefore, degrees of freedom when refining the unified conceptual model were described and its applicability based on two exemplary tools was demonstrated. Additionally, for both tools, metrics for unification were computed to demonstrate their application. The unified conceptual model provides a foundation for comparing and communicating current research as well as for designing novel tools for managing variability in space and time.

12. Evaluation of Unified Operations

Chapter 6 presented the unified operations to support the evolution of a variable system that copes with variability in space and time (C2). This chapter presents their evaluation following the GQM method [27].

Section 12.1 introduces the goals and questions of the evaluation. Section 12.2 presents the specialized evaluation process of the unified operations. Section 12.3 encompasses the first part of the evaluation which comprises a quantitative analysis based on metrics to quantify the unification of operations with respect to the analyzed tools. The second part of the evaluation demonstrates an application of the unified operations based on evolution scenarios from the literature in Section 12.4. Section 12.5 offers answers to the posed questions while threats to validity are considered in Section 12.6. Section 12.7 comprises a discussion of the limitations and future work of the unified operations. A summary of the main insights in Section 12.8 closes the chapter.

12.1. Goals and Questions

Since the individual tools (used for constructing the unified operations) have already either been published at peer-reviewed scientific venues, or are widely adopted and successful in practice (e.g., Git and SVN), properties, such as correctness or scalability, of the unified operations are not validated. Instead, the first goal of the evaluation is to analyze whether the unified operations appropriately unify operations from the individual tools without losing any functionality while adding additional semantics for coping with variability in space and time simultaneously where necessary. The second goal of the evaluation is to demonstrate the applicability of the unified operations. Therefore, the following three sub-goals are defined regarding granularity, coverage and applicability that shall be met by the unified operations:

***Granularity*:** The granularity of the unified operations shall be appropriate, that is, each operation shall have one responsibility (i.e., deal with exactly one concern) which it covers fully and shall not address any other.

***Coverage*:** The unified operations shall cover all functionality of the selected tools in both variability dimensions.

***Applicability*:** The unified operations shall be applicable to common variability-related evolution scenarios.

Based on these goals, the following questions are asked:

Q 2.1 Are the unified operations of appropriate granularity?

Q 2.2 Are the unified operations of appropriate coverage?

Q 2.3 To what extent are the unified operations applicable to support different development paradigms (i.e., platform-oriented or product-oriented development)

Q 2.4 To what extent are the unified operations applicable to support different edit modalities (i.e., direct or view-based editing)?

Answering these questions allows for assessing whether the unified operations meet the defined goals.

12.2. Specialized Evaluation Process

Figure 12.1 shows the specialized evaluation process of the unified operations. It is consecutive to the unification process shown in Figure 6.1 and based on the general evaluation process described in Section 10.1. Step ⑤ represents the quantitative analysis. The use case mappings (Step ② in Figure 6.1) are input to the metric computation in Step ⑤. Finally, Step ⑥ encompasses a demonstration of the applicability of the unified operations based on diverse variability scenarios from the literature.

Quantitative analysis: For the quantitative analysis, I applied the metrics for unification (see Section 10.2) to the unified operations with respect to the elicited tools. The metrics *laconicity* and *lucidity* quantify the granularity of

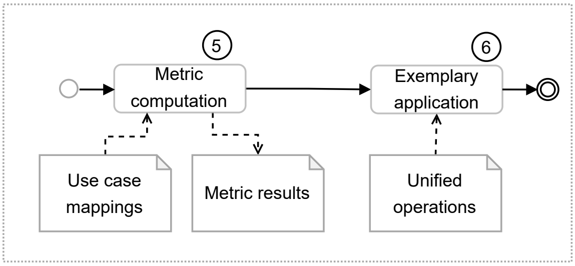


Figure 12.1.: Evaluation process of the unified operations.

the unified operations. The metrics *completeness* and *soundness* quantify the coverage of the unified operations.

Exemplary Application: To demonstrate the applicability of the unified operations, I identified variability scenarios from the literature, and analyzed how these scenarios can be addressed by the unified operations.

12.3. Quantitative Analysis

In the following, Step ⑤ shown in Figure 12.1 is described. It encompasses an application of the metrics for unification (see Section 10.2).

12.3.1. Metrics

The metrics *laconicity* (see Definition 10.1) and *lucidity* (see Definition 10.2) quantify the granularity of unified operations, that is, whether each operation addresses exactly one responsibility (Q 2.1). The metrics *completeness* (see Definition 10.3) and *soundness* (see Definition 10.4) quantify their coverage, that is, whether there is any unused or missing functionality (Q 2.2).

The unification U is the set of unified operations UO. A tool $T \in \mathcal{T}$ is a set of *tool operations* $t \in T$. The *mappings* of unified operations in UO onto tool operations in T are displayed in Table 6.2 and Table 6.4.

The set of unified operations UO comprises the 21 direct editing operations UO_{Direct} as well as the seven view-based operations:

$$UO = UO_{Direct} \cup \{eD, iD, eP, iP, iC, eUS, iUS\}$$

In the following, an example for each metric is shown by applying it to the tool SuperMod. Note that this tool employs view-based operations that are used in either platform-oriented development or involved in both platform and product-oriented development. SuperMod does not employ direct editing operations (that allow the user to directly edit the `Unified System`). SuperMod implements two operations:

$$T_{\text{SuperMod}} = \{\text{checkout}, \text{commit}\}$$

Laconicity According to the operation mapping in Table 6.4, four unified operations are implemented by two operations in SuperMod: *eD* (that is used in platform-oriented development) and *eP* (that is used in both development paradigms) via *checkout*, and *iD* and *iC* (that are both used in platform-oriented development) via *commit*. This is due to the fact that SuperMod uses both its operations as a two-step process: First, to configure the domain for one point in time (i.e., via a `System Revision`) followed by a configuration of the product (i.e., via `Features`) based on the externalized domain. Analogously, the integration of changes into the `Repository` in SuperMod comprises both the internalization of the updated domain as well as of the changed product. The unified operations that do not map to any operation in SuperMod (i.e., *iP*, *eUS* and *iUS*) do not affect the metric. The laconicity for SuperMod is therefore even zero:

$$\text{laconicity}_{\text{SuperMod}}(UO, T_{\text{SuperMod}}) = \frac{0+0}{2} = \frac{0}{2} = 0.0$$

Lucidity All unified operations (view-based or direct editing operations used in product or platform-oriented development), according to the operation mappings in Table 6.3 and Table 6.4, are either implemented by at most one tool operation or by no tool operation in SuperMod. The lucidity of the unified operations with respect to SuperMod is thus ideal:

$$\text{lucidity}_{\text{SuperMod}}(UO, T_{\text{SuperMod}}) = \frac{21+1+1+1+1+1+1+1}{28} = \frac{28}{28} = 1.0$$

Completeness In the example of SuperMod, according to the operation mapping in Table 6.4, there are no operations in SuperMod that do not map to any unified operation. Both its two operations can be mapped to at least one unified operation. The completeness for SuperMod is therefore ideal:

$$\text{completeness}_{\text{SuperMod}}(UO, T_{\text{SuperMod}}) = \frac{1+1}{2} = \frac{2}{2} = 1.0$$

Soudness Regarding SuperMod, out of the 28 unified operations, four are implemented by at least one operation in SuperMod, according to the concept mappings in Table 6.3 and Table 6.4. This is due to SuperMod not employing direct editing operations at all (out of which there are 21 operations) along with operations used only for product-oriented development (i.e., *iP*). Thus, the soundness of the unified operations with respect to only SuperMod is quite low:

$$\text{soundness}_{\text{SuperMod}}(M, T_{\text{SuperMod}}) = \frac{0+1+1+1+0+1+0+0}{28} = \frac{4}{28} = 0.14$$

12.3.2. Results

Table 12.1 shows the values for the four metrics (columns) per tool (rows), separated by the development paradigm of an operation, i.e., platform-oriented, product-oriented, or both as well as in total. In case of lucidity and soundness, each row shows the percentage and the absolute number of unified operations that satisfy the condition for each metric. In case of laconicity and completeness, each row shows the percentage and the absolute number of individual tool operations that satisfy the condition for each metric. A horizontal line indicates that a tool does not employ operations of either or both development paradigms. For instance, the tool ECCO supports product-oriented development (i.e., the *iP* operation), no operation for platform-oriented development, but operations for both development paradigms (i.e., *eP*, *eUS*, and *iUS*). The metric values for lucidity and completeness are at 100 % for all analyzed tools, indicating that the unified operations are not too coarse-grained and do not miss any concern of the individual tools. For almost every tool, the metric values for laconicity are at 100%, indicating that the unified operations are not too fine-grained. An exception is the tool SuperMod, which is the only tool whose metric values for laconicity are at 0%. On the one hand, this is due to the fact that both *eD* and *eP* are part of SuperMod's *checkout* operation (note that this operation is counted as one that supports both the platform-oriented and the product-oriented development paradigm). On the other hand, both *iD* and *iC* art part of SuperMod's *commit* operation. Consequently, both tool operations are not laconic as they implement more than one unified operation. Another tool whose laconicity value is lower at 82% is VTS. Both eP and eUS art part of VTS's *get* operation, while both iC and iUS art part of VTS's *put* operation. In contrast to SuperMod though, VTS employs direct editing operations, such as *add*, *update* and *delete* of the concepts `Mapping, Fragment and Feature`. Consequently, while its view-based operations are not laconic, its direct editing operations are. Finally, the metric values for soundness of the unified operations vary between 7% and 64%. This is due to the fact the three direct editing operations (*add/update/delete* `System Revision`) are not implemented by at least one tool.

Table 12.2 shows the aggregated results over all tools. The four metrics are shown as columns for the view-based and direct edit modality and for the development paradigm of an operation, i.e., platform-oriented, product-oriented, or both. The metric values are at or close to 100%. While the lower values for laconicity are due to the described view-based operations of SuperMod and VTS, the lower values for soundness are due to the direct editing operations for `System Revisions` that none of the tools employ.

12.4. Exemplary Application

This section describes Step ⑥ shown in Figure 12.1 to demonstrate the applicability of the unified operations based on variability scenarios.

12.4.1. Variability Scenarios

From the literature, eight variability scenarios (i.e., self-contained activities of a developer with a specific intent that are related to the management of a variable system) were identified. While the collected scenarios may not be exhaustive, they shall be diverse enough to demonstrate the applicability of the unified operations. Therefore, the following requirements are defined:

1. Scenarios shall holistically address variability management and therefore involve both the problem space and the solution space.
2. Scenarios shall be diverse and therefore involve product line evolution, clone-and-own development and distributed development.

Table 12.1.: Metric results for each tool individually.

	for	laconicity	lucidity	completeness	soundness
Feature IDE	*Platform*	100% (15/15)	100% (24/24)	100% (15/15)	63% (15/24)
	Product	– (0/0)	100% (1/1)	– (0/0)	0% (0/1)
	Both	100% (2/2)	100% (3/3)	100% (2/2)	67% (2/3)
	Total	100% (17/17)	100% (28/28)	100% (17/17)	61% (17/28)
VTS	*Platform*	100% (9/9)	100% (24/24)	100% (9/9)	42% (10/24)
	Product	– (0/0)	100% (1/1)	– (0/0)	0% (0/1)
	Both	0% (0/2)	100% (3/3)	100% (2/2)	100% (3/3)
	Total	82% (9/11)	100% (28/28)	100% (11/11)	46% (13/28)
SiPL	*Platform*	100% (15/15)	100% (24/24)	100% (15/15)	63% (15/24)
	Product	– (0/0)	100% (1/1)	– (0/0)	0% (0/1)
	Both	100% (1/1)	100% (3/3)	100% (1/1)	33% (1/3)
	Total	100% (16/16)	100% (28/28)	100% (16/16)	57% (16/28)
SVN	*Platform*	– (0/0)	100% (24/24)	– (0/0)	0% (0/24)
	Product	100% (1/1)	100% (1/1)	100% (1/1)	100% (1/1)
	Both	100% (1/1)	100% (3/3)	100% (1/1)	33% (1/3)
	Total	100% (2/2)	100% (28/28)	100% (2/2)	7% (2/28)
Git	*Platform*	– (0/0)	100% (24/24)	– (0/0)	0% (0/24)
	Product	100% (1/1)	100% (1/1)	100% (1/1)	100% (1/1)
	Both	100% (3/3)	100% (3/3)	100% (3/3)	100% (3/3)
	Total	100% (4/4)	100% (28/28)	100% (4/4)	14% (4/28)
ECCO	*Platform*	– (0/0)	100% (24/24)	– (0/0)	0% (0/24)
	Product	100% (1/1)	100% (1/1)	100% (1/1)	100% (1/1)
	Both	100% (3/3)	100% (3/3)	100% (3/3)	100% (3/3)
	Total	100% (4/4)	100% (28/28)	100% (4/4)	14% (4/28)
Super Mod	*Platform*	0% (0/1)	100% (24/24)	100% (1/1)	13% (3/24)
	Product	– (0/0)	100% (1/1)	– (0/0)	0% (0/1)
	Both	0% (0/1)	100% (3/3)	100% (1/1)	33% (1/3)
	Total	0% (0/2)	100% (28/28)	100% (2/2)	14% (4/28)
Delta Ecore	*Platform*	100% (18/18)	100% (24/24)	100% (18/18)	75% (18/24)
	Product	– (0/0)	100% (1/1)	– (0/0)	0% (0/1)
	Both	100% (1/1)	100% (3/3)	100% (1/1)	0% (0/3)
	Total	100% (19/19)	100% (28/28)	100% (19/19)	64% (18/28)
Darwin SPL	*Platform*	100% (12/12)	100% (24/24)	100% (12/12)	50% (12/24)
	Product	– (0/0)	100% (1/1)	– (0/0)	0% (0/1)
	Both	100% (1/1)	100% (3/3)	100% (1/1)	33% (1/3)
	Total	100% (13/13)	100% (28/28)	100% (13/13)	46% (13/28)
VaVe	*Platform*	100% (1/1)	100% (24/24)	100% (1/1)	4% (1/24)
	Product	– (0/0)	100% (1/1)	– (0/0)	0% (0/1)
	Both	100% (1/1)	100% (3/3)	100% (1/1)	33% (1/3)
	Total	100% (2/2)	100% (28/28)	100% (2/2)	7% (2/28)

Table 12.2.: Metric results over all tools.

Kind	for	laconicity	lucidity	completeness	soundness
View-Based	*Platform*	67% (2/3)	100% (3/3)	100% (3/3)	100% (3/3)
	Product	100% (3/3)	100% (1/1)	100% (3/3)	100% (1/1)
	Both	81% (13/16)	100% (3/3)	100% (16/16)	100% (3/3)
	Total	82% (18/22)	100% (7/7)	100% (22/22)	100% (7/7)
Direct	*Platform*	100% (68/68)	100% (21/21)	100% (68/68)	86% (18/21)
	Product	– (0/0)	– (0/0)	– (0/0)	– (0/0)
	Both	– (0/0)	– (0/0)	– (0/0)	– (0/0)
	Total	100% (68/68)	100% (21/21)	100% (68/68)	86% (18/21)
All	*Platform*	99% (70/71)	100% (24/24)	100% (71/71)	88% (21/24)
	Product	100% (3/3)	100% (1/1)	100% (3/3)	100% (1/1)
	Both	81% (13/16)	100% (3/3)	100% (16/16)	100% (3/3)
	Total	96% (86/90)	100% (28/28)	100% (90/90)	89% (25/28)

Table 12.3.: Scenarios and operation sequences.

	Scenario	Platform-Oriented Sequence	Product-Oriented Sequence
S1[1]	Transfer entire system	*iUS*	*iUS*
S2[1]	Transfer subset of features	*eUS*, *iUS*	*eUS*, *iUS*
S3[1]	Transfer individual product	*eUS*, *iUS*	(*eUS*, *iUS*) | (*eP*, *iP*)
S4	Retrieve supported product	*eP*	*eP*
S5	New combination of existing features	*eD*, *iD*, (*eP*, (*iC*)+)+	*eP*, *iP*
S6	New separation of existing features	*eD*, *iD*, (*eP*, (*iC*)+)+	*eP*, *iP*
S7	Add new feature	*eD*, *iD*, (*eP*, (*iC*)+)+	*eP*, *iP*
S8	Update feature implementation in all products	(*eP*, (*iC*)+)+	(*eP*, *iP*)+

[1] Distributed development scenario. .

Table 12.3 shows the variability scenarios and the platform-oriented or product-oriented operation sequences (noted as regular expressions) for addressing them. The scenarios are lifted to the unified conceptual model and extended with temporal aspects in cases where a scenario considers only spatial variations. For example, scenarios that include updates or deletions shall not actually update or delete objects in the unified system, and instead create new revisions to enable or disable them.

Scenarios S1, S2, and S3 are distributed development scenarios supported in both paradigms. Scenario *S1* [229, 127, 129] describes the transfer and integration of all contents of one unified system into another. Scenario *S2* (e.g., "propagating a feature" [104]) [229, 99, 129] describes the transfer and integration of only a subset of all features (and the corresponding subset of mappings and fragments). Scenario *S3* deals with the transfer and integration of a product [161] (i.e., the relevant features, fragments, and mappings) from a remote unified system into the local unified system. Scenario *S4* [12, 129] describes the retrieval of a product from a unified system based on a valid configuration that is already supported (i.e., the product has already been internalized via *iP*, or all required features and feature interactions have been internalized via *iC*). Scenario *S5* (e.g., "asset merging" [183], "merge" [200]) and *S6* (e.g., "asset splitting" [183], "split asset" [168]) deal with the retrieval of a product, where two features or feature revisions are combined or separated that have never been combined or separated in any previously internalized product. Missing or surplus feature interaction fragments may have to be added or deleted. The manually completed product shall then be integrated into the unified system. In Scenario *S7* ("merge" [200, 129]), a new feature (including the relevant fragments, constraints, and mappings) is added to the unified system. Finally, in Scenario *S8* ("variant synchronization" [230, 110]), the intention is to change the implementation of a feature (e.g., to fix a bug), and have it take effect in all products with that feature.

12.4.2. Results

For Scenario S1, only operation iUS is needed. For Scenario S2, eUS is used before iUS, with a partial configuration that deselects all undesired features to derive a unified system containing only the desired subset of features. For Scenario S3, a complete configuration is used to derive a unified system containing a single product via eUS, followed by iUS. Alternatively, the product can be externalized from one unified system via *eP*, and internalized into another via *iP*. Scenario S4 is trivially possible via the operation *eP*, where a configuration is specified that is already supported in the unified system (i.e., a product with that configuration has been internalized before via *iP* or the necessary features have been internalized via *iC*). Scenarios S5, S6, and S7 are addressed using the same sequences of operations. In case of platform-oriented development, the platform is edited via *eD* and *iD* to add new features and delete constraints, such that new feature combinations are allowed. This is followed by a repetition of *eP* and *iC* to add fragments and corresponding mappings to the platform, until the desired product can be derived entirely by the *eP* operation. During product-oriented development, an incomplete product is externalized via *eP* by reusing relevant feature and feature interaction fragments from previously internalized products. The product must be modified by adding fragments that implement the interaction of features (S5), removing surplus fragments (S6), or adding missing fragments (S7). Finally, the finished product is internalized via *iP*. The operation sequences for Scenario *S8* are similar except that the domain (i.e., options and constraints) does not need to be modified in the case of platform-oriented development. For product-oriented development, this is a challenging scenario, since the sequence, albeit the same as before, may need to be repeated many times—in the worst case once per product.

12.5. Discussion

From the quantitative analysis and the exemplary application, insights could be obtained to answer the questions and discuss the evaluation results.

Q 2.1: *Are the unified operations of appropriate granularity?*

Laconicity and lucidity indicate whether granularity is appropriate. While the metric values for lucidity are always at 100% (i.e., a unified operation is implemented by at most one operation of a tool), the laconicity values of view-based unified operations vary between 67% and 81% (i.e., a tool operation implements at most one unified operation). Specifically, the tools VTS and SuperMod provide two operations that each cover two unified operations. For instance, VTS's operation *put* represents both iUS and eUS and SuperMod's operation *checkout* represents both *eD* and *eP*. Considered as a whole, this affects all view-based operations that support the platform-oriented paradigm (i.e., *eD*, *iD*, *iC*) or both paradigms (i.e., *eP*, eUS, iUS). Nonetheless, the affected unified operations were not merged to avoid ambiguities of an operation's concern and to clearly separate their concerns and responsibilities. Besides the direct editing operations (whose laconicity value as at 100%), the only remaining view-based unified operation that is fully laconic is the *iP* operation which is implemented in the tools SVN, Git and ECCO by exactly one operation.

To summarize, the results provide evidence that the unified operations are of appropriate granularity. Operations should neither be merged nor split up.

Q 2.2: *Are the unified operations of appropriate coverage?*

Completeness and soundness indicate whether coverage is appropriate. While the metric values for completeness are always at 100% (i.e., a tool operation is represented by at least one unified operation), the soundness values of direct unified operations are at 86% (i.e., a unified operation is implemented by at least one operation in a tool). This is due to the fact that directly adding, deleting and updating a `System Revision` is not supported by any tool. Not supporting any direct edit operation for the `System Revision` would increase the metric values for soundness. However, the direct editing of `System Revisions` was retained to provide a complete set of direct editing operations that is as comprehensive as possible.

To summarize, the results provide evidence that the unified operations achieve high coverage. While the direct editing of `System Revisions` represents unused operations, there are no missing operations.

Q 2.3: *To what extent are the unified operations applicable to support different development paradigms?*

A comparison of the operation-execution sequences when solving each scenario with operations of either development paradigm offers an answer to this question. While both paradigms support the same scenarios, the number of necessary operations differs in several cases. A major distinguishing factor is whether only a few or many products shall be affected. It is not surprising that, in the latter case, platform-oriented development is more suitable, since SPLE is tailored to that requirement. In the former case, it can be more convenient to use product-oriented development, since developers can focus on individual products, which has always been an advantage of clone-and-own [75]. However, thanks to an increase in automated support for product-oriented development (e.g., automated reuse among cloned variants via feature location [153, 20, 245, 160] or clone synchronization techniques [186, 198]), this line starts to blur [127, 129]. This is also evidenced by the support for intensional versioning [48] in product-oriented development provided by the operations.

In summary, each development paradigm has its merits and should be chosen based on the scenario. While the unified view-based operations fully support each development paradigm individually, it is still not possible to freely alternate between them, which remains an open challenge.

Q 2.4: *To what extent are unified operations applicable to support different edit modalities?*

To answer this question, an analysis is performed whether view-based editing operations can be used to perform the same edits as the direct editing operations shown in Table 6.3, except those editing the time dimension (i.e., revisions). The rationale for this comparison is that direct editing operations cover the entire evolution spectrum, since there is no evolutionary change in any scenario that cannot be covered (albeit with significant manual effort) via direct editing operations. In contrast, view-based editing operations offer a higher degree of automation (such as tracking revisions or modifying multiple mappings at once), and are therefore often more convenient to use, but may be limited in what they can achieve. View-based editing operations were found to support the same edits as direct editing operations with the operation sequence (*eD*,*iD*)?(*eP*,(*iC*)+)+. However, changing mappings with view-based operations is not as straightforward as with direct editing, since the view (i.e., product) only shows fragments but not the mappings. The only way to modify a mapping via view-based operations is by creating a product via *eP*, modifying its fragments, and then internalizing it via *iC*, where new mapping expressions are computed automatically and only for fragments that were either added or deleted from the product. This limits the way in which mapping expressions can be modified. If the desired change of the current mapping expression m to the new mapping expression m' is such that $m' \Rightarrow m$, the mapping expression becomes more restrictive and the fragment will be contained in a subset of the previous products. This can be achieved via *iC*, since the new mapping expression is computed by appending the expression provided by the developer to the old mapping expression via a logical *and*. If the relation between m and m' is such that $m \Rightarrow m'$ (i.e., it becomes less restrictive and the fragment will be contained in additional products) or there is no relation, this cannot be achieved via view-based operations.

In summary, the unified view-based operations cannot fully replace the unified direct editing operations, as the former do not support arbitrary modifications of mappings. However, this limitation of view-based operations is also present in the studied tools and therefore represents an open challenge.

12.6. Threats to Validity

This section describes threats to the validity and how they were mitigated.

Internal Validity. The unification of the operations was performed on the level of abstraction of the unified conceptual model. Consequently, there may be further details in the behavior of tools that may be missed. This threat was mitigated by two means: i) conferring with the tool experts, and ii) verifying that the unified operations maintained the functionality and support the same edits as the individual tools (the metric values for completeness are always at 100%). Another potential threat to internal validity is the involvement of tool experts in the unification process of the operations, which could have introduced bias towards their tools. However, this had the advantage that current, detailed, and reliable information could be elicited instead of only inspecting the respective tool-related publications. Furthermore, this threat was mitigated by involving further researchers and practitioners in the unification process.

External Validity. A diverse and representative set of tools was elicited and analyzed from both the SPLE and SCM research areas. The selected tools are based on different concepts and support different modalities and paradigms. Nonetheless, there may be other tools comprising further operations that cope with variability in space and time. Thus, the set of unified operations may not be fully generalizable. This threat was mitigated by analyzing to what extent the operations can be applied to different variability scenarios reported in the literature.

12.7. Limitations and Future Work

The discussed evaluation results indicate limitations of the unified operations. In the following, the limitations as well as future work are discussed.

The modification of mappings can be performed straightforward with direct editing. However, as the evaluation results showed, unified view-based operations (and, respectively, the analyzed view-based tool operations) provide only limited possibilities to change mappings, since the view (i.e., product) only shows fragments but not the mappings. This is a current limitation of the unified operations. Future work could address this shortcoming by combining both edit modalities to leverage benefits of both. While view-based unified operations provide a high degree of automation, direct editing allows for flexibility where view-based operations don't. However, combining both edit modalities is challenging. The challenge would be to provide an editable view including mappings, since this could require a particular view for every type of artifact. VTS is the only studied tool that supports both edit modalities. Based on its *get* operation, it either returns a product (that can be edited in a view-based manner) or a partial product line (that can be edited directly). In VTS, an editable view that also includes mappings can easily be provided, as fragments are lines of text and mappings are annotations. While existing approaches allow for dynamically switching between editable product and platform views, they are also limited to textual fragments [162, 31]. Note that the direct editing of time concepts is discouraged in this research to prevent the user from accidentally changing the history.

Further future work is the combination of development paradigms, that is envisioned to further support the evolution of a variable system. The ability to arbitrarily alternate between platform-oriented and product-oriented development would make it possible to leverage the benefits of both. However, this remains an open challenge in state of the art, as none of the studied tools allow to switch between the two paradigms. Platform-oriented development would be the paradigm of choice to make modifications to the platform, but it requires high cognitive effort [141, 214]. Product-oriented development reduces complexity as it allows a developer to focus on a single product, but does not produce fine-grained mappings that are necessary for platform-oriented development. Consequently, additional techniques such as feature location [160, 20, 199] are required. Otherwise, it has only limited support for intensional versioning (i.e., the derivation of new products [48]). Thus, allowing a developer to pick the paradigm that is best suited for any given development task can provide a substantial advantage.

12.8. Summary

This chapter presented the evaluation of the appropriateness and applicability of the unified operations. A quantitative analysis was performed to quantify the appropriateness of the unified operations with respect to the analyzed tools. Therefore, metrics for unification were applied (see Section 10.2). Main insights showed that while the unified operations do not miss any functionality of the considered tools, they provide operations that are not employed by any tools (i.e., the direct editing of system revisions). Furthermore, the application of the unified operations was demonstrated based on variability scenarios identified from the literature. While the scenarios were fully supported, the number of necessary operations when solving each scenario differs based on the development paradigm. Furthermore, the modification of mappings is only supported in a limited way via unified view-based operations (and, consequently, in the analyzed tools). Based on the insights, future work as well as open challenges of combining editing modalities and development paradigms are discussed.

13. Evaluation of the Unified Approach

Chapter 8 presented the unified approach to support the consistent evolution of systems that are subject to variability in space and time, and that are composed of heterogeneous artifacts (C5). This chapter presents the evaluation of its functional suitability based on the GQM method [27].

Section 13.1 presents the overall evaluation goal. Section 13.2 introduces two real-world case study systems that are used for evaluation. For every inconsistency type that is detected and resolved by the unified approach, i.e., Inconsistency Type 2 (*feature model to product consistency*, as described in Section 8.6.1), Type 5 (*product to feature model consistency*, as described in Section 8.6.2), and Type 6 (*product consistency*, as described in Section 8.6.3), a tailored evaluation is provided in Section 13.3–Section 13.5. It encompasses an overview of the questions and metrics, the applied evaluation process, and a discussion of the evaluation results. Section 13.6 comprises a brief description of the employed technologies. Section 13.7 involves a discussion of the limitations and future work of the unified approach. A summary of the main insights in Section 13.9 closes the chapter.

13.1. Overall goal

The overall goal of the evaluation of the unified approach is to provide evidence on its functional suitability[1]. In particular, the approach shall fully support the evolution of a variable system (i.e., *functional completeness*), detect and repair inconsistencies correctly (i.e., *functional correctness*) with the desired degree of automation (i.e., *functional appropriateness*).

[1] `https://iso25000.com/index.php/en/iso-25000-standards/iso-25010`

13.2. Case Studies

To evaluate the unified approach holistically and realistically, case studies must meet several requirements. In the following, the selection criteria is introduced and the selected case studies are described.

13.2.1. Selection Criteria

The goal is to evaluate the unified approach by taking into account both variability dimensions, the heterogeneity of implementation artifacts, and the handling of variability-related inconsistencies that were caused or repaired in the solution space (i.e., the heterogeneous implementation). Therefore, case study systems were selected based on the following selection criteria:

1. The case study is variable in space, for example, by managing a set of products and/or features and providing a variability model.
2. The case study is variable in time by having an evolution history, ideally also of the variability model.
3. The case study has an implementation comprised of different types of artifacts.
4. The case study is a real-world system.
5. The case study is publicly available.

ArgoUML-SPL and MobileMedia are two case study systems that satisfy the described selection criteria and that are introduced in the following. Both are implemented in Java and use a preprocessor as variability mechanism. Since the feature model only evolved in MobileMedia and not in ArgoUML-SPL, MobileMedia was used for the evaluation of Type 2 and Type 5. For Type 6, the evaluation was applied to both case studies.

13.2.2. ArgoUML-SPL

ArgoUML-SPL that has been extracted from a snapshot of ArgoUML, a UML modeling tool, in 2009.[2] The ArgoUML-SPL serves as real-world case study in the SPL research community that is widely used [49, 153, 159, 154]. ArgoUML-SPL is implemented in Java, has about 130 KLOC and uses conditional compilation via the Java preprocessor as variability mechanism. Figure 13.1 shows the feature model of the ArgoUML-SPL. It consists of three core features (i.e., `ArgoUML-SPL`, `Diagrams`, `Class`) and seven optional features (e.g., `Activity`, `Cognitive` and `Logging`). Unfortunately, ArgoUML-SPL has not been co-evolved with the original ArgoUML system. Although ArgoUML kept evolving, the ArgoUML-SPL remained in its inital revision. To remedy this, I manually evolved ArgoUML-SPL by retroactively replaying, i.e., comparing and merging, the revision history of ArgoUML on the ArgoUML-SPL. In total, 28 commits of ArgoUML were analyzed, with a total of 32 changed Java files with 619 additions and 407 deletions that were merged into the ArgoUML-SPL. Successive ArgoUML revisions that affected the same feature expression were grouped into a single ArgoUML-SPL revision, which ultimately resulted in nine ArgoUML-SPL revisions. Table 13.1 shows the nine revisions and the respective expressions comprising features or feature interactions that are affected (i.e., changed within a respective Java preprocessor annotation) per revision. In total, four distinct feature expressions were affected: *Core*, *Cognitive*, *Logging*, $Cognitive \wedge Logging$. In revision one, three, six, and eight, only the core features are affected by changes. In revision two, seven, and nine, the `Logging` feature is additionally changed. In the fourth revision, only the `Cognitive` feature is changed, while in the fifth revision, both the `Cognitive` feature along with the feature interaction $Cognitive \wedge Logging$ is affected. For the evaluation, mandatory core features are considered that are part of every product, the two optional cross-cutting features `Cognitive` and `Logging`, and the additional optional `Activity` diagram feature. While the data set contains four additional diagram features, they are omitted from this evaluation, as they were not affected by any of the subsequent revisions. To establish heterogeneous implementation artifacts for this evaluation, UML class diagrams are automatically extracted for every product of the ArgoUML-

[2] `https://github.com/argouml-tigris-org/argouml/commit/43d8b97eebec3e9d366744528465b1b253cbf482`

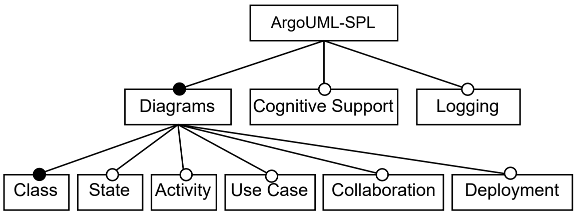


Figure 13.1.: Feature model of the ArgoUML-SPL [49, Fig. 2].

Table 13.1.: Internalization expressions of the ArgoUML-SPL data set.

Revision	Feature Expression(s)
1	*Core*
2	*Core*, *Logging*
3	*Core*
4	*Cognitive*
5	*Cognitive*, $Cognitive \land Logging$
6	*Core*
7	*Core*, *Logging*
8	*Core*
9	*Core*, *Logging*

SPL from its Java source code using functionality provided by Vitruvius (not considering features, the product line, or variability at all).

13.2.3. MobileMedia

The second case study for the evaluation is MobileMedia[3] [71]. MobileMedia is an application for mobile devices to manage media, i.e., photos, music, and videos. MobileMedia is implemented in Java, has approximately 3 KLOC and uses conditional compilation via the Antenna preprocessor[4] as variability mechanism. In contrast to the ArgoUML-SPL, where the feature model does not evolve, the feature model of MobileMedia changes in every revision. Figure 13.2 shows a simplified feature model of MobileMedia at revision 5. It has 7 core features (e.g., the supported media type `Photo`, and the operations to `Label` and `View/Play` media) and four optional features (e.g., `Sort`,

[3] `https://sourceforge.net/projects/mobilemedia/`, last visited on March 10, 2022.
[4] `http://antenna.sourceforge.net/wtkpreprocess.php`, last visited on March 10, 2022.

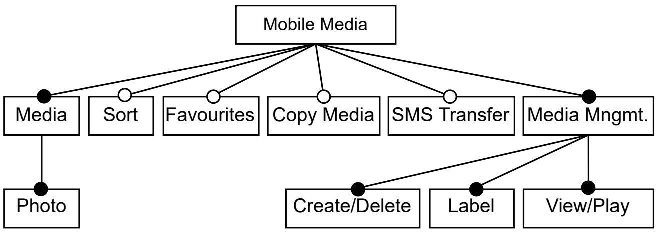


Figure 13.2.: Simplified feature model of MobileMedia at revision 5. Adapted from [71, Fig. 3].

Table 13.2.: Internalization expressions of the MobileMedia data set.

Revision	Feature Expression(s)
1	*Core*
2	*Sort, Label*
3	*Core, Favourite*
4	*Core, Sort, Favourite, Sort* $\wedge$ *Favourite, Copy*
5	*Core, Copy, SMS*

`Favourites`). Table 13.2 shows the first five revisions (i.e., Git commits) starting from an empty product line and the respective expressions comprising features or feature interactions that are affected (i.e., changed within a respective Java preprocessor annotation) per revision. In the first revision, the core features are added to MobileMedia, that are grouped by the expression `Core`. In the second revision, both features `Sort` and `Label` are affected by changes in the implementation. In the third revision, changes affect the `Core` and the `Favourite` feature. The fourth revision comprises changes to most features. Specifically, `Core`, `Sort`, `Favourite` and `Copy` change, as well as the feature interaction *Sort* $\wedge$ *Favourite*. Finally, in the fifth revision, the `Core`, `Copy`, and `SMS` features are affected by changes. Analogously to ArgoUML-SPL, heterogeneous implementation artifacts are created by automatically extracting UML class diagrams for every product of MobileMedia from its Java source code.

13.3. Feature Model to Product Consistency

This section presents the evaluation for the Inconsistency Type 2 (*feature model to product consistency*, as described in Section 8.6.1).

13.3.1. Questions and Metrics

Following the overall goal of functional suitability, four questions are asked. In case of adding new features or removing constraints between features, new configurations and thus new products become viable. The first question concerns the MobileMedia data set:

Q 3.1.1 How many products became valid after each feature model change?

This question can be answered based on metric **M3.1.1**:

Let C_r be the set of valid configurations in the feature model at revision r.

Let $C_r^+ = C_r \setminus C_{r-1}$ be the set of newly valid configurations at revision r compared to the previous revision $r-1$.

M3.1.1 = $|C_r^+|$ is the number of newly valid configurations after the feature model evolution.

If the implementation of the new feature(s) or feature interaction(s) is missing, this leads to a problem space–solution space inconsistency (see Section 2.7). Building upon Q 3.1.1, the second question asks:

Q 3.1.2 Which fraction of the newly valid products is inconsistent?

Moreover, the beneficial characteristics of the view-based edit modality of the unified approach is considered:

Q 3.1.3 What is the proportion of the manually repaired products to restore consistency in all other affected products?

Both questions can be answered based on **M3.1.2** and **M3.1.3**:

Let $\hat{P}_r^+$ be the set of newly valid products at revision r whose implementation does not match the implementation of the respective ground truth products.

M3.1.2 = $\frac{|\hat{P}_r^+|}{|P_r^+|}$ is the fraction of newly valid products that are inconsistent.

Let $\bar{P}_r^+$ be the set of newly valid products that needed to be manually repaired and internalized via *iC* before all newly valid products P_r^+ became consistent with the ground truth.

M3.1.3 = $\frac{|\bar{P}_r^+|}{|P_r^+|}$ is the fraction of newly valid products that needed to be repaired manually to restore consistency in all other affected products.

Finally, to check whether the unified approach provides the correct hints, the causing features or feature interactions of the inconsistencies are considered:

Q 3.1.4 How many of the causing features or feature interactions were hinted at by the unified approach?

This question can be answered based on metric **M3.1.4**:

Let H_r be the set of hints computed by the unified approach during *iD* of revision r.

Let $H_{r,gt}$ be the set of correct hints based on the ground truth, i.e., the set of features and feature interactions causing the inconsistencies at revision r.

M3.1.4 = $|H_r \cap H_{r,gt}|$ is the number of causing features and feature interactions that were hinted at by the unified approach.

13.3.2. Evaluation Process

Figure 13.3 shows a UML activity diagram of the applied evaluation process. Unified operations are highlighted in grey. For every revision of MobileMedia, the domain is externalized via *eD* which provides the feature model at a particular revision. Based on the ground truth feature model at the succeeding revision, the provided feature model is evolved and internalized back via *iD* into the unified system. Upon every internalization operation, a configuration space analysis of the evolved feature model is performed. In case of added features or the removal of constraints between features, the configurable space increases. If this is the case, a hint is provided for every new feature and feature combination whose implementation is missing and may thus lead to an inconsistency (see Section 2.7). While not all inconsistency-causing

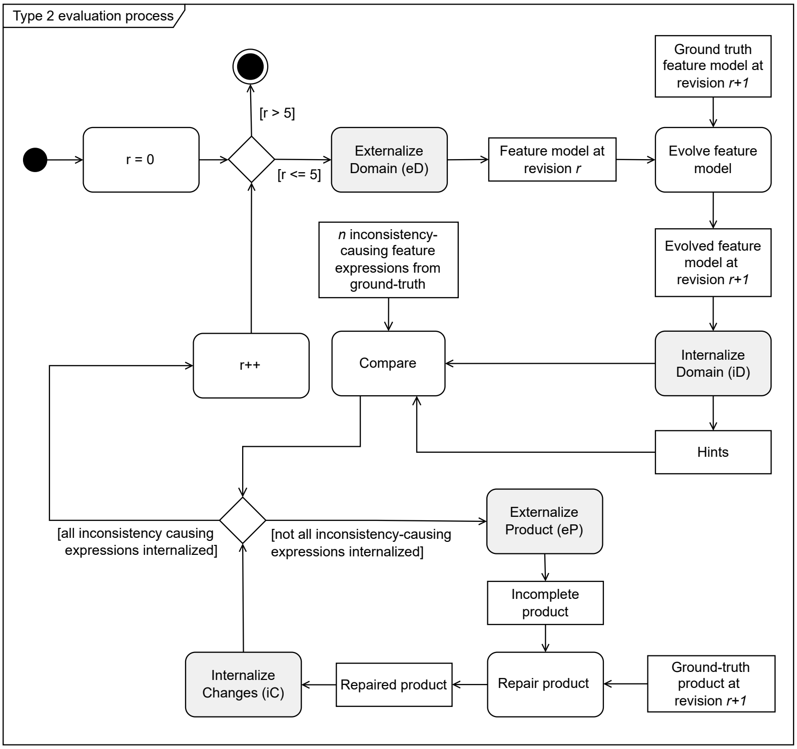

Type 2 evaluation process
r = 0
[r > 5]
[r <= 5]
Externalize Domain (eD)
Feature model at revision r
Ground truth feature model at revision r+1
Evolve feature model
Evolved feature model at revision r+1
n inconsistency-causing feature expressions from ground-truth
r++
Compare
Internalize Domain (iD)
Hints
[all inconsistency causing expressions internalized]
[not all inconsistency-causing expressions internalized]
Externalize Product (eP)
Incomplete product
Internalize Changes (iC)
Repaired product
Repair product
Ground-truth product at revision r+1

Figure 13.3.: UML activity diagram of the evaluation process for feature model to product consistency (Inconsistency Type 2).

expressions have been internalized, a product is externalized via *eP* and manually repaired based on the ground truth product at the succeeding revision r+1 and internalized into the system via *iP*, which addresses the corresponding hint. The complete process is repeated for each revision of MobileMedia.

13.3.3. Results

Table 13.3 shows the results of the evaluation for Inconsistency Type 2. Each cell shows the results for one particular metric (**M.3.1.1**–**M3.1.4**) (columns) per revision (rows).

Table 13.3.: MobileMedia evaluation results for Type 2.

Rev. \ Metrics	M3.1.1	M3.1.2	M3.1.3	M3.1.4
1	+1 = 1 (+Root)	1/1 (100%)	1/1 (100%)	1/1
2	+2 − 1 = 2 (+Label, +Sort)	2/2 (100%)	2/2 (100%)	5/2
3	+2 = 4 (+Favourite)	2/2 (100%)	1/2 (50%)	4/1*
4	+4 = 8 (+Copy)	4/4 (100%)	1/4 (25%)	5/1
5	+8 = 16 (+SMS)	8/8 (100%)	1/8 (12,5%)	6/1

* `Favourite` ∧ `Sort` combination was hinted at but internalized later in revision 4.

Metric **M3.1.1** indicates that the number of newly valid configurations increases with every revision. In the first revision, the feature `Root` is added to the feature model which leads to one product only consisting of the `Root` feature. In the second revision, the mandatory `Label` feature and optional `Sort` feature are added to the feature model, leading to two new products (i.e., {Label}, {Label,Sort}) while a configuration with only the `Root` feature is not possible anymore. In the third revision, the optional feature `Favourite` is added to the model, leading to two new products (four in total). In revision four, the optional feature `Copy` is added to the feature model, thus leading to four new products and eight products in total. Finally, in revision five, the optional feature `SMS` is added to the feature model, leading to eigth new products and 16 products in total.

Metric **M3.1.2** indicates that newly valid products are always inconsistent (i.e., the implementation of the valid products does not match the implementation of the respective ground truth products). To restore consistency in the respective products, metric **M3.1.3** indicates that, in most cases, only one product must be repaired.

Finally, metric **M3.1.4** indicates that for all revisions, the approach hinted at *at least* the number of actually causing features and feature interactions. While the inconsistency was caused in most cases by a missing feature implementation and rarely by a missing feature interaction implementation (compared to the implementation of the respective ground truth products), the approach additionally provided hints for all newly valid feature combinations. For instance, in the second revision, the five hints involved two hints for the newly added features `Label` and `Sort` as well as three hints for the newly possible feature combinations {`Root,Label`}, {`Root,Sort`}, and {`Label,Sort`}.

13.3.4. Discussion

From the computed metrics for the MobileMedia case study, several insights can be derived to answer the questions and discuss the evaluation results.

Q 3.1.1: *How many products become valid after each feature model change?*

The feature model evolves incrementally and constantly in every revision. Per revision, one optional feature is added to the feature model, doubling the number of valid product from the previous revision. Additionally, in the second revision, also a mandatory feature is added as a child of the `Root` feature, which is therefore also a core feature (i.e., selected in every valid configuration). While this does not affect the number of valid configurations, it affects all valid configurations such that all previously valid configurations must now additionally contain the new core feature `Label`.

Q 3.1.2: *Which fraction of the newly valid products is inconsistent?*

The fraction of the newly valid products that are inconsistent is always at 100%. Specifically, all newly valid configurations are missing the implementation of the newly added optional feature. Consequently, every new product has become inconsistent when compared to the ground truth implementation of the respective product. Note that, in every revision, also other changes were performed that were not related to a change in the feature model and thus were not relevant for this type of inconsistency.

Q 3.1.3: *What is the proportion of the manually repaired products to restore consistency in all other affected products?*

The evaluation results indicate a synergy between view-based development and developing variable systems based on products as supported by the unified approach. Since all inconsistent products have the same cause (i.e., the newly added feature) and the new features are optional and independent of the other features (i.e., no dependencies or interactions), they can be implemented in one product and propagated as is to other products. For instance, adding the implementation for the feature `SMS` only requires to change one product and internalize the changes via *iC* once, to let all other products with that feature benefit from it. This effect increases with the number of products.

Q 3.1.4: *How many of the causing features or feature interactions were hinted at by the unified approach?*

The approach always correctly provides a hint for the newly added feature(s) and new pair-wise feature combinations that needed to be implemented in the respective revision. Additionally, it also provides surplus hints at new feature combinations that did not need to be implemented. Interestingly, a case occurred when a feature interaction was hinted at whose implementation was delayed to a later revision. For instance, the combination `Favourite` and `Sort` became valid in the third revision, while its implementation was added later in the fourth revision. Thus, their combination either led to an (undesired) feature interaction or a desired interaction was missing, which was realized by the developers in the following revision.

To summarize, the configurable space of MobileMedia increased constantly by means of added optional features. In most cases, the missing implementation of the respective feature led to an inconsistency. The approach provides hints of all newly added features and new valid pair-wise feature combinations to make the user aware of possible inconsistencies caused by a missing feature implementation or potentially undesired feature interactions.

13.4. Product to Feature Model Consistency

This section presents the evaluation for the Inconsistency Type 5 (*product to feature model consistency*, as described in Section 8.6.2).

13.4.1. Questions and Metrics

Following the overall goal of functional suitability, three questions are asked. The first question concerns the MobileMedia data set:

Q 3.2.1 How many constraints are added to the feature model per revision?

This question can be answered based on metric **M3.2.1**:

Let $CT_r^+ = (TC_r \setminus TC_{r-1}) \cup (CTC_r \setminus CTC_{r-1})$ be the set of newly added tree constraints and cross-tree constraints at revision r.

M3.2.1 = $|CT_r^+|$ is the number of tree constraints and cross-tree constraints that were added at revision r.

To understand whether constraints added in the problem space also exist in the implementation, the second question asks:

Q 3.2.2 How many of these constraints are reflected in the implementation?

This question can be answered based on metric **M3.2.2**:

Let $CT_{I,r}^+ = CT_r^+ \cap CT_{I,r}$ be the set of newly added tree and cross-tree constraints that are reflected in the implementation of the products at revision r.

M3.2.2 = $|CT_{I,r}^+|$ is the number of newly added tree and cross-tree constraints that are reflected in the implementation at revision r.

Finally, the functional suitability of the unified approach is examined:

Q 3.2.3 How many of these constraints can be automatically suggested?

This question can be answered based on metric **M3.2.3**:

Let $CT_{A,r}$ be the set of constraints that are suggested by the approach after all executions of iC at revision r.

M3.2.3 = $|CT_{I,r}^+ \cap CT_{A,r}|$ is the number of newly added constraints that are reflected in the implementation ($CT_{I,r}^+$) and automatically suggested by the approach ($CT_{A,r}$).

13.4.2. Evaluation Process

Figure 13.4 shows a UML activity diagram of the applied evaluation process. For every revision of MobileMedia, a product is externalized via *eP* which provides a product at a selected system revision along with a selected feature revision per feature. Based on the ground truth product at the succeeding system revision, the externalized product is evolved and changes are internalized back via *iC* into the unified system, providing the expression that was

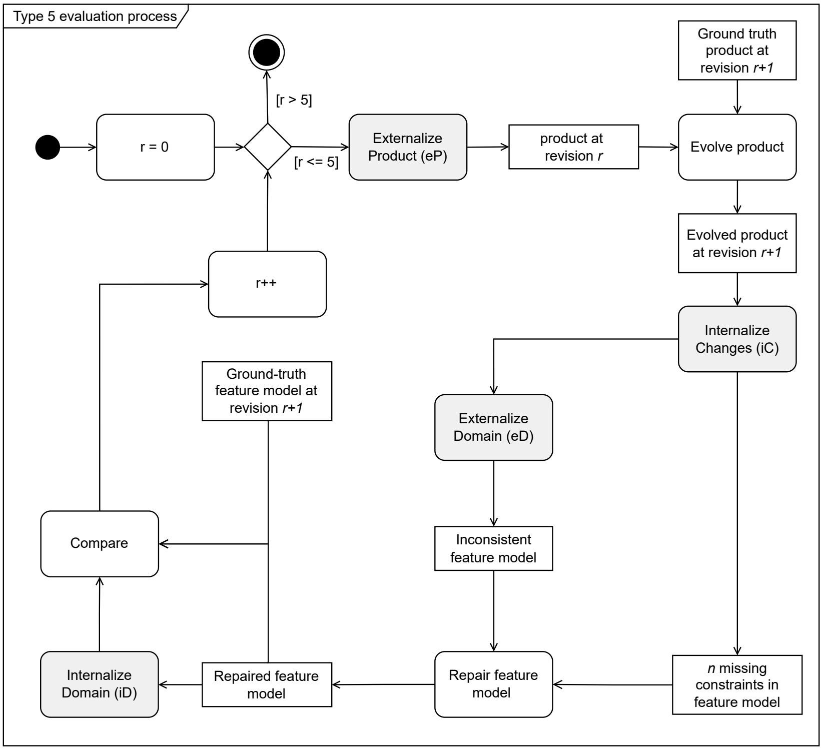

Type 5 evaluation process
r = 0
[r > 5]
[r <= 5]
Externalize Product (eP)
product at revision *r*
Evolve product
Ground truth product at revision *r+1*
Evolved product at revision *r+1*
Internalize Changes (iC)
r++
Ground-truth feature model at revision *r+1*
Externalize Domain (eD)
Inconsistent feature model
Compare
Internalize Domain (iD)
Repaired feature model
Repair feature model
n missing constraints in feature model

Figure 13.4.: UML activity diagram of the evaluation process for product to feature model consistency (Inconsistency Type 5).

manually identified from the evolution history of the MobileMedia data set. After every internalization operation of changes, the approach performs a dependency analysis of the solution space. In case constraints are identified that are not reflected on the problem space between features, the domain is externalized via *eD* at the succeeding system revision and repair the feature model by adding the missing constraints to it. Finally, the changed feature model is internalized via *iD* and the repaired feature model is compared with the ground truth feature model of the succeeding system revision. The complete process is performed for each revision of MobileMedia.

13.4.3. Results

Table 13.4 shows the results of the evaluation for Inconsistency Type 5. Each cell shows the results for one particular metric (**M.3.2.1**–**M3.2.3**) (columns) per revision (rows).

Metric **M3.2.1** indicates that at least one constraint is added in every revision. In the first revision, the `Root` feature is added to the feature model that must always be selected in any valid configuration. In the second revision, two constraints are added to the feature model: `Label` is added as *mandatory* feature and `Sort` is added as *optional* feature. In the following three revisions, every added feature (i.e., `Favourite, Copy, SMS`) is *optional.*

Out of the added constraints to the feature model (i.e., the problem space), metric **M3.2.2** answers how many of these are actually reflected in the implementation (i.e., the solution space). Therefore, the annotated implementation of the data set was manually inspected at every revision. Note that dependencies of the data set are not exhaustively analyzed manually, but instead a dependency between features is considered present if at least one dependency between their implementing artifacts could be detected.

In the first revision, the `Root` feature has been added. As it is the only feature at this revision, there are no dependencies to other features yet. In the second revision, the features `Label` and `Sort` are added as children of the `Root` feature and thus depend on it. This dependencies are reflected in the implementation, since several elements of the `Sort` and `Label` features are contained in, or refer to elements of the `Root` feature. The `Sort` feature is added as optional in the feature model which could be confirmed in the implementation as no element of any other feature is contained in, or referring to any element of the `Sort` feature. Although the `Label` feature was documented as mandatory child of `Root` in the feature model, no dependency was found of the `Root` feature's implementation to the implementation of the `Label` feature in this revision. Since an optional child feature represents a dependency of the child to the parent feature (instead of additionally a dependency of the parent feature to the child feature, which would represent a mandatory child feature), this case was counted as 1.5 out of 2 reflected constraints. Interestingly, in the following third revision, elements of the `Root` feature indeed referred to elements that were added when the `Label` feature was added, which thereby also become mandatory in the implementation. However, this is a special case: the implementation of the `Label` feature could not be distinguished

Table 13.4.: MobileMedia evaluation results for Type 5.

Rev. \ Metrics	M3.2.1	M3.2.2	M3.2.3
1	1 (+Root)	1/1	1/1 (100%)
2	2 (+mand. Label, +opt. Sort)	1.5/2	1.5/1.5 (100%)
3	1 (+optional Favourite)	1.5/1*	1.5/1.5 (100%)
4	1 (+optional Copy)	1/1	1/1 (100%)
5	1 (+optional SMS)	2/1**	2/2 (100%)

* Core ⇒ Label (Label becomes mandatory in solution space one revision later wrt. the problem space)

** SMS ⇒ Copy (solution space dependency is not reflected in ground truth problem space)

from the `Root` feature's implementation in subsequent revisions as it is not annotated in the implementation of MobileMedia. In the fourth revision, the `Copy` feature can be confirmed to be an optional child of the `Root` feature. In the fifth revision, the `SMS` feature is added as optional child of `Root`. However, additionally, the `SMS` feature also requires the `Copy` feature, since a statement in the `SMS` implementation calls a method in the `Copy` implementation.

Finally, metric **M3.2.3** shows that the unified approach could automatically and correctly identify and add all dependencies among features to the feature model that also exist in the implementation.

13.4.4. Discussion

From the computed metrics for the MobileMedia case study, several insights can be derived to answer the questions and discuss the evaluation results.

Q 3.2.1: *How many constraints are added to the feature model per revision?*

Per revision, at least one new feature and corresponding constraint is added to the feature model. All new features are children of either the `Root` feature or another core feature. Moreover, all features except for `Label` are optional, and there are no constraints between any of the features other than their dependency to the `Root` feature. Thus, the implementations of the optional features must also be independent of each other. Furthermore, elements in their implementation may only be contained in, or refer to elements of the implementation of the `Root` feature or the mandatory `Label` feature. Otherwise, products with an invalid implementation could be externalized. An exception is the `Label` feature. Since it is also a core feature, all other features, including the `Root` feature, may depend on it.

Q 3.2.2: *How many of these constraints are reflected in the implementation?*

Interestingly, the actual implementation does not always correspond to the documentation of the evolved feature model. For instance, in revision two, the `Label` feature is initially *optional* in the solution space and becomes mandatory one revision after it was documented as mandatory in the problem space. While this does not cause an inconsistency, it might be unnecessarily restrictive. Moreover, in revision five, the implementation of the optional `SMS` feature requires the implementation of the optional `Copy` feature. This is interesting, as it is not reflected in the ground truth feature model which is therefore inconsistent with the ground truth implementation. Thus, it is possible to externalize four products with compilation errors, since they comprise the `SMS` feature but not the `Copy` feature.

Q 3.2.3: *How many of these constraints can be automatically suggested?*

Based on the dependency analysis between features in the solution space, the unified approach is able to detect all dependencies, lift them to the problem space, and automatically add them to the feature model in case they are not reflected in the problem space. This was not only sufficient to automatically add all constraints present in the ground truth feature model, but also to add an additional constraint that was wrongfully missing in the ground truth feature model. The results indicate that the approach is able to reduce manual effort for developers and also to reduce the chance of human error. Please note that, in most cases, more than one valid repair exists and that the approach chooses one such valid repair. Which repair is suggested and applied automatically depends on the order in which mappings are processed and can thus vary.

13.5. Product Consistency

This section presents the evaluation for the Inconsistency Type 6 (*product consistency*, as described in Section 8.6.3).

13.5.1. Questions and Metrics

Following the overall goal of functional suitability, several questions are asked. Since this inconsistency type particularly addresses inconsistencies that can occur between different types of artifacts, the first question asks:

Q 3.3.1 How many products become inconsistent after evolving one artifact type of a variable system via unified view-based operations (a) *without* consistency preservation and (b) *with* consistency preservation?

The question Q 3.3.1a can be answered based on metric **M3.3.1a**:

> $D_{r,c}$ is the set of differences between the original ground truth UML model of the product with configuration c at revision r and the product with configuration c at revision $r - 1$.
>
> **M3.3.1a** = $|\{P_c \mid D_{r,c} > 0\}|$ is the number of products for which there is at least one difference after evolution without consistency preservation (i.e., only considering the ground truth products).

The question Q 3.3.1b can be answered based on metric **M3.3.1b**:

> $\hat{D}_{r,c}$ is the set of differences between the original ground truth UML model of the product with configuration c and the product generated by the unified approach via the *eP* operation with configuration c at revision r.
>
> **M3.3.1b** = $|\{P_c \mid \hat{D}_{r,c} > 0\}|$ is the number of products for which there is at least one remaining difference to the ground truth product after evolution with consistency preservation.

The second question is concerned with the effort for repairing inconsistent products and how much of it can be reduced by the unified approach.

Q 3.3.2 How many of the necessary changes to repair an inconsistent product could be performed automatically?

The question Q 3.3.2 can be answered based on metric **M3.3.2a** and **M3.3.2b**:

M3.3.2a = $|D_{r,c} \setminus \hat{D}_{r,c}|$ is the number of necessary changes that could be performed automatically.

M3.3.2b = $|D_{r,c}|$ is the number of necessary changes.

13.5.2. Evaluation Process

Figure 13.5 shows a UML activity diagram of the applied evaluation process. In every revision of ArgoUML-SPL and MobileMedia, both product lines are gradually constructed, starting from a single product only consisting of the core features. The product space is iteratively increased by applying *eP* and adding each new feature and respective feature interaction based on the ground truth product at the succeeding system revision only to a single artifact model (Java) of a single product each. After every change, the consequential changes of Vitruvius are executed to preserve consistency in the UML model. Finally, changes that have been applied based on the ground truth product are integrated into the unified system via *iC*. For each revision r, the set of differences $\hat{D}_{r,c}$ is computed between the original ground truth UML model of each respective product with configuration c, extracted by Vitruvius directly from the Java source code generated from the ArgoUML-SPL and MobileMedia at revision r, and the UML model constructed by the externalization operation *eP* of the unified approach. To put the results into perspective, also the set of differences $D_{r,c}$ is computed between the original UML model of each product with configuration c at revision $r-1$ and r. This reflects the effort that would be needed to manually evolve each UML model from revision $r-1$ to revision r.

13.5.3. Results

Table 13.5 shows the evaluation results for the ArgoUML-SPL, and Table 13.6 shows the evaluation results for MobileMedia. Each cell shows the number of automatically performed changes (**M3.3.2a**) out of the total number of changes (**M3.3.2b**) per configuration (rows) and revision (columns) in the form $M3.3.2a/M3.3.2b$. The last row shows the number of inconsistent products without consistency preservation (**M3.3.1a**) and with consistency preservation (**M3.3.1b**) in the form $M3.3.1a/M3.3.1b$. Configurations that either have not existed yet or do not exist anymore in the respective revision are represented by an empty cell with a dash.

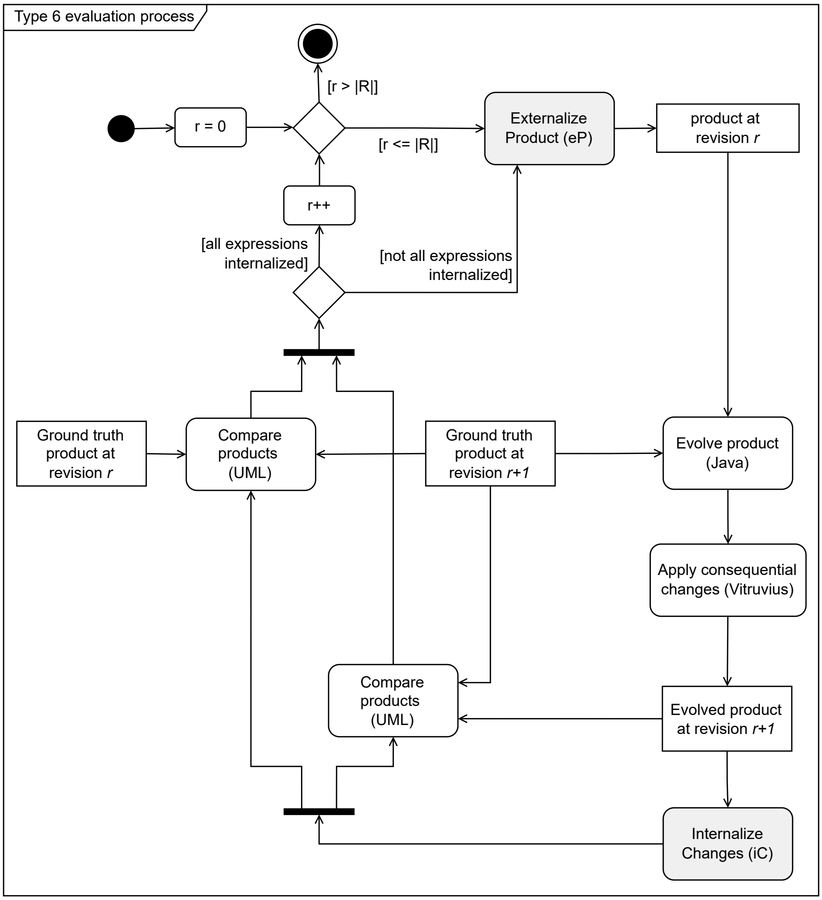

Type 6 evaluation process
r = 0
[r > |R|]
[r <= |R|]
Externalize Product (eP)
product at revision *r*
r++
[all expressions internalized]
[not all expressions internalized]
Ground truth product at revision *r*
Compare products (UML)
Ground truth product at revision *r+1*
Evolve product (Java)
Apply consequential changes (Vitruvius)
Compare products (UML)
Evolved product at revision *r+1*
Internalize Changes (iC)

Figure 13.5.: UML activity diagram of the evaluation process for product consistency (Inconsistency Type 6).

Metric *M*3.3.1*a* shows that, without automated consistency preservation, in seven of nine revisions of ArgoUML-SPL and in all five revisions of MobileMedia, the UML model of at least one product became inconsistent with its Java model. With automated consistency preservation, the Java and UML models of all products in all revisions of both case study systems could always be kept consistent.

Table 13.5.: ArgoUML-SPL evaluation results for Type 6.

	Configuration \ Revision	0	1	2	3	4	5	6	7	8	9
M3.3.2a/M3.3.2b	{ Core, Class }	0/0	26/26	0/0	18/18	0/0	0/0	4/4	0/0	3/3	0/0
	{ Core, Class, Cognitive }	0/0	26/26	0/0	18/18	5/5	3/3	4/4	0/0	3/3	0/0
	{ Core, Class, Logging }	0/0	26/26	0/0	18/18	0/0	0/0	4/4	0/0	3/3	4/4
	{ Core, Class, Activity }	0/0	26/26	0/0	18/18	0/0	0/0	4/4	0/0	3/3	0/0
	{ All }	0/0	26/26	0/0	18/18	5/5	3/3	4/4	0/0	3/3	4/4
M3.3.1a/M3.3.1b		0/0	5/0	0/0	5/0	2/0	2/0	5/0	0/0	5/0	2/0

Table 13.6.: MobileMedia evaluation results for Type 6.

	Configuration \ Revision	0	1	2	3	4	5
M3.3.2a/M3.3.2b	∅	0/0	-	-	-	-	-
	{ Core }	-	630/630	-	-	-	-
	{ Core, Label }	-	-	150/150	12/12	306/306	0/0
	{ Core, Label, Sort }	-	-	171/171	12/12	326/326	0/0
	{ Core, Label, Fav. }	-	-	-	25/25	306/306	0/0
	{ Core, Label, Sort, Fav. }	-	-	-	25/25	326/326	0/0
	{ Core, Label, Copy }	-	-	-	-	341/341	7/7
	{ Core, Label, Sort, Copy }	-	-	-	-	361/361	7/7
	{ Core, Label, Fav., Copy }	-	-	-	-	341/341	7/7
	{ Core, Label, Sort, Fav., Copy }	-	-	-	-	361/361	7/7
	{ Core, Label, Copy, SMS }	-	-	-	-	-	230/230
	{ Core, Label, Fav., Copy, SMS }	-	-	-	-	-	230/230
	{ Core, Label, Sort, Fav., Copy, SMS }	-	-	-	-	-	230/230
	{ Core, Label, Sort, Copy, SMS }	-	-	-	-	-	230/230
M3.3.1a/M3.3.1b		0/0	1/0	2/0	4/0	8/0	8/0

For all products in all revisions of both case study systems, it holds that **M3.3.2a** = **M3.3.2b**, indicating that all changes were successfully propagated to all affected products during view-based development and that the UML model was kept consistent with the Java model fully automatically. The value of *M*3.3.2*b* indicates how many changes were needed to automatically propagate changes during externalization. For example, in ArgoUML-SPL, the product with configuration $\{Core, Class, Logging\}$ in revision three yielded a total of 18 changes, such as the addition of a method, the change of the visibility or input parameters of a method, to the previous revision of the

UML model of the same product. None of them remained after the automated change propagation, i.e., all 18 changes could be propagated fully automatically to the UML model after editing the Java model. In contrast to ArgoUML-SPL, MobileMedia is developed from scratch and evolves in larger increments as new features are still added in every revision. Consequently, the number of products increases per revision while in ArgoUML it remains identical. Notably in ArgoUML-SPL, changes whose expression involved more than one feature, i.e., a feature interaction, were always very fine-grained and, thus, did not affect the UML models.

13.5.4. Discussion

The evaluation shows promising results for the functional suitability of the unified approach. It provides initial evidence for the synergy between view-based development of variable systems and automated view-based consistency preservation. From the computed metrics for the real-world case studies ArgoUML-SPL and MobileMedia, several insights can be derived to answer the questions and discuss the evaluation results.

Q 3.3.1a: *How many products become inconsistent after evolving one artifact type of a variable system via unified view-based operations without consistency preservation?*

Without automated consistency preservation, over all nine revisions with five products each in ArgoUML-SPL, the UML models of 26 products became inconsistent with the Java model and required to be fixed manually. For MobileMedia, over all five revisions and 27 products, the UML models of 23 products became inconsistent with the Java model. Thus, roughly half of the products of ArgoUML-SPL and almost all of the products of MobileMedia became inconsistent.

Q 3.3.1b: *How many products become inconsistent after evolving one artifact type of a variable system via unified view-based operations with consistency preservation?*

With automated consistency preservation, none of inconsistent products required manual repair. All changes in the Java models could automatically and correctly be propagated to the UML models of all affected products.

Q 3.3.2: *How many of the necessary changes to repair an inconsistent product could be performed automatically?*

Any change performed in one artifact model of a specific product is automatically propagated to other depending models and products. The proportion of the number of automatically performed repair operations $M3.3.2a$ to the number of overall operations $M3.3.2b$ necessary to repair a product is in all cases 100%. The developer neither needs to maintain mappings nor manage dependencies and redundancies across different types of artifacts manually, which is highly error-prone [180]. Finally, the results indicate that the approach is suitable for dealing with changes of variability in space, e.g., the addition of new features or feature interactions, variability in time, e.g., modifications of existing features and the addition of new feature revisions and system revisions, and heterogeneous types of artifacts.

13.6. Implementation

The unified approach was implemented in Java using the Eclipse Modeling Framework (EMF) [225]. Thus, the concrete metamodel is realized as an Ecore model. To parse Java source code and represent it as Ecore model, the Java Model Parser and Printer (JaMoPP) was employed [92].[5] Minor non-intrusive adaptions to the ArgoUML-SPL source code were necessary to be able to successfully parse it with JaMoPP, e.g., adding imports of static fields or methods to ensure that Ecore proxy objects could be resolved. Since the unified approach relies on deltas as variability mechanism, EMFCompare[6] was used to diff successive revisions of Java model instances of products and compute edit scripts between them. Additionally, EMFCompare was employed to compute the differences between UML model instances of products, and thus, determining the evaluation metrics. Finally, the unified approach is integrated with Vitruvius [116], as described in Section 8.2.2. Specifically, its incremental consistency preservation capabilities are utilized for propagating changes between different artifact models of a product, i.e., Java and UML as well as between different products.

[5] `https://github.com/DevBoost/JaMoPP`

[6] `https://www.eclipse.org/emf/compare/`

13.7. Threats to Validity

This section describes threats to the validity and how they were mitigated.

Internal Validity. Threats to internal validity target the real-world case study ArgoUML-SPL. While ArgoUML kept evolving, the ArgoUML-SPL has not been co-evolved and remained in its inital revision. To make reliable claims about the approach and its suitability to support the evolution of a variable system, ArgoUML-SPL was manually evolved by retroactively replaying, i.e., comparing and merging, the revision history of ArgoUML on the ArgoUML-SPL. This threat was mitigated by closely inspecting the merged changes and careful documentation, providing an artifact to be further validated and applied.[7]

Construct Validity. In contrast to ArgoUML-SPL, a documentation of the feature model evolution is provided [71]. To increase construct validity, I verified dependencies between features (documented in the feature model's evolution) by additionally analyzing the provided implementation of Mobile-Media, which is reflected by Metric **M3.2.2**. A threat to construct validity is that the dependency analysis was performed manually, which was, however, inevitable since state of the art does not provide techniques to extract a feature model fully automatically from arbitrary implementation artifacts [123]. Closest work is provided by Nadi et al. [165] to extract configuration constraints from C code. Moreover, Mendonça et al. [158] present an approach for reverse engineering feature models based on a multi-objective optimization algorithm, which however assumes solution space dependencies between features as input for the optimization.

External Validity. Since the unified approach was applied to a single case study for inconsistency Type 2 and 5 and to two case studies for inconsistency Type 6 involving Java and UML, the results may be not easily generalizable to other data sets and artifact models, which threatens the external validity. Therefore, I am currently applying the approach to other real-world case studies, e.g., in the automotive domain. Moreover, one might question whether the used data set is representative, as it only contains a relatively small number of features. Since the ArgoUML-SPL is a real-world system that has been

[7] `https://github.com/SofiaAnanieva/argouml-spl-evolved`

widely adopted by the SPL community and is commonly applied as benchmark system, this case study strengthens the comparability of this work.

13.8. Limitations and Future Work

This section comprises a discussion of the limitations of the unified approach as well as future work to address these.

The unified conceptual model was refined to support feature modeling, which is a de-facto standard for variability modeling in research and industry [117, 194, 50, 107]. Although other types of variability models are not explicitly supported, feature models can be automatically transformed into other forms of variability models [66]. However, this would require an additional transformation step. As another limitation of the approach, it currently does not support extensions to feature modeling such as *cardinalities* to specify the number of clones for a given feature [193, 52], or *attributes* to include more information about features [107, 29, 32]. Future work could address these shortcomings by extending the concrete metamodel of the unified approach shown in Figure 8.2 accordingly.

The unified conceptual model (C1) and unified operations (C2) were conceived to support distributed development. In case of the unified conceptual model, this is represented by the relation of a unified system to multiple other unified systems. In case of the unified operations, the operations eUS and iUS allow for cloning of a unified system and internalizing changes back into the original unified system. However, the unified approach (C5) is limited to local operations and, thus, does currently not support distributed development. This is considered a limitation of the implementation and not of the conceptual basis of the approach as it builds on prior contributions that explicitly support distributed development.

As explained in Section 6.2.1, operations could be classified according to the edit modality (i.e., direct or view-based editing) and development paradigm (i.e., product-oriented or platform-oriented development). The unified approach employs *view-based editing* and supports *platform-oriented development.* While they have their distinct advantages, this choice also comes with limitations of the approach that are discussed in the following. The unified approach is limited to unified operations that support view-based editing and platform-oriented development (i.e., *eP*, *iC*, *eD*, *iD*). Thus, it promotes the evolution of a variable system via views that represent a particular product of the variable system hiding the variability mechanism from the user as proposed by the upcoming research area of VarCS (see Definition 2.2). This enables a high degree of automation for evolving variable systems (e.g., the automated creation of feature revisions and system revisions, the computation of mappings, and automated consistency preservation), which provides great aid for developers as it reduces cognitive complexity [214, 141]. However, the unified approach inherits the limitation of view-based development via product views by which it is challenging to arbitrarily modify mappings between fragments and features without modifying the fragments, since they are computed fully automatically and cannot be edited directly. An idea and future work to address this limitation is to combine both edit modalities to leverage benefits of both, as described in Section 12.7. However, the challenge would be to provide an editable view that includes mappings, since they require to be displayed differently for every type of artifact.

Moreover, the unified approach is limited to unified operations that support platform-oriented development. Consequently, changes cannot be integrated conveniently by simply providing the changed product, but requires the solution space engineer to manually provide a Boolean expression which comprises the features and feature interactions that are affected by the changes upon the operation *iC*. An idea and future work to address this limitation is to support an arbitrary alternation between both paradigms to leverage benefits of both. However, product-oriented edits do not provide fine-grained mappings that are necessary for platform-oriented editing. Thus, the unified approach would require to additionally employ further techniques such as *feature trace recording* [39] or *feature location* [199, 20] to automatically extract fine-grained mappings during product-oriented development.

A debatable limitation of the unified approach is its application to legacy systems, such as the Linux kernel [233]. The approach is applicable in a greenfield scenario where the underlying data structure is populated with constructs, i.e., features, mappings, fragments, feature revisions and system revisions, incrementally as the system evolves. As described in Section 1.1, legacy systems using current tool support require sophisticated approaches to retroactively extract or mine information, such as the evolution of features (that is explicitly modeled by the unified approach). To apply the unified approach to an existing variable system (e.g., using a preprocessor in combination with Git as is the case for the Linux kernel), several possibilities can be considered as future work: the migration of legacy systems could, on the one hand, be performed by automatically replaying annotated version histories to automatically compute feature-to-fragments mappings [195], or, on the other hand, by reverse engineering a product line from a set of products [155, 128].

Moreover, the evaluation considers the propagation of changes from models of lower abstraction (Java) to models of higher abstraction (UML). Thus, the consistency preservation mechanisms of Vitruvius can restore consistency fully automatically. The opposite direction is more challenging and requires user interaction in cases where multiple valid repair options exist. As Vitruvius already supports semi-automatic repairs, such scenarios open up potential for future work. Specifically, the user decisions should also be stored and mapped to features so that they can be replayed during product externalization.

A current limitation of the unified approach in this regard is, that it is not possible to freely change the direction of consistency preservation. Once an artifact has been chosen for applying original changes to (e.g., Java) further original changes cannot be applied to another artifact type created by change propagation (e.g., UML). This is caused by the fact that consequential changes may differ across products, and elements created by consequential changes cannot be guaranteed to receive the same unique identifiers in every product as, in contrast to original changes, consequential changes are not stored in the unified system. Thus, it is not easily possible to have original changes refer to elements created by consequential changes.

Another conceptual limitation of the unified approach and important future work is the ability to better deal with varying insertion positions of elements that depend on the context in which they are inserted, as other features may have added or removed surrounding elements: deltas that are recorded in a product view and applied in another product may insert the elements not at the right position, as the insertion position of elements depends on the configuration of the product in which they are inserted. As a consequence, the insertion position used by delta operations cannot be determined by a constant index. Instead, a (partial) order relation among them and all their surrounding elements is needed to determine the appropriate insertion position in each configuration. This is a special case of feature interaction where not new elements must be inserted into a product's implementation to address it, but where the order among existing elements that map to the interacting features must be determined.

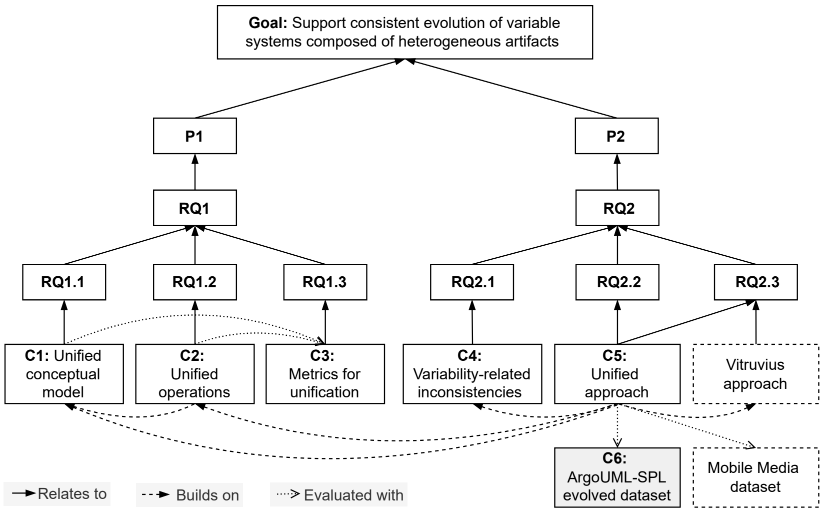

Goal: Support consistent evolution of variable systems composed of heterogeneous artifacts
P1
P2
RQ1
RQ2
RQ1.1
RQ1.2
RQ1.3
RQ2.1
RQ2.2
RQ2.3
C1: Unified conceptual model
C2: Unified operations
C3: Metrics for unification
C4: Variability-related inconsistencies
C5: Unified approach
Vitruvius approach
C6: ArgoUML-SPL evolved dataset
Mobile Media dataset
Relates to
Builds on
Evaluated with

Figure 13.6.: Contribution of Chapter 13 of the thesis.

Regarding the consistency preservation of Type 2, the current implementation only considers feature combinations where all features in a combination are positive (i.e., selected). However, albeit much less common, there can also be feature combinations where one or more features are negative (i.e., deselected), as also the absence of a feature (i.e., a feature being deselected in a configuration) can cause an interaction with other features. Note that such a case did not occur in the data sets used in the case studies.

Finally, the evaluation of the unified approach is a preliminary proof of concept. It requires further evidence to be gathered by additional real-world case studies comprising artifact types from engineering disciplines other than software engineering.

13.9. Summary and Conclusion

This chapter presented an evaluation of the functional suitability of the unified approach. First, two real-world case studies were presented, the well-known ArgoUML-SPL [49] and the MobileMedia [71]. Since the ArgoUML-SPL has not been co-evolved and remained in its inital revision while the origin ArgoUML kept evolving, the ArgoUML-SPL was evolved by replaying the original changes from the ArgoUML. Thus, a data set is provided that also represents the final contribution of this thesis (C6). For every considered inconsistency type (i.e., Inconsistency Type 2 (*feature model to product consistency*, see Section 8.6.1), Type 5 (*product to feature model consistency*, see Section 8.6.2), and Type 6 (*product consistency*, see Section 8.6.3), an evaluation was presented following the GQM method [27]. The results showed that all three inconsistency types occur during the evolution of the real-world case study systems and that the unified approach is capable of detecting and repairing these, ranging from hints to the developer to fully automated consistency preservation. Nonetheless, there are several limitations of the unified approach, such as the arbitrary modification of mappings or its application to legacy systems. To sum up, the evaluation showed promising results for the functional suitability of the unified approach and provides initial evidence for the synergy between view-based development of variable systems and automated view-based consistency preservation for variability-related inconsistencies caused or repaired in the solution space.

Figure 13.6 shows an overview of the contributions of the thesis and highlights the contribution of this chapter (C6) in grey.

Part IV.

Reflections

14. Related Work

This chapter builds on publications at VaMoS [6], VariVolution [9], SPLC [3, 7], and Empirical Software Engineering [5].

This chapter presents related work for each of the major contributions of this thesis: the unified conceptual model (see Chapter 5), the unified operations (see Chapter 6), and the unified approach (see Chapter 8).

Section 14.1 comprises related work to the unified conceptual model. Section 14.2 encompasses a discussion of work related to the unified operations. Related work to the unified approach in Section 14.3 closes this chapter.

14.1. Unified Conceptual Model

Conceptual models were initially proposed in several areas of computer science to model conceptualizations of a domain used for purposes of understanding, communication and problem solving [42, 88]. Conceptual models represent mental representations comprised of concepts and relationships between them, while clarifying the meaning of various, usually ambiguous terms. In both the SPLE and SCM community, much research has focused on concepts and terminology, resulting in various conceptualizations of the respective research area. In this section, conceptual models of both communities as well as related surveys of variability in space and time are discussed, and related with this research.

Conceptual Models for Variability in Space. Research in software product-line engineering (SPLE) proposes several modeling concepts and taxonomies to describe and specify concepts of variability in space, such as *variation points* or *variants* [69, 191, 12, 178]. Despite these efforts, varying terminologies have evolved. For instance, the term *variant* can constitute a product (e.g., in the tool ECCO (see Table 11.1)), a configuration (e.g., in the tool FeatureIDE (see Table 11.1)), or an option of a variation point in the orthogonal variability model [191]. In SPLE, a prominent way of specifying terminology and providing conceptual models for the domain is *variability modeling* [191, 12, 51, 206, 167]. Although several types of variability models exist, concepts of variability in space along with its varying terminology have never been unified. Further limitations of existing models are that they are specific to certain tools or that they only consider concepts from the solution space or the problem space. The presented unified conceptual model tackles these problems from a wider perspective by considering the unification of concepts and relations from both the SPLE and SCM research areas. The unified conceptual model describes systems with variability in space and time in both the problem space and solution space while employing names that are neither associated with SPLE nor SCM terminology nor have ambiguous definitions.

Conceptual Models for Variability in Time. Likewise to SPLE, research in software configuration management (SCM) proposes conceptual models, taxonomies and terminology to describe and specify concepts of variability in time, such as *revisions* or *changes* [109, 201, 145, 190, 151]. Prominent conceptual models in SCM are arguably *version models* for managing changes within directories or files that describe several concepts for version control, such as the supported graph topology or the objects to be versioned. Conradi and Westfechtel [48] study several approaches from SCM and show that different version models, versioning paradigms and concepts are employed with a varying terminology, conceptual differences and whether they stem from the product space (i.e., the solution space in SPLE [7]) or the version space (i.e., the problem space in SPLE [7]). The authors propose an overview of version models while defining and relating fundamental concepts. An obvious conceptual difference compared to SPLE is the semantics of the term *version*. While in SPLE it is commonly used to describe the state of a system that supersedes the previous state, in SCM a version describes an abstract concept that is specialized by either a *revision* (i.e., a version that is intended to supersede its predecessor) or a *variant* (i.e., versions that are intended to coexist). While this pioneer work already relates terms and concepts, both research areas were not far developed and some of the modern concepts (such as `Feature Revisions`) could not be considered.

Related Surveys of Variability in Space and Time. As described above, Conradi and Westfechtel [48] classify multiple approaches used for version control and relate fundamental concepts and terminology. Consecutively, Westfechtel et al. [241, 47] propose the *Uniform Version Model (UVM)* along with its underlying *layered architecture* which is considered the closest research to the unified conceptual model. Likewise to the conceptual model which does not prescribe a particular variability model but basic concepts for any variability model (i.e., `Options` and `Constraints`), the UVM is also independent from particular version models as it employs a common base consisting of version rules (i.e., `Constraints`) to allow for realizing different version models. Also, both models are designed to support different dimensions of evolution (i.e., variability in space and time) and any structure of `Fragments` (i.e., tree-based or graph-based). In contrast to the unified conceptual model, the authors describe the UVM to be built only on a small number of selected concepts while employing implementation specifics like deltas or propositional logic to manage variability.

Schwägerl [212] builds upon the UVM and contributes an extension that constitutes a conceptual framework for the integration of SPLE and SCM based on MDSD. In contrast to the unified conceptual model, several design decisions are made while developing the conceptual framework, such as employing symmetric deltas and, in case of concurrent modifications, a three-way merge support for model-driven software product lines. Based on the conceptual framework, Schwägerl proposes the tool `SuperMod`, which was part of the study to derive the unified conceptual model that is independent of realization by abstracting from implementation details of a certain tool. Linsbauer et al. [141] classify and compare several VarCS (see Definition 2.2) and present core concepts, such as `Revisions` or variable `Entities` (i.e., `Features`). Several variation control systems of the study were also included in the unification. Thus, tools were analyzed that have been published more recently (i.e., `VTS`, `SuperMod`, and `ECCO`).

Mahmood et al. [152] propose the *virtual platform*, a method for incrementally transitioning from clone-and-own to software configuration with an integrated platform. The proposed method builds on earlier work that introduces governance levels ranging from clone-and-own [61] to a fully integrated platform [11]. In this work, the authors propose conceptual structures and operators that relate to the unification effort. Similar to the process for conceiving the unified conceptual model and its concepts and relations, the authors survey the literature and propose concepts and relations to support both strategies. While some concepts for variability in space and/or time overlap (e.g., `Features` and `Revisions`), others are less generic by specifying i) feature modeling (whereas the conceptual model employs the concepts `Feature Option` and `Constraint` to allow for arbitrary types of variability models), ii) a tree structure (whereas the `Fragments` in the unified conceptual model can be organized as a graph that is a generalization of a tree structure), and iii) presence conditions to be used for `Mappings` (whereas the form of a `Mapping` is not specified in the unified conceptual model). While this research aimed for appropriate unification based on the elicited tools, the concepts of the virtual platform were selected by the authors to support development activities targeting clone management and the transition to an integrated platform.

Finally, there is research that analyzes and compares approaches and tools for SPLE or SCM [189, 185, 26, 201, 77]. In contrast to the unified conceptual model, these works do not focus on unifying the concepts and relations of the identified tools, but on classifying and comparing them.

14.2. Unified Operations

Rubin et al.[198, 200] analyze three industrial case studies following the clone-and-own practice. The authors propose a *cloned product line management framework* that comprises seven conceptual operators to i) support the transition from cloned products to a product line and ii) maintain the existing clones in a more efficient manner. For instance, the authors propose an operator *find Features* that returns a set of features of a particular product, the operator *interact?* that determines interactions between an arbitrary number of feature implementations based on features while specifying the form of interactions to be checked, and the operator *merge* that combines several systems into a single system based on artifacts that are considered similar and a specification for resolving interactions between input functionalities. While this work relates to the unified operations by means of the high abstraction level of the operators and discussing sequences of their application, the pragmatics of the operations differ, leading to a different set of operations. The authors propose operators for managing cloned variants whereas unified operations target the wider perspective of supporting the evolution of a system that copes with variability in space and time based on a reusable platform (i.e., the `Unified System`). Moreover, sources of data differ. While the unified operations are conceived based on an expert survey starting from concrete individual tool operations, the proposed conceptual operators were derived from observing three industrial organizations and subsequently applying the conceptual operators to concrete development activities. Finally, the *cloned product line man-*

agement framework does neither explicitly consider pre and post-conditions of operations nor view-based operations and their degree of automation.

As described in Section 14.1, Mahmood et al. [152] propose the *virtual platform*, a method for continuously recording meta-data (e.g., mappings between features and their locations or clone traces among artifacts) during clone-and-own [61] to incrementally transition to software configuration with a reusable platform. The proposed work builds on earlier work that introduces governance levels representing a spectrum between ad hoc clone-and-own and a fully integrated platform [11]. In this work, the authors propose conceptual structures that form the basis of operators for the virtual platform that relates to the unification effort. Besides similarities and differences of the employed concepts to those of the unified conceptual model (see Section 14.1), the authors propose two categories of operators: *traditional asset-oriented* such as *Add Asset* or *Map Asset To Feature* to manage assets (i.e., `Fragments`) and *feature-oriented* such as *Add Feature* or *Propagate To Feature* devoted to features and their locations in assets. Since the proposed operators build on the conceptual operators proposed by Rubin et al. [200], again the purpose of the operations is significantly different from the purpose of the unified operations to support the evolution of a variable system presuming a reusable platform. Nonetheless, it can be argued that the unified operations are also capable of supporting clone traces as the virtual platform via the unified operation `externalize Unified System` that is mapped to the *clone* operation of `Git`. Indeed, it would be interesting to include the virtual platform as another tool in the set of studied tools for the unified conceptual model and the unified operations to consider an even broader set of evolution purposes.

Further related work is conducted by Hinterreiter et al. [99, 98]. The authors propose local and distributed operations for feature-oriented development [99]. Local operations only affect one platform such as *Commit Features* while distributed operations involve multiple platforms, such as *Clone Features*. Since this work is based on the tool `ECCO` that was part of the considered set of tools for conceiving the unified operations, the operations proposed by the authors are covered by the unification. The authors also compare approaches dealing with *temporal feature modeling (TFM)* for capturing the evolution of a feature model by its change history or the planning of future releases, and propose common operations for temporal feature modeling (i.e., the *TFM API*) [98]. Such operations comprise, for instance, the creation of a feature or the modification of a feature group type. Highly similar to the process of devising the unified operations, the authors conceive the API by analyzing a set of tools while aiming for, as the authors call it, *harmonization*. While the purpose of harmonization can be considered similar to the purpose of unification, it only takes into account the coverage (i.e., whether the TFM API covers all operations of the respective tools, yet does not provide more operations than that) and not the granularity (i.e., whether a harmonized operation targets one particular concern). While this work partially considers the same tools (i.e., DarwinSPL, DeltaEcore and FeatureIDE), it particularly targets feature modeling.

Further research is closely related with the unified operations: Projectional editing [240] is proposed for SPLE to reduce the complexity when developing a variable system by introducing (partial) views on variable systems which is essentially the same as view-based editing. Since projectional editing is the foundation of VTS, this edit modality is incorporated in the unification, specifically the operation *Externalize Unified System* that provides a partial view on the `Unified System`. Also closely related is the study of variation control systems performed by Linsbauer et al. [141]. The authors identify two general types of operations, namely *Internalization* operations to modify a variable system and *Externalization* operations that create output from it. While these general types were used to scope the unified operations and align terminology, they were also considerably extended. For example, by explicitly specifying the input and output of the operations and distinguishing between different internalization operations such as *Internalize Changes*, *Internalize Product* or *Internalize Domain*.

Finally, Westfechtel et al. [241] introduce the uniform version management. As described in Section 14.1, the *uniform version model (UVM)* comprises and relates concepts of variability in space and time and proposes view-based operations as common for SCM. While the operations are not specified thoroughly, the rather recent tool SuperMod builds on the UVM and is part of the set of tools studied in this thesis. As a consequence, the proposed operations are considered for the unification.

14.3. Unified Approach

The unified approach supports the consistent, view-based development and evolution of variable systems composed of heterogeneous artifacts which is a relevant and challenging problem. In the following, related work to the unified approach is presented.

Consistency-Aware Variability Management

One of the closest research is conducted by Nieke et al. [169, 174, 175, 100, 171, 173, 176, 170]. The authors present several contributions to support the consistent evolution of an SPL. First, by supporting the creation and re-planning of feature model evolution plans [100, 176]. As a starting point, the authors propose a *temporal feature model (TFM)* that describes the evolution of a feature model at different points in time. A central concept of a TFM is the *temporal element* that constitutes the basis for storing a feature model timeline along with a *temporal validity* that defines a time interval in which a temporal element is valid. Each element of the feature model specializes the temporal element and thus allows for specifying its temporal validity. Based on the literature, the authors propose a set of basic user-level operators for feature models, such as *Create Feature* or *Change Group Type* of features. The re-planning of a feature model evolution plan and thus an application of respective operators may lead to violations of its structural consistency, specified by means of well-formedness rules of the TFM (e.g., a feature group must consist of at least two features or feature names must be unique at each point in time). To preserve the consistency, the authors reason on the impact of evolution operations and model these as *structural operational semantics (SOS)* rules that describe pre-conditions along with a transition from one state to another if and only if all pre-conditions are satisfied. By analyzing the entire evolution timeline of a TFM, the proposed approach additionally detects semantical inconsistencies (e.g., dead or false optional features) and provides an explanation for the time the inconsistency was introduced while also identifying the causing evolution operators [173]. To enable evolution planning for artifacts other than feature models, the authors propose to augment existing metamodels with the concept of *temporal elements* to store and plan evolution similar to a TFM [171]. Finally, changes in the feature model or mappings between features and their realization may invalidate previously valid configurations. To this end, the authors propose an approach for guiding the evolution of configurations while preserving the behavior of a product [174, 175]. For example, in cases where features are merged, a configuration is computed that maintains the product behavior by automatically transitioning it to a configuration with the same set of artifacts based on mappings. The authors formalize a set of configuration update operations, such as the *Replace* operation to update the mapping between a feature and its artifacts. Domain engineers can use these operations to define

guidance upon which application engineers are able to automatically update configurations. All described concepts and methods are implemented in the tool suite `DarwinSPL` [170].
The research is closely related to the unified approach. A significant similarity is the support for the consistent evolution of variable systems by means of automated consistency preservation. Contrary, inconsistencies are handled that are caused or repaired in the problem space (i.e., TFM and configurations), while the unified approach addresses inconsistencies that are caused or repaired in the solution space (i.e., between heterogeneous artifacts of a product as well as between different products). Thus, this research can be considered as complementary to ours. Considering concepts and operations dealing with variability in space and time, the unified approach builds on the described research since the tool `DarwinSPL` is considered in the unification. For example, a `temporal validity` represents a `System Revision` while the operation *get Copy Of Valid Model* of `DarwinSPL` represents the *Externalize Domain* operation. To this end, the described research provides feature model specific concepts and operations whereas the unified approach builds on a unified basis consisting of both `System Revisions` and `Feature Revisions`.

Research by Feichtinger et al. [65] relates to the handling of variability-related Inconsistency Type 5 (*product to feature model consistency*). Based on a static code analysis and feature-to-artifact mappings, the proposed approach lifts dependencies to feature level. The computed feature dependencies are aggregated to create a dependency evolution matrix that summarizes the relationships between features. Represented as links between features in the feature model, dependencies are visualized to developers to support fixing potential inconsistencies. The approach proposed by the authors and the unified approach share the same pragmatics of lifting dependencies between features found on implementation level to the domain abstraction level of the SPL for supporting a consistent evolution of the solution space and the problem space. In contrast, the unified approach analyzes dependencies between deltas (representing the entire product line) instead of performing a static code analysis of individual products. Moreover, the unified approach also analyzes dependencies between features in the feature model (using SAT) and thus gains knowledge about existing dependencies in the problem space which allows for automatically repairing inconsistencies by adding the missing constraints, respectively. However, the unified approach only performs a dependency analysis between deltas based on references to determine requiring or excluding deltas, while the static code analysis also considers control and data dependencies. The combination of such analyses of the solution space with the analysis of the unified approach of the problem space along with automated repair can be considered as complementary.

Unified Variability Management

Seidl et al. [220, 216] research integrated management of variability in space and time based on delta modeling. To be able to apply their approach to different types of artifacts, support for automatically deriving delta languages for metamodels is provided. The authors point out that, while delta modeling can be used to realize both variability in space and time by adding, modifying or removing fragments, the purpose of delta modules can differ. A *configuration delta module* changes functionality of a product by enabling or disabling functionality associated with a feature while maintaining the identity of modified artifacts. An *evolution delta module* updates a feature in order to meet new requirements or fix defects. Modifying identifiers or refactorings (that are explicitly not supposed to change the functionality) are thus considered evolution operations. Furthermore, the authors propose *Hyper Feature Models (HFMs)* [218] that extend regular feature models by `Feature Revisions` as an explicit construct. DeltaEcore [219] realizes the described integrated management and can be used in conjunction with HFMs. Since this tool was considered in the unification, the unified approach builds on its concepts and additionally employs `System Revisions` and the respective *enables* relation between `Feature Revisions` and `System Revisions`. Identically to DeltaEcore, `Constraints` in the unified conceptual model can be defined on `Feature Options` (i.e., `Features` and `Feature Revisions`). As the authors state, a new `Feature Revision` should only be created in case the solution space of the respective feature changes (e.g., due to bug fixes) while in case there are no effects on possible combinations with revisions of other features, changes to one feature do not necessarily have to yield a new `Feature Revision`. Changes of a feature in the problem space, such as the modification of an optional feature to a mandatory one, is explicitly not considered a `Feature Revision` but an evolution of the feature model (i.e., to be captured by a `System Revision`). This consideration is very similar to the semantics of both revision types in this thesis. Likewise, changes in the problem space only yield a new `System Revision` that represents a new revision of the domain. If changes are performed on the solution space and internalized via *Internalizes Changes*, only the feature(s) specified in the manually provided input expression (comprising feature or feature interactions affected by the performed changes) obtain a new `Feature Revision`. However, a new `Feature Revision` is always created without considering the semantics of a change. While the approach proposed by the authors expects the user to directly edit the `Unified System` (e.g., manually create `Delta Modules`, `Feature Revisions` and `Mappings` between them), the unified approach is fully view-based and supports the evolution of a variable system exclusively via the modification of a particular product (see Section 8.3). This is an essential difference to this related work. Since the user is not supposed to provide delta modules and mappings manually but instead performs changes on a product which are recorded and mappings are computed fully automatically, no delta languages are needed. This also makes the use of application conditions between delta modules and a topological sorting, as performed in DeltaEcore during product externalization, superfluous, since the internalization order of changes (and thus the created mappings) reflects the order in which they must be applied during externalization. Thus, externalizing a product is achieved with less effort, demonstrating the synergy between variability management and view-based development. While the unified approach currently does not differentiate between evolution and configuration delta modules, this could be enabled by further refining `Delta Modules` as `Fragments` into both delta module types and providing more sophisticated diffing techniques and heuristics to determine the pragmatics of changes.

Lity et al. [143, 142] propose *higher-order delta modeling*, a formalism and extension to delta modeling for supporting the evolution of a delta-oriented SPL. Just as deltas transform models, higher-order deltas transform deltas. Thereby, a delta model (that specifies an entire SPL at one point in time) can be evolved to its new revision. Thus, higher-order deltas represent evolution steps describing the differences between system revisions of the SPL. A higher-order delta model encompasses higher-order deltas and represents the entire evolution history of an SPL. As a consequence, higher-order deltas enable a change impact analysis of the evolution of products. Both the unified approach and higher-order delta modeling support the integrated and explicit modeling of variability in space and time. Also, the unified approach employs delta modeling for evolving and deriving products and thus managing both variability dimensions by the same means. In contrast, higher-order deltas are used to evolve an SPL (thus capturing the differences between `System Revisions`) and not its individual features (and its `Feature`

Revisions). Therefore, the authors mention as future work the integration of higher-order deltas with hyper feature modeling [218] (which is essentially the problem space of DeltaEcore and thus supports Feature Revisions). Since DeltaEcore was considered in the unification, the unified conceptual model would also be well suited for combination with higher-order deltas by using them as refinement of the Fragment concept. Consequently, an option of the unified approach is to not only use deltas as Fragments but also in addition higher-order deltas as Fragments.

Schulze [210] proposes an approach for automatically transitioning from variants of software components (created via clone-and-own) to a reusable software platform. The approach is tailored to the automotive domain and supports compliance with norms such as CMMI [184] and ISO26262 [97]. The performed research proposes sophisticated similarity analyses to identify different types of clones (e.g., *exact clones* that comprise identical fragments or *semantic clones* that behave identically but differ with respect to their structure). Similarity analyses are performed on the architecture, interfaces, test cases and behavior specification for extracting a software product line, and are part of the proposed *Similarity Analysis (SimA)* framework. The research integrates into an existing tool landscape and employs an industrial tool for variability management. While this research is similar to work proposed by Rubin et al. [200, 198] and Mahmood et al. [152] as described above, it also comprises several commonalities to the unified approach. The main commonality of both approaches constitutes the development paradigm: both approaches promote reactive development based on product views. Thus, evolving a variable system benefits from the convenience of clone-and-own while employing a reusable platform common for SPLE to systematically manage products. The main difference of both approaches lies in their different pragmatics (i.e., transition from clone-and-own to a reusable platform vs. consistent evolution support for a system already employing a reusable platform) and thus in the different supported operations, processes and analyses. For example, this concerns the semantics of integrating changes into the variable system. While the unified approach supports the integration of changes via *Internalize Changes*, the approach proposed by Schulze follows an incremental reactive paradigm to enable an automated migration to a software product line. This essentially conforms to the product-oriented unified operation *Internalize Product* combined with feature localization. Moreover, the author utilizes an industrial tool for variability management that supports Feature Revisions, and additionally references an SVN repository that provides `System Revisions` for solution space artifacts such as Simulink models or code. Consequently, both approaches support variability in space and time while only the unified approach provides explicit constructs to relate instances of both revision types within a single tool.

View-Based Management and Heterogeneous Artifacts

Atkinson et al. [22] propose *Orthographic Software Modeling (OSM)*, an approach for view-based software development (see Section 2.6.1). OSM lays the foundation for several principles of the VITRUVIUS approach that is integrated with the unified approach to preserve consistency between heterogeneous artifacts (see Section 2.6.2). Among other concepts, the OSM approach proposes a *dimension-based view navigation*, a scheme for navigating along different perspectives of the system. A dimension represents a property of a system's description, such as its composition (e.g., the (de)composition of components into sub-components) or its abstraction level (e.g., the platform independent model (PIM) or implementation (e.g., Java)). The number of dimensions induce a multi-dimensional cube, while every dimension exposes several options. As a consequence, a cell in the multi-dimensional cube represents a view on the system. Moreover, and thus related to the research of this thesis, variants of the system are proposed as a further dimension. However, it can be argued that, to represent variability in space, a dimension for variants would not scale well and a dimension for features should be used. Furthermore, the authors do not consider variability in time. I propose to consider system revisions as an additional dimension and feature revisions as yet another dimension. It should also be considered that a dimension in the cube for revisions would not be able to express branches and merges (i.e., a revision graph) but only a linear sequence of revisions. Finally, the authors propose a projective approach for the creation and management of views that are created on demand from the system, and that a developer might be allowed to add, rename, or delete elements from the system via views, for instance, to create new variants. This could be considered as pioneer work towards *projectional editing* [240] of product lines as performed by VarCS [141] (see Definition 2.2).

Finally, a plethora of work and research is performed outside the SPLE community in the area of checking and preserving consistency between heterogeneous artifacts of the solution space (not considering variability at all) [24, 55, 118, 150, 122, 242, 227, 226, 96]. Meanwhile, numerous approaches have been proposed for variability-related inconsistency detection and repair as describe in Chapter 7, also targeting consistency preservation of the solution space by propagating changes across products. However, existing approaches to preserve consistency between different types of artifacts are not utilized in SPLE research yet. With the research presented in this thesis, the gap between existing approaches explicitly proposed for consistency preservation between heterogeneous artifacts and variability-related inconsistencies that can occur in the solution space is bridged by embedding the consistency preserving mechanisms of Vitruvius in the evolution process of a variable system. Specifically, consistency preservation is performed i) during the externalization of a product such that all its artifact types are constructed consistently, and ii) whenever an artifact type of the product is modified by the developer, changes are propagated to other dependent artifact types of that product.

15. Conclusion

This thesis presented contributions to enable consistent management of systems that are subject to variability in space and time and composed of heterogeneous artifacts. This chapter concludes the thesis starting with presenting the contributions and reflecting the insights. Section 15.1 comprises a summarized answer to every research question. Section 15.2 presents the relevance of the contributions for practitioners. Finally, future work that builds upon the contributions closes the chapter in Section 15.3.

15.1. Summary

In the following, results are summarized with respect to the questions (see Section 1.2).

The first research question focused on the unification of existing approaches that cope with variability in space, time, and both to provide a common foundation. Based on the sub-questions, the results are summarized.

RQ 1.1: Unified Concepts of Variability in Space and Time
Systems evolve rapidly and exist in many variations to address different requirements, leading to subsequent revisions (variability in time) and concurrent product variants (variability in space). During the last years, the unification of variability in space and time has gained momentum. While managing both variability dimensions has been considered in both SPLE and SCM research areas, the isolation of both engineering disciplines has led to a plethora of research, approaches and tools [220, 219, 214, 170, 141, 126, 140, 172]. Thus, various concepts exist to deal with variability in space and time, hampering the understanding of concepts and impeding the design of novel approaches to unify variability in space and time. Starting from discussions at a dedicated Dagstuhl seminar [34], the initial conceptual model was conceived that documented concepts of both variability dimensions from both

disciplines, yet did not unify them [9]. Thus, the initial conceptual model was systematically refined based on ten elicited tools into the *unified conceptual model* that appropriately unifies existing research while closing identified gaps in state of the art [7, 5]. The unified conceptual model is the first contribution of this thesis (see Chapter 5). Additionally, ten well-formedness rules are proposed that specify the static semantics of the conceptual model. In the evaluation, the model's granularity and coverage of concepts and relations were evaluated with respect to the selected tools, and two refinement processes of the unified conceptual model were demonstrated to show its applicability. The evaluation results showed that the model appropriately covers all concepts and relations of the considered tools. It can play a descriptive role by describing state of the art concepts and relations for dealing with variability in space, time and both, and can also be used to build novel approaches to cope with variability in space and time, specifically for explicitly dealing with `Feature Revisions` and `System Revisions` simultaneously (as none of the considered tools currently support). In sum, the conceptual model fills a gap that has been the focus of recent research. It increases the understanding of concepts for variability in space and time, enables scoping and comparing research, and provides guidance to design novel approaches for managing both variability dimensions.

RQ 1.2: Unified Operations for Variability in Space and Time
Providing a conceptual base for coping with both variability dimensions does not suffice to support the evolution of a variable system. Tools for coping with variability in space, time, or both employ operations following different paradigms, modalities and even propose opposing pre-conditions. Thus, a common understanding of operations is still missing for establishing a body of knowledge on unified management of variability in space and time. Following the same unification process as for the unified conceptual model, a diverse set of contemporary tools was analyzed while expert surveys were conducted to ensure that current, detailed, and reliable information on a tool's functionality was obtained. The goal was to understand the respective tools' operations, their inputs, outputs, pre and post-conditions and semantics for supporting either or both variability dimensions, and conceive operations for coping with variability in space and time based on the abstract concepts of the unified conceptual model. *Edit modalities* (i.e., editing the `Unified System` *directly* or via *views*) and *development paradigms* (i.e., platform-oriented or product-oriented) were identified as useful means to compare tools. Based on the insights and following the design principle of separation of concerns (i.e., one

operation per concern) *unified operations* were conceived that constitute the second contribution of this thesis (see Chapter 6) [6]. The unified operations comprise 21 direct editing operations (one *add, update, delete* operation per concept of the unified conceptual model), seven view-based operations and four predicates used as pre and post-conditions of the view-based operations. Analogously to the evaluation of the unified conceptual model, the granularity and coverage of the unified operations was evaluated. Specifically, whether an operation addresses more than one concern or one concern only partially (affecting the granularity of the unified operation regarding the studied tools), and whether the operations do not address concerns of considered tools or addressed concerns are not covered by any tool (affecting the coverage of the unified operations). Additionally, their application was demonstrated based on variability scenarios from literature. The evaluation results showed that the unified operations cover all concerns of the considered tools and can also be used to realize the scenarios while explicitly supporting the management of `Feature Revisions` and `System Revisions` simultaneously. None of the considered tools employ all of the unified operations nor support both revision types. The advantages and shortcomings of the unified view-based operations, such as the high degree of automation and their limited capabilities for editing `Mappings`, were discussed. Finally, open challenges were identified that encompass the combination of i) edit modalities and ii) development paradigms. Building on the unified conceptual model, the unified operations provide further means to support researchers and practitioners in managing both variability dimensions, to scope and compare their work as well as to analyze the compatibility of tools based on the inputs, outputs, and semantics of operations.

RQ 1.3: Quantification of Unification

Evaluating the appropriateness of an abstraction for a diverse set of tools is difficult as the abstraction is supposed to describe a common mental model that suits the mental model encoded in each analyzed tool. In this research, the appropriateness of a unification is quantified based on the *granularity* and *coverage* of the elements of an abstraction with respect to the elements of the considered tools. Specifically, elements of the abstraction should i) neither be unnecessarily fine-grained (i.e., too specific) nor unnecessarily coarse-grained (i.e., too generic), and ii) cover all elements of considered tools while not comprising unnecessary elements (i.e., not employed by any tool). Guizzardi et al. [90] research means to assess the granularity and coverage of an abstraction. The authors introduce the properties *laconic*, *lucid*, *complete*, and *sound* to evaluate the appropriateness of modeling languages. Although the same properties are used in this work, their framework is augmented in several ways and metrics for quantifying the appropriateness of a unification are proposed. The metrics for unification constitute the third contribution of this thesis (see Section 10.2) [7, 5]. They were applied to quantify the appropriateness of the unified conceptual model and the unified operations with respect to the selected set of tools. Additionally, their application is explained and illustrated in Section 11.5 by computing the metrics for two exemplary tools that refine the unified conceptual model. The proposed metrics for unification are envisioned to be useful in different application areas, such as for evaluating taxonomies in software engineering [108] or for the assessment of data integration and consolidation approaches [106].

RQ 2.1: Variability-Related Inconsistency Types
Providing concepts and operations based on state of the art for coping with both variability dimensions does not suffice to holistically support the evolution of a variable system. During development and maintenance of a variable system composed of heterogeneous artifacts, inconsistencies can easily be introduced. In contrast to single-product systems, variable systems are subject to a plethora of inconsistencies that vary for the problem space, the solution space, and when both spaces are involved. Thus, notions of consistency in SPLE are manifold. To obtain a body of knowledge regarding variability-related inconsistency types, their causes, effects and possible repair options and organize the research landscape in this field, the results of a literature survey have been generalized, mapped to a classification schema, and gaps in the schema have been filled. A classification of variability-related inconsistency types is proposed according to the problem space and the solution space, and whether the cause or repair of an inconsistency is performed in a product or in the variable system. In total, six *product-level inconsistency types* (i.e., caused or repaired in a product), six *system-level inconsistency types* (i.e., caused or repaired in the variable system), and two *cross-system inconsistency types* were identified, constituting the fourth contribution of this thesis (see Chapter 7). The main observations revealed a varying amount of research regarding the problem space and the solution space. While several approaches propose explanations of inconsistencies or even fully-automated repair options, this particularly addresses the problem space. Consistency preservation involving the solution space is addressed significantly less, especially when it comes to heterogeneous artifacts and distributed development.

RQ 2.2: Augmentation of Unified Operations with Consistency
Building on the gained knowledge of concepts and operations coping with

variability in space and time along with variability-related inconsistency types identified in state of the art, the unified approach was devised that constitutes the fifth contribution [3] (see Chapter 8). The unified conceptual model and unified operations were refined by means of supporting feature modeling and employing `deltas` as variability mechanism. The unified approach represents a VarCS (see Definition 2.2) and thus encourages specific modalities for evolving a variable system. Specifically, the employed variability mechanism is used internally and is hidden from the developer, eliminating the need to directly edit `Mappings` which is cognitively highly demanding [214, 141]. Thus, the unified approach allows to evolve the variable system based on product views that filter irrelevant details such as variable artifacts. As a consequence, it provides (unified) view-based operations that allow for a high degree of automation such as computing `Mappings` or creating `System Revisions` and `Feature Revisions` upon the internalization of changes. As a consequence, the unified approach encourages the development of a variable system conveniently by means of a clone-and-own strategy by evolving products, while it internally employs a reusable platform and a variability mechanism that is hidden from the developer. Moreover, for each product-level inconsistency type, the causing operation, the affected artifact, respective repair operations and the artifact in which the repair should be performed were identified. In addition to narrowing the field of focus to inconsistencies caused or repaired in a product, the unified approach offers consistency preservation to deal with a selected subset of inconsistencies that are caused or repaired in the solution space of a product. This comprised three inconsistency types: i) *feature model to product consistency* (i.e., in case the feature model is changed such that it allows for new products, the approach provides hints to the developer about new features or new valid feature combinations whose implementation might be missing upon the externalization of an affected product), ii) *product to feature model consistency* (i.e., in case a product's implementation is changed, the unified approach lifts new dependencies from the solution space to the feature model in the problem space by automatically adding the missing constraints), and iii) *product consistency* (the propagation of changes across heterogeneous artifacts in the solution space, such as Java or UML models, as well as across different products). Enabling *product consistency* leverages the consistency preserving mechanisms of the Vitruvius approach, leading to the final research question.

RQ 2.3: Vitruvius for Variability-Aware Consistency Preservation
A plethora of work and research is performed outside the SPLE community in

the area of checking and preserving consistency between heterogeneous artifacts of the solution space (not considering variability at all) while numerous approaches have been proposed for variability-related inconsistency detection and repair (see Chapter 7). However, existing approaches to preserve consistency between different types of solution space artifacts are hardly considered in SPLE research yet. To this end, Vitruvius is an approach for view-based development that supports (semi-) automated consistency preservation between heterogeneous types of artifacts based on manually predefined consistency preservation rules between two metamodels [116]. With this research, the gap between consistency preservation across heterogeneous artifacts and variability-related inconsistencies is bridged that can occur in the solution space: Consistency preserving mechanisms of Vitruvius are embedded in the unified approach, enabling *product consistency*: during the externalization of a product, its solution space artifacts (e.g., Java and UML models) are constructed consistently by propagating changes across the dependent artifact models, and ii) whenever an artifact model of the product is modified by a developer, changes are propagated to other dependent artifact models of that product and added to the unified system to allow for propagation to other products. In turn, the unified approach extends Vitruvius with concepts of the problem space to enable unified variability management.
The unified approach was implemented as VaVe 2.0 tool and evaluated based on two real-world case study systems ArgoUML-SPL und MobileMedia to gather evidence on its functional suitability (see Chapter 13). Since the ArgoUML-SPL has not been co-evolved and remained in its inital revision, while the original ArgoUML kept evolving, the ArgoUML-SPL was retroactively evolved by manually replaying the original changes from the ArgoUML. The publicly provided data set constitutes the sixth contribution of this thesis. The evaluation results showed that the unified approach is capable of detecting and repairing variability-related solution space inconsistency types. As a consequence, the evaluation provides evidence for the synergy between view-based development of variable systems, as encouraged by VarCS, and automated view-based consistency preservation of variability-related inconsistencies. To sum up, the unified approach offers consistency preservation to deal with inconsistency types that are either caused or repaired in the solution space during view-based evolution of systems comprised of heterogeneous artifact types while uniformly coping with variability in space and time. Thus, it supports consistent unified management of variable systems, which was the goal of this thesis.

15.2. Relevance for Industrial Applications

The research presented in this thesis has relevance for many application areas and practices of software engineering, considering software is becoming continuously prevalent in almost all areas of life.

Areas of Application. Besides its relevance for variable software systems, consistency-aware view-based management of systems coping with variability in space and time composed of heterogeneous artifacts is considered in several areas of application in industry. In the following, selected application areas are presented. On the one hand, it can be highly useful in automotive systems engineering that has to deal with a high diversity of possible car variants that, in addition, exist in various generations and comprise heterogeneous artifact types describing different domains (i.e., mechanical engineering, electrical engineering and software engineering). The complexity of automotive systems is rapidly increasing, revealing a growing need for reusability and traceability of existing functionality across the entire product landscape [38, 180]. On the other hand, the presented research is also relevant in industrial plant engineering that customizes and constructs a plant's software from reusable modules such as standard machines and automation components including heterogeneous artifact types describing different domains [8, 4, 68, 222, 67]. Finally, the conducted research has relevance in machine manufacturing for configuration of machinery [74].

Software Over The Air Updates. The *digital twin* is a term frequently used in the context of digitization and represents a research area that is becoming increasingly important [166]. It was initially introduced as an *"integrated multi-physics, multi-scale, probabilistic simulation of a vehicle or system that uses the best available physical models, sensor updates, fleet history, etc., to mirror the life of its flying twin"* [221, p. 11]. Such a virtual environment that allows for realistic simulations is often used in the context of production and automotive industry [36]. For instance, a digital twin can describe a configured variant of an individual car before it is being built. Particularly in the context of *Software Over The Air (SOTA)* updates in the automotive industry [89], a digital twin could be used to assess the compatibility of a new feature revision (e.g., to eliminate a bug) to products comprising different revisions of features. Moreover, with the introduced regulations of

UNECE[1], providing evidence that a car can perform software updates safely and securely while keeping a revision history of all software components will be mandatory for all manufacturers. The unified conceptual model can be considered a potential starting point for the software update management by explicitly modeling system revisions, feature revisions and their relations.

Feature-Oriented Agile Software Development. Agile software development constitutes a software engineering practice based on iterative and incremental development. Practices commonly employed during agile development are, for example, DevOps, continuous integration, or continuous deployment. Augmenting changes performed on a product variant with the information which feature or feature interaction is affected allows for new future possibilities during agile development. For example, it could be used for access control such that developers can only modify features they are allowed to change, or to reject changes if they affect features that were not intended to be modified by the developer (i.e., the specified expression in the *Internalize Changes* operation deviates from the actually changed features).

15.3. Future Work

This thesis proposed an approach for unified consistent management of variable systems composed of heterogeneous artifacts. Beyond the scope of the thesis and based on the presented contributions, it opens up potential for future work that is summarized in this section for each main contribution.

Unified Conceptual Model. The unified conceptual model (see Chapter 5) has been designed based on a diverse set of tools from the SPLE and SCM engineering disciplines. As described in Section 11.9, potential future work on the unified conceptual model encompasses its application to a set of real-world case studies from engineering disciplines other than SPLE and SCM to identify limitations or shortcomings of the model. This would expand the application area of the unified conceptual model beyond the current state of the art in SPLE and SCM.

Unified Operations. Gained insights during the construction of the unified operations (see Chapter 6) allowed to identify open challenges in state of the

[1] `https://wiki.unece.org/pages/viewpage.action?pageId=60362218`

art that open up future research avenues, as described in Section 12.7. On the one hand, this constitutes the combination of development paradigms (i.e., platform-oriented and product-oriented development). While the goal would be to allow for arbitrary alternation between both paradigms to leverage benefits of both (which is currently not supported by any tool), product-oriented edits do not provide fine-grained mappings that are necessary for platform-oriented editing. Thus, additional techniques such as *feature location* would be required. Further future work constitutes the combination of edit modalities (i.e., direct and view-based editing). While the goal would be to combine both edit modalities to leverage benefits of both, the challenge is to provide an editable view that includes mappings, which need to be displayed differently for every type of artifact. While the combination of both edit paradigms is supported by VTS, it only supports text as artifact type.

Unified Approach. The unified approach (see Chapter 8) paves the way for future works as discussed in Section 13.8. Since it builds on the unified view-based operations, it inherits the limitation for editing mappings. As a consequence, the combination of edit modalities, as described above, also represents future work for the unified approach. Moreover, the unified approach is currently only applicable in a greenfield scenario where the underlying data structure is populated incrementally as the system evolves. Thus, it would be interesting to consider an automated migration of legacy systems for applying the approach to existing variable systems. Possible migration options comprise the replay of annotated version histories to automatically compute feature-to-fragment mappings [195], or reverse engineering a product line from a set of products [155, 128]. Moreover, in this research, the consistency preservation mechanisms of Vitruvius are used to preserve consistency fully automatically. However, user decisions may be necessary in case several valid repair options exist. Thus, future work comprises the consideration of user decisions to replay them during product externalization. Last but not least, the evaluation can be extended by applying the unified approach to additional real-world case studies comprising artifact types from other engineering disciplines.

Bibliography

[1] Frederik Ahlemann and Gerold Riempp. "RefMod^{PM}: A Conceptual Reference Model for Project Management Information Systems". In: *Wirtschaftsinformatik* 50.2 (2008), pp. 88–97. DOI: `10.1365/s11576-008-0028-y`.

[2] Mauricio Alférez, Roberto E. Lopez-Herrejon, Ana Moreira, Vasco Amaral, and Alexander Egyed. "Supporting Consistency Checking between Features and Software Product Line Use Scenarios". In: *International Conference on Top Productivity through Software Reuse (ICSR)*. Springer-Verlag, 2011, 20–35. DOI: `10.1007/978-3-642-21347-2_3`.

[3] Sofia Ananieva. "Consistent Management of Variability in Space and Time". In: *International Systems and Software Product Line Conference (SPLC) - Volume B*. ACM, 2021, 7–12. DOI: `10.1145/3461002.3473067`. URL: `https://doi.org/10.1145/3461002.3473067`.

[4] Sofia Ananieva, Erik Burger, and Christian Stier. "Model-Driven Consistency Preservation in AutomationML". In: *International Conference on Automation Science and Engineering, CASE*. IEEE, 2018, pp. 1536–1541. DOI: `10.1109/COASE.2018.8560343`.

[5] Sofia Ananieva, Sandra Greiner, Timo Kehrer, Jacob Krüger, Thomas Kühn, Lukas Linsbauer, Sten Grüner, Anne Koziolek, Henrik Lönn, S. Ramesh, and Ralf H. Reussner. "A conceptual model for unifying variability in space and time: Rationale, validation, and illustrative applications". In: *Empirical Software Engineering* 27.5 (2022), p. 101. DOI: `10.1007/s10664-021-10097-z`.

[6] Sofia Ananieva, Sandra Greiner, Jacob Krüger, Lukas Linsbauer, Sten Grüner, Timo Kehrer, Thomas Kühn, Christoph Seidl, and Ralf Reussner. "Unified Operations for Variability in Space and Time". In: *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2021, pp. 1–10. DOI: `10.1145/3510466.3510483`.

[7] Sofia Ananieva, Sandra Greiner, Thomas Kühn, Jacob Krüger, Lukas Linsbauer, Sten Grüner, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, Sebastian Krieter, Christoph Seidl, S. Ramesh, Ralf Reussner, and Bernhard Westfechtel. "A Conceptual Model for Unifying Variability in Space and Time". In: *International Conference on Systems and Software Product Line (SPLC) - Volume A*. ACM, 2020, pp. 1–12. DOI: 10.1145/3382025.3414955.

[8] Sofia Ananieva and Prerna Juhlin. "Integriertes Engineering". In: *return* 7.5 (2020), pp. 48–49. DOI: 10.1007/s41964-020-0322-y.

[9] Sofia Ananieva, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, S. Ramesh, Andreas Burger, Gabriele Taentzer, and Bernhard Westfechtel. "Towards a Conceptual Model for Unifying Variability in Space and Time". In: *International Systems and Software Product Line Conference (SPLC)*. ACM, 2019, pp. 44–48. DOI: 10.1145/3307630.3342412.

[10] Sofia Ananieva, Heiko Klare, Erik Burger, and Ralf Reussner. "Variants and Versions Management for Models with Integrated Consistency Preservation". In: *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2018, pp. 3–10. DOI: 10.1145/3168365.3168377.

[11] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, undefinedtefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. "Flexible Product Line Engineering with a Virtual Platform". In: *International Conference on Software Engineering (ICSE)*. ACM, 2014, 532–535. DOI: 10.1145/2591062.2591126.

[12] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013. DOI: 10.1007/978-3-642-37521-7.

[13] Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. "Model Superimposition in Software Product Lines". In: *International Conference on Theory and Practice of Model Transformations (ICMT)*. Springer, 2009, 4–19. DOI: 10.1007/978-3-642-02408-5_2.

[14] Sven Apel and Christian Kästner. "An Overview of Feature-Oriented Software Development". In: *Journal of Object Technology* 8.5 (2009), pp. 49–84. DOI: 10.5381/jot.2009.8.5.c5.

[15] Sven Apel, Christian Kästner, and Christian Lengauer. “FEATUREHOUSE: Language-independent, automated software composition”. In: *International Conference on Software Engineering (ICSE)*. IEEE, 2009, pp. 221–231. DOI: 10.1109/ICSE.2009.5070523.

[16] Sven Apel, Christian Kästner, and Christian Lengauer. “Language-Independent and Automated Software Composition: The Feature-House Experience”. In: *Transactions on Software Engineering* 39.1 (2013), pp. 63–79. DOI: 10.1109/TSE.2011.120.

[17] Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. “An Evolutionary Process for Product-Driven Updates of Feature Models”. In: *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. Association for Computing Machinery, 2018, 67–74. DOI: 10.1145/3168365.3168374.

[18] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. “Automated Repairing of Variability Models”. In: *International Systems and Software Product Line Conference (SPLC) - Volume A*. ACM, 2017, 9–18. DOI: 10.1145/3106195.3106206.

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

[20] Wesley Klewerton Guez Assunção and Silvia Regina Vergilio. “Feature Location for Software Product Line Migration: A Mapping Study”. In: *International Software Product Line Conference (SPLC): Companion Volume for Workshops, Demonstrations and Tools - Volume 2*. ACM, 2014, 52–59. DOI: 10.1145/2647908.2655967.

[21] Wesley K.G. Assunção, Silvia R. Vergilio, and Roberto E. Lopez-Herrejon. “Automatic extraction of product line architecture and feature models from UML class diagram variants”. In: *Information and Software Technology* 117 (2020). DOI: 10.1016/j.infsof.2019.106198.

[22] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. “Orthographic Software Modeling: A Practical Approach to View-Based Development”. In: *Evaluation of Novel Approaches to Software Engineering (ENASE)*. Ed. by Leszek A. Maciaszek, César González-Pérez, and Stefan Jablonski. Springe, 2010, pp. 206–219. DOI: 10.1007/978-3-642-14819-4_15.

[23] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. "Supporting View-Based Development through Orthographic Software Modeling". In: *Evaluation of Novel Approaches to Software Engineering (ENASE).* 2009, pp. 71–86. DOI: `10.5220/0001953200710086`.

[24] Francois Bancilhon and Nicolas Spyratos. "Update Semantics of Relational Views". In: *ACM Trans. Database Syst.* 6.4 (1981), 557–575. DOI: `10.1145/319628.319634`.

[25] Jorge Barreiros and Ana Moreira. "A Cover-Based Approach for Configuration Repair". In: *Proceedings of the 18th International Software Product Line Conference (SPLC) - Volume 1.* ACM, 2014, 157–166. DOI: `10.1145/2648511.2648528`.

[26] Rabih Bashroush, Muhammad Garba, Rick Rabiser, Iris Groher, and Goetz Botterweck. "CASE Tool Support for Variability Management in Software Product Lines". In: *ACM Computing Surveys* 50.1 (2017), 1–45. DOI: `10.1145/3034827`.

[27] Victor R. Basili, Gianluigi Caldiera, and Dieter H. Rombach. "The Goal Question Metric Approach". In: vol. I. John Wiley & Sons, 1994, pp. 528–532.

[28] Don Batory. "A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite". In: *Generative and Transformational Techniques in Software Engineering.* Springer, 2006, pp. 3–35. DOI: `10.1007/11877028_1`.

[29] Don Batory. "Feature Models, Grammars, and Propositional Formulas". In: *International Conference on Software Product Lines (SPLC).* Vol. 3714. Springer, 2005, pp. 7–20. DOI: `10.1007/11554844_3`.

[30] Maurice H. ter Beek, Klaus Schmid, and Holger Eichelberger. "Textual Variability Modeling Languages: An Overview and Considerations". In: *International Systems and Software Product Line Conference (SPLC).* ACM, 2019, 151–157. DOI: `10.1145/3307630.3342398`.

[31] Benjamin Behringer, Jochen Palz, and Thorsten Berger. "PEoPL: Projectional Editing of Product Lines". In: *International Conference on Software Engineering (ICSE).* IEEE, 2017, pp. 563–574. DOI: `10.1109/ICSE.2017.58`.

[32] David Benavides, Antonio Ruiz-Cortés, and Pablo Trinidad. “Automated Reasoning on Feature Models”. In: *International Conference on Advanced Information Systems Engineering (CAISE)*. 2005, pp. 491–503. DOI: 10.1007/11431855_34.

[33] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. “Automated Analysis of Feature Models 20 Years later: A Literature Review”. In: *Information Systems* 35.6 (2010), pp. 615–636. DOI: 10.1016/j.is.2010.01.001.

[34] Thorsten Berger, Marsha Chechik, Timo Kehrer, and Manuel Wimmer. *Software Evolution in Time and Space: Unifying Version and Variability Management (Dagstuhl Seminar 19191)*. Ed. by Thorsten Berger, Marsha Chechik, Timo Kehrer, and Manuel Wimmer. Dagstuhl Reports. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 1–30. DOI: 10.4230/DagRep.9.5.1.

[35] Danilo Beuche. “pure::variants”. In: *Systems and Software Variability Management - Concepts, Tools and Experiences*. Ed. by Rafael Capilla, Jan Bosch, and Kyo Chul Kang. Springer, 2013. DOI: 10.1007/978-3-642-36583-6_12.

[36] Florian Biesinger and Michael Weyrich. “The Facets of Digital Twins in Production and the Automotive Industry”. In: *International Conference on Mechatronics Technology (ICMT)*. 2019, pp. 1–6. DOI: 10.1109/ICMECT.2019.8932101.

[37] Big Lever Software Inc. *Gears: A Software Product Line Engineering Tool*. Website. Available online at https://biglever.com/solution/gears/; visited on March 30th, 2022. 2010.

[38] Damir Bilic, Daniel Sundmark, Wasif Afzal, Peter Wallin, Adnan Causevic, and Christoffer Amlinger. “Model-Based Product Line Engineering in an Industrial Automotive Context: An Exploratory Case Study”. In: *International Systems and Software Product Line Conference (SPLC) - Volume 2*. ACM, 2018, 56–63. DOI: 10.1145/3236405.3237200.

[39] Paul Maximilian Bittner, Alexander Schultheiß, Thomas Thüm, Timo Kehrer, Jeffrey M. Young, and Lukas Linsbauer. “Feature Trace Recording”. In: *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2021, 1007–1020. DOI: 10.1145/3468264.3468531.

[40] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. “A Theory of Software Product Line Refinement”. In: *Theor. Comput. Sci.* 455 (2012), 2–30. DOI: `10.1016/j.tcs.2012.01.031`.

[41] Jan Bosch. “Toward Compositional Software Product Lines”. In: *IEEE Software* 27.3 (2010), pp. 29–34. DOI: `10.1109/MS.2010.32`.

[42] Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt, eds. *On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages, Book resulting from the Intervale Workshop 1982.* Springer, 1984. ISBN: 978-3-540-90842-5.

[43] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. “A Feature-Based Survey of Model View Approaches”. In: *Software and Systems Modeling* 18.3 (2019), pp. 1931–1952. DOI: `10.1007/s10270-017-0622-9`.

[44] Thomas Buchmann and Bernhard Westfechtel. “Mapping feature models onto domain models: ensuring consistency of configured domain models”. In: *Software & Systems Modeling* 13 (2014), pp. 1495–1527. DOI: `10.1007/s10270-012-0305-5`.

[45] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley Professional, 2001. ISBN: 0-201-70332-7.

[46] Benoit Combemale et al. “A Hitchhiker’s Guide to Model-Driven Engineering for Data-Centric Systems”. In: *IEEE Software* 38.4 (2021), pp. 71–84. DOI: `10.1109/MS.2020.2995125`.

[47] Reidar Conradi and Bernhard Westfechtel. “Towards a Uniform Version Model for Software Configuration Management”. In: *Workshop on System Configuration Management.* ICSE ’97. Springer, 1997, 1–17. DOI: `10.5555/647176.716423`.

[48] Reidar Conradi and Bernhard Westfechtel. “Version Models for Software Configuration Management”. In: *ACM Computing Surveys* 30.2 (1998), pp. 232–282. DOI: `10.1145/280277.280280`.

[49] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. “Extracting Software Product Lines: A Case Study Using Conditional Compilation”. In: *European Conference on Software Maintenance and Reengineering*. 2011, pp. 191–200. DOI: `10.1109/CSMR.2011.25`.

[50] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000. ISBN: 978-0-201-30977-5. DOI: `10.1007/3-540-46020-9_38`.

[51] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. "Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches". In: *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2012. DOI: `10.1145/2110147.2110167`.

[52] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. "Formalizing Cardinality-Based Feature Models and Their Specialization". In: *Software Process: Improvement and Practice* 10 (2005), pp. 7–29.

[53] Krzysztof Czarnecki, Chang Hwan, Peter Kim, and KT Kalleberg. "Feature Models are Views on Ontologies". In: *International Software Product Line Conference (SPLC)*. IEEE, 2006, pp. 41–51. DOI: `10.1109/SPLINE.2006.1691576`.

[54] Krzysztof Czarnecki and Krzysztof Pietroszek. "Verifying Feature-Based Model Templates against Well-Formedness OCL Constraints". In: *International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 2006, 211–220. DOI: `10.1145/1173706.1173738`.

[55] Cristine. Dantas, Leonardo Gresta Paulino Murta, and Claudia Maria Lima Werner. "Consistent evolution of UML models by automatic detection of change traces". In: *International Workshop on Principles of Software Evolution (IWPSE)*. 2005, pp. 144–147. DOI: `10.1109/IWPSE.2005.10`.

[56] Andreas Demuth, Markus Riedl-Ehrenleitner, Alexander Nöhrer, Peter Hehenberger, Klaus Zeman, and Alexander Egyed. "DesignSpace: An Infrastructure for Multi-User/Multi-Tool Engineering". In: *Symposium on Applied Computing*. ACM, 2015, 1486–1491. DOI: `10.1145/2695664.2695697`.

[57] Deepak Dhungana, Paul Grünbacher, Rick Rabiser, and Thomas Neumayer. "Structuring the Modeling Space and Supporting Evolution in Software Product Line Engineering". In: *Systems and Software* 83.7 (2010), 1108–1122. DOI: `10.1016/j.jss.2010.02.018`.

[58] Deepak Dhungana, Thomas Neumayer, Paul Grünbacher, and Rick Rabiser. "Supporting Evolution in Model-Based Product Line Engineering". In: *International Software Product Line Conference (SPLC)*. IEEE, 2008, pp. 319–328. DOI: `10.1109/SPLC.2008.26`.

[59] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. “FEVER: Extracting Feature-Oriented Changes from Commits”. In: *Proceedings of the 13th International Conference on Mining Software Repositories.* ACM, 2016, 85–96. DOI: 10.1145/2901739.2901755.

[60] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. “Specifying Overlaps of Heterogeneous Models for Global Consistency Checking”. In: *Models in Software Engineering*. Ed. by Juergen Dingel and Arnor Solberg. Springer, 2011, pp. 165–179. ISBN: 978-3-642-21210-9.

[61] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. “An Exploratory Study of Cloning in Industrial Software Product Lines”. In: *European Conference on Software Maintenance and Reengineering (CSMR).* IEEE, 2013, pp. 25–34. DOI: 10.1109/CSMR.2013.13.

[62] Alexander Egyed, Klaus Zeman, Peter Hehenberger, and Andreas Demuth. “Maintaining Consistency across Engineering Artifacts”. In: *Computer* 51.02 (2018), pp. 28–35. DOI: 10.1109/MC.2018.1451666.

[63] Jacky Estublier. “Software Configuration Management: A Roadmap”. In: *Conference on the Future of Software Engineering*. FOSE. ACM, 2000. DOI: 10.1145/336512.336576.

[64] Hafiyyan Sayyid Fadhlillah, Kevin Feichtinger, Lisa Sonnleithner, Rick Rabiser, and Alois Zoitl. “Towards Heterogeneous Multi-Dimensional Variability Modeling in Cyber-Physical Production Systems”. In: *International Systems and Software Product Line Conference (SPLC) - Volume B*. New York, NY, USA: ACM, 2021, 123–129. DOI: 10.1145/3461002.3473941. URL: https://doi.org/10.1145/3461002.3473941.

[65] Kevin Feichtinger, Daniel Hinterreiter, Lukas Linsbauer, Herbert Prähofer, and Paul Grünbacher. “Guiding feature model evolution by lifting code-level dependencies”. In: *Journal of Computer Languages* 63 (2021). DOI: https://doi.org/10.1016/j.cola.2021.101034.

[66] Kevin Feichtinger, Johann Stöbich, Dario Romano, and Rick Rabiser. “TRAVART: An Approach for Transforming Variability Models”. In: *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 2021. DOI: 10.1145/3442391.3442400.

[67] Stefan Feldmann, Julia Fuchs, and Birgit Vogel-Heuser. “Modularity, variant and version management in plant automation - Future challenges and state of the art”. In: *International Design Conference (DESIGN).* May 2012, pp. 1689–1698. ISBN: 9789537738174.

[68] Stefan Feldmann, Sebastian J. I. Herzig, Konstantin Kernschmidt, Thomas Wolfenstetter, Daniel Kammerl, Ahsan Qamar, Udo Lindemann, Helmut Krcmar, Christiaan J. J. Paredis, and Birgit Vogel-Heuser. “Towards Effective Management of Inconsistencies in Model-Based Engineering of Automated Production Systems”. In: *IFAC-PapersOnLine* 48 (2015), pp. 916–923. DOI: `10.1016/j.ifacol.2015.06.200`.

[69] Wolfram Fenske, Thomas Thüm, and Gunter Saake. “A Taxonomy of Software Product Line Reengineering”. In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 2014, pp. 1–8. DOI: `10.1145/2556624.2556643`.

[70] Felype Ferreira, Paulo Borba, Gustavo Soares, and Rohit Gheyi. “Making Software Product Line Evolution Safer”. In: *Brazilian Symposium on Software Components, Architectures and Reuse.* IEEE, 2012, pp. 21–30. DOI: `10.1109/SBCARS.2012.18`.

[71] Eduardo Figueiredo, Nelio Cacho, Claudio Sant’Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. “Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability”. In: *International Conference on Software Engineering (ICSE).* ACM, 2008, pp. 261–270. DOI: `10.1145/1368088.1368124`.

[72] Anthony Finkelstein, Jeff Kramer, Bashar Nuseibeh, L. Finkelstein, and Michael Goedicke. “Viewpoints: A Framework for Integrating Multiple Perspectives in System Development”. In: *Software Engineering and Knowledge Engineering* 2.1 (1992), pp. 31–57. DOI: `10.1142/S0218194092000038`.

[73] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. “The ECCO Tool: Extraction and Composition for Clone-and-Own”. In: *International Conference on Software Engineering.* ICSE. IEEE, 2015, 665–668. DOI: `10.1109/ICSE.2015.218`.

[74] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, Alexander Egyed, and Rudolf Ramler. “Bridging the Gap between Software Variability and System Variant Management: Experiences from an Industrial Machinery Product Line”. In: *Euromicro Conference on Software Engineering and Advanced Applications.* 2015, pp. 402–409. DOI: `10.1109/SEAA.2015.57`.

[75] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. “Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants”. In: *International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2014, pp. 391–400. DOI: 10.1109/icsme.2014.61.

[76] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. “Combinators for Bi-Directional Tree Transformations: A Linguistic Approach to the View Update Problem”. In: *SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM, 2005, 233–246. DOI: 10.1145/1040305.1040325.

[77] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. “Variability in Software Systems—A Systematic Literature Review”. In: *Transactions on Software Engineering* 40.3 (2014). DOI: 10.1109/TSE.2013.56.

[78] Nadia Gámez and Lidia Fuentes. “Software Product Line Evolution with Cardinality-Based Feature Models”. In: *International Conference on Software Reuse (ICSR).* Ed. by Klaus Schmid. Vol. 6727. Springer, 2011, pp. 102–118. DOI: 10.1007/978-3-642-21347-2_9.

[79] Bernhard Ganter, Gerd Stumme, and Rudolf Wille, eds. *Formal Concept Analysis, Foundations and Applications.* Vol. 3626. Lecture Notes in Computer Science. Springer, 2005. DOI: 10.1007/978-3-540-31881-1.

[80] Bernhard Ganter and Rudolf Wille. *Formal Concept Anlaysis – Mathematical Foundations.* Springer, 1999. DOI: 10.1007/978-3-642-59830-2.

[81] Lea Gerling. “Automated Migration Support for Software Product Line Co-Evolution”. In: *International Conference on Software Engineering: Companion Proceeedings (ICSE).* Association for Computing Machinery, 2018, 456–457. DOI: 10.1145/3183440.3183441.

[82] Lea Gerling, Sandra Greiner, Kristof Meixner, and Gabriela Karoline Michelon. “International Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution)”. In: *International Systems and Software Product Line Conference (SPLC) - Volume A.* Association for Computing Machinery, 2021, p. 204.

[83] Rohit Gheyi, Tiago Massoni, and Paulo Borba. “Algebraic Laws for Feature Models”. In: *Journal of Universal Computer Science* 14.21 (2008), pp. 3574–3591.

[84] Thomas Goldschmidt. “View-based textual modelling”. PhD thesis. Karlsruhe Institute of Technology, 2011. DOI: 10.5445/KSP/1000022234.

[85] Hassan Gomaa and Michael Shin. “A Multiple-View Meta-modeling Approach for Variability Management in Software Product Lines”. In: *International Conference on Software Reuse(ICSR)*. 2004, pp. 274–285. DOI: 10.1007/978-3-540-27799-6_23.

[86] Jack Greenfield, Keith Short, and Steve Cook. “Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools”. In: vol. 3154. John Wiley & Sons, 2004. DOI: 10.5555/983189.

[87] Sandra Greiner and Bernhard Westfechtel. “On Preserving Variability Consistency in Multiple Models”. In: *VaMoS’21: 15th International Working Conference on Variability Modelling of Software-Intensive Systems, Virtual Event / Krems, Austria, February 9-11, 2021*. ACM, 2021, 7:1–7:10. DOI: 10.1145/3442391.3442399.

[88] Nicola Guarino, Giancarlo Guizzardi, and John Mylopoulos. “On the Philosophical Foundations of Conceptual Models”. In: *International Conference on Information Modelling and Knowledge*. Ed. by Ajantha Dahanayake, Janne Huiskonen, Yasushi Kiyoki, Bernhard Thalheim, Hannu Jaakkola, and Naofumi Yoshida. Vol. 321. IOS Press, 2019, pp. 1–15. DOI: 10.3233/FAIA200002.

[89] Houssem Guissouma, Axel Diewald, and Eric Sax. “A Generic System for Automotive Software Over the Air (SOTA) Updates Allowing Efficient Variant and Release Management”. In: *International Conference on Information Systems Architecture and Technology*. Vol. 852. Springer International Publishing, 2018, 78–89. DOI: 10.1007/978-3-319-99981-4_8.

[90] Giancarlo Guizzardi, Luís Ferreira Pires, and Marten van Sinderen. “An Ontology-Based Approach for Evaluating the Domain Appropriateness and Comprehensibility Appropriateness of Modeling Languages”. In: *International Conference on Model Driven Engineering Languages and Systems*. MODELS. Springer, 2005, 691–705. DOI: 10.1007/11557432_51.

[91] Jianmei Guo, Yinglin Wang, Pablo Trinidad, and David Benavides. “Consistency maintenance for evolving feature models”. In: *Expert Syst. Appl.* 39 (2012), pp. 4987–4998. DOI: 10.1016/j.eswa.2011.10.014.

[92] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. “Closing the gap between modelling and java”. In: *International Conference on Software Language Engineering*. Springer. 2009, pp. 374–383. DOI: 10.1007/978-3-642-12107-4_25.

[93] Wolfgang Heider, Rick Rabiser, and Paul Grünbacher. “Facilitating the Evolution of Products in Product Line Engineering by Capturing and Replaying Configuration Decisions”. In: *Int. J. Softw. Tools Technol. Transf.* 14.5 (2012), 613–630. DOI: 10.1007/s10009-012-0229-y.

[94] Marc Hentze, Tobias Pett, Thomas Thüm, and Ina Schaefer. “Hyper Explanations for Feature-Model Defect Analysis”. In: *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. Ed. by Paul Grünbacher, Christoph Seidl, Deepak Dhungana, and Helena Lovasz-Bukvova. ACM, 2021, 14:1–14:9. DOI: 10.1145/3442391.3442406.

[95] Thomas Hettel, Michael Lawley, and Kerry Raymond. “Model Synchronisation: Definitions for Round-Trip Engineering”. In: *International Conference on Theory and Practice of Model Transformations)*. Springer, 2008, 31–45. DOI: 10.1007/978-3-540-69927-9_3.

[96] Soichiro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. “Feature-Based Classification of Bidirectional Transformation Approaches”. In: *Software and Systems Modeling* 15 (Jan. 2015). DOI: 10.1007/s10270-014-0450-0.

[97] Martin Hillenbrand. “Funktionale Sicherheit nach ISO 26262 in der Konzeptphase der Entwicklung von Elektrik/Elektronik Architekturen von Fahrzeugen”. PhD thesis. 2012. DOI: 10.5445/KSP/1000025616.

[98] Daniel Hinterreiter, Michael Nieke, Lukas Linsbauer, Christoph Seidl, Herbert Prähofer, and Paul Grünbacher. “Harmonized Temporal Feature Modeling to Uniformly Perform, Track, Analyze, and Replay Software Product Line Evolution”. In: *International Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 2019, pp. 115–128. DOI: 10.1145/3357765.3359515.

[99] Daniel Hinterreiter, Herbert Prähofer, Lukas Linsbauer, Paul Grünbacher, Florian Reisinger, and Alexander Egyed. “Feature-Oriented Evolution of Automation Software Systems in Industrial Software Ecosystems”. In: *International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. IEEE, 2018, pp. 107–114. DOI: 10.1109/ETFA.2018.8502557.

[100] Adrian Hoff, Michael Nieke, Christoph Seidl, Eirik Halvard Sæther, Ida Sandberg Motzfeldt, Crystal Chang Din, Ingrid Chieh Yu, and Ina Schaefer. “Consistency-preserving evolution planning on feature models”. In: *International Systems and Software Product Line Conference (SPLC)*. Ed. by Roberto Erick Lopez-Herrejon. ACM, 2020, 8:1–8:12. DOI: 10.1145/3382025.3414964.

[101] Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes. “Software Product Line Engineering: A Practical Experience”. In: *International Systems and Software Product Line Conference (SPLC)*. ACM, 2019, 164–176. DOI: 10.1145/3336294.3336304.

[102] ISO/IEC/IEEE. “Systems and software engineering – Architecture description”. In: *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)* (Jan. 2011), pp. 1 –46. DOI: 10.1109/IEEESTD.2011.6129467.

[103] Markus Jahn, Rick Rabiser, Paul Grünbacher, Markus Löberbauer, Reinhard Wolfinger, and Hanspeter Mössenböck. “Supporting Model Maintenance in Component-based Product Lines”. In: *Joint Working Conference on Software Architecture and European Conference on Software Architecture*. 2012, pp. 21–30. DOI: 10.1109/WICSA-ECSA.212.10.

[104] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. “Maintaining Feature Traceability with Embedded Annotations”. In: *International Systems and Software Product Line Conference (SPLC)*. ACM, 2015, 61–70. DOI: 10.1145/2791060.2791107.

[105] Martin Fagereng Johansen, Franck Fleurey, Mathieu Acher, Philippe Collet, and Philippe Lahire. “Exploring the Synergies Between Feature Models and Ontologies”. In: *International Conference on Software Product Lines (SPLC)*. 2010, pp. 163–170. DOI: SPLC.

[106] Sven Jordan. “Co-evolving Digital Architecture Twins”. In: *European Conference on Software Architecture (ECSA)*. Vol. 2978. ACM, 2021.

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

[108] Angelika Kaplan, Thomas Kühn, Sebastian Hahner, Niko Benkler, Jan Keim, Dominik Fuchß, Sophia Corallo, and Robert Heinrich. "Introducing an Evaluation Method for Taxonomies". In: *International Conference on Evaluation and Assessment in Software Engineering (EASE)*. Association for Computing Machinery, 2022, 311–316. ISBN: 9781450396134. DOI: 10.1145/3530019.3535305.

[109] Randy H. Katz. "Toward a Unified Framework for Version Modeling in Engineering Databases". In: *Computing Surveys* 22.4 (1990), 375–409. DOI: 10.1145/98163.98172.

[110] Timo Kehrer, Thomas Thüm, Alexander Schultheiß, and Paul Maximilian Bittner. "Bridging the Gap Between Clone-and-Own and Software Product Lines". In: *International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 2021, pp. 21–25. DOI: 10.1109/ICSE-NIER52604.2021.00013.

[111] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-oriented programming". In: *European Conference on Object-Oriented Programming (ECOOP)*. Ed. by Mehmet Akşit and Satoshi Matsuoka. Springer, 1997, pp. 220–242. ISBN: 978-3-540-69127-3.

[112] Jörg Christian Kirchhof, Michael Nieke, Ina Schaefer, David Schmalzing, and Michael Schulze. "Variant and Product Line Co-Evolution". In: *Model-Based Engineering of Collaborative Embedded Systems: Extensions of the SPES Methodology*. Ed. by Wolfgang Böhm, Manfred Broy, Cornel Klein, Klaus Pohl, Bernhard Rumpe, and Sebastian Schröck. Springer, 2021, pp. 333–351. DOI: 10.1007/978-3-030-62136-0_18.

[113] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. "Systematic literature reviews in software engineering – A systematic literature review". In: *Information and Software Technology* 51.1 (2009), pp. 7–15. DOI: https://doi.org/10.1016/j.infsof.2008.09.009.

[114] Heiko Klare. "Building Transformation Networks for Consistent Evolution of Interrelated Models". PhD thesis. Karlsruhe Institute of Technology, Germany, 2021. URL: https://nbn-resolving.org/urn:nbn:de:101:1-2021080405072192372510.

[115] Heiko Klare. "Designing a Change-Driven Language for Model Consistency Repair Routines". Master's Thesis. Karlsruhe: Karlsruhe Institute of Technology (KIT), 2016. DOI: 10.5445/IR/1000080138.

[116] Heiko Klare, Max E. Kramer, Michael Langhammer, Dominik Werle, Erik Burger, and Ralf Reussner. “Enabling consistency in view-based system development – The Vitruvius approach”. In: *Journal of Systems and Software* 171 (2021). DOI: 10.1016/j.jss.2020.110815.

[117] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. “Is There a Mismatch between Real-World Feature Models and Product-Line Research?” In: *Joint Meeting on Foundations of Software Engineering*. ACM, 2017, 291–302. DOI: 10.1145/3106237.3106252.

[118] Dimitrios Kolovos, Richard Paige, and Fiona Polack. “Detecting and Repairing Inconsistencies across Heterogeneous Models”. In: *International Conference on Software Testing, Verification, and Validation*. 2008, pp. 356–364. DOI: 10.1109/ICST.2008.23.

[119] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. “DeltaJ 1.5: delta-oriented programming for Java 1.5”. In: *International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools (PPPJ)*. 2014, 63–74. DOI: 10.1145/2647508.2647512.

[120] Matthias Kowal, Sofia Ananieva, and Thomas Thüm. “Explaining Anomalies in Feature Models”. In: *International Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 2016, 132–143. DOI: 10.1145/2993236.2993248.

[121] Max E. Kramer. “Specification Languages for Preserving Consistency between Models of Different Languages”. PhD thesis. Karlsruhe Institute of Technology, Germany, 2017. ISBN: 978-3-7315-0784-0. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000069284.

[122] Roland Kretschmer, Djamel Eddine Khelladi, and Alexander Egyed. “An Automated and Instant Discovery of Concrete Repairs for Model Inconsistencies”. In: *International Conference on Software Engineering: Companion Proceeedings (ICSE)*. ACM, 2018, 298–299. DOI: 10.1145/3183440.3194979.

[123] Jacob Krüger. “Understanding the re-engineering of variant-rich systems : an empirical work on economics, knowledge, traceability, and practices”. PhD thesis. Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, 2021. DOI: 10.25673/39349.

[124] Christian Kröher, Lea Gerling, and Klaus Schmid. “Identifying the Intensity of Variability Changes in Software Product Line Evolution”. In: *International Systems and Software Product Line Conference (SPLC) - Volume 1*. ACM, 2018, 54–64. DOI: 10.1145/3233027.3233032.

[125] Charles W. Krueger. “Easing the Transition to Software Mass Customization”. In: Berlin, Heidelberg: Springer-Verlag, 2001, 282–293. ISBN: 3540436596.

[126] Jacob Krüger, Sofia Ananieva, Lea Gerling, and Eric Walkingshaw. “International Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution)”. In: *Conference on Systems and Software Product Line (SPLC): Volume A*. ACM, 2020. DOI: 10.1145/3382025.3414944.

[127] Jacob Krüger and Thorsten Berger. “An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse”. In: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2020, pp. 432–444. DOI: 10.1145/3368089.3409684.

[128] Jacob Krüger, Wolfram Fenske, Thomas Thüm, Dirk Aporius, Gunter Saake, and Thomas Leich. “Apo-Games: A Case Study for Reverse Engineering Variability from Cloned Java Variants”. In: *International Systems and Software Product Line Conference (SPLC) - Volume 1*. ACM, 2018, 251–256. DOI: 10.1145/3233027.3236403.

[129] Jacob Krüger, Wardah Mahmood, and Thorsten Berger. “Promote-pl: A Round-Trip Engineering Process Model for Adopting and Evolving Product Lines”. In: *International Systems and Software Product Line Conference (SPLC)*. ACM, 2020, 2:1–12. DOI: 10.1145/3382025.3414970.

[130] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. “FeatureIDE: A Tool Framework for Feature-Oriented Software Development”. In: *International Conference on Software Engineering (ICSE)*. IEEE, 2009, pp. 70–85. DOI: 10.1109/ICSE.2009.5070568.

[131] Elias Kuiter, Jacob Krüger, Sebastian Krieter, Thomas Leich, and Gunter Saake. “Getting Rid of Clone-and-Own: Moving to a Software Product Line for Temperature Monitoring”. In: *International Systems and Software Product Line Conference (SPLC) - Volume 1*. ACM, 2018, 179–189. DOI: 10.1145/3233027.3233050.

[132] Kim Lauenroth and Klaus Pohl. "Towards Automated Consistency Checks of Product Line Requirements Specifications". In: *International Conference on Automated Software Engineering*. ACM, 2007, 373–376. DOI: 10.1145/1321631.1321687.

[133] Duc Minh Le, Hyesun Lee, Kyo Chul Kang, and Lee Keun. "Validating Consistency between a Feature Model and Its Implementation". In: *International Conference on Software Reuse (ICSR)*. Vol. 7925. Springer, 2013, pp. 1–16. DOI: 10.1007/978-3-642-38977-1_1.

[134] M.M. Lehman. "Programs, life cycles, and laws of software evolution". In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076. DOI: 10.1109/PROC.1980.11805.

[135] Jörg Liebig, Christian Kästner, and Sven Apel. "Analyzing the discipline of preprocessor annotations in 30 million lines of c code". In: *International Conference on Aspect Oriented Software Development (AOSD)*. ACM, 2011, pp. 191–202. DOI: 10.1145/1960275.1960299.

[136] Mark H. Liffiton and Karem A. Sakallah. "Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints". In: *J. Autom. Reason.* 40.1 (2008), 1–33. DOI: 10.1007/s10817-007-9084-z.

[137] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. "A Classification of Variation Control Systems". In: *International Conference on Generative Programming: Concepts & Experience (GPCE)*. ACM, 2017, pp. 49–62. DOI: 10.1145/3136040.3136054.

[138] Lukas Linsbauer, Alexander Egyed, and Roberto Erick Lopez-Herrejon. "A Variability Aware Configuration Management and Revision Control Platform". In: *International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 803–806. DOI: 10.1145/2889160.2889262.

[139] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. "Variability Extraction and Modeling for Product Variants". In: *Software and Systems Modeling* 16.4 (2017), 1179–1199. DOI: 10.1007/s10270-015-0512-y.

[140] Lukas Linsbauer, Somayeh Malakuti, Andrey Sadovykh, and Felix Schwägerl. "International Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution)". In: *International Systems and Software Product Line Conference (SPLC)*. ACM, 2018. DOI: 10.1145/3233027.3241372.

[141] Lukas Linsbauer, Felix Schwägerl, Thorsten Berger, and Paul Grünbacher. “Concepts of variation control systems”. In: *Systems and Software* 171 (2021), pp. 1–25. DOI: `10.1016/j.jss.2020.110796`. URL: `https://doi.org/10.1016/j.jss.2020.110796`.

[142] Sascha Lity. “Model-Based Product-Line Regression Testing of Variants and Versions of Variants”. PhD thesis. Braunschweig University of Technology, Germany, 2020.

[143] Sascha Lity, Matthias Kowal, and Ina Schaefer. “Higher-Order Delta Modeling for Software Product Line Evolution”. In: *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2016, 39–48. DOI: `10.1145/3001867.3001872`. URL: `https://doi.org/10.1145/3001867.3001872`.

[144] Malte Lochau, Dennis Reuling, Johannes Bürdek, Timo Kehrer, Sascha Lity, Andy Schürr, and Udo Kelter. “Model-Based Round-Trip Engineering and Testing of Evolving Software Product Lines”. In: *Managed Software Evolution*. Ed. by Ralf Reussner, Michael Goedicke, Wilhelm Hasselbring, Birgit Vogel-Heuser, Jan Keim, and Lukas Märtin. Springer, 2019, pp. 141–173. DOI: `10.1007/978-3-030-13499-0_7`.

[145] Jon Loeliger and Matthew McCullough. *Version Control with Git*. O’Reilly, 2012. ISBN: 978-0-596-52012-0.

[146] R. E. Lopez-Herrejon and A. Egyed. “C2MV2: Consistency and Composition for Managing Variability in Multi-view Systems”. In: *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2011, pp. 347–350. DOI: `10.1109/CSMR.2011.49`.

[147] Roberto Lopez-Herrejon and Alexander Egyed. “Detecting Inconsistencies in Multi-View Models with Variability”. In: *European conference on Modelling Foundations and Applications*. ACM, 2010, pp. 217–232. DOI: `10.1007/978-3-642-13595-8_18`.

[148] Roberto Lopez-Herrejon, Alexander Egyed, Salvador Trujillo, Josune Sosa, and Maider Azanza. “Using Incremental Consistency Management for Conformance Checking in Feature-Oriented Model-Driven Engineering”. In: ACM, 2010, pp. 93–100.

[149] Roberto E. Lopez-Herrejon and Alexander Egyed. “Towards Fixing Inconsistencies in Models with Variability”. In: *International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS)*. ACM, 2012, 93–100. DOI: `10.1145/2110147.2110158`.

[150] Nuno Macedo, Tiago Jorge, and Alcino Cunha. “A Feature-Based Classification of Model Repair Approaches”. In: *Transactions on Software Engineering* 43.7 (2015), pp. 615–640. DOI: `10.1109/TSE.2016.2620145`.

[151] Stephen A. MacKay. “The State of the Art in Concurrent, Distributed Configuration Management”. In: *Software Configuration Management (ICSE)*. Ed. by Jacky Estublier. Vol. 1005. Springer, 1995, pp. 180–193. DOI: `10.1007/3-540-60578-9\_17`.

[152] Wardah Mahmood, Daniel Strüber, Thorsten Berger, Ralf Lämmel, and Mukelabai Mukelabai. “Seamless Variability Management With the Virtual Platform”. In: *International Conference on Software Engineering (ICSE)*. IEEE, 2021, 1658–1670. DOI: `10.1109/ICSE43902.2021.00147`.

[153] Jabier Martinez, Nicolas Ordoñez, Xhevahire Tërnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. “Feature Location Benchmark with ArgoUML SPL”. In: *International Systems and Software Product Line Conference (SPLC) - Volume 1*. Association for Computing Machinery, 2018, 257–263. DOI: `10.1145/3233027.3236402`.

[154] Jabier Martinez, Daniele Wolfart, Wesley K. G. Assunção, and Eduardo Figueiredo. “Insights on Software Product Line Extraction Processes: ArgoUML to ArgoUML-SPL Revisited”. In: *International Conference on Systems and Software Product Line (SPLC) - Volume A*. ACM, 2020, pp. 1–6. DOI: `10.1145/3382025.3414971`.

[155] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. “Bottom-Up Technologies for Reuse: Automated Extractive Adoption of Software Product Lines”. In: *International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 67–70. DOI: `10.1109/ICSE-C.2017.15`.

[156] Johannes Meier, Heiko Klare, Christian Tunjic, Colin Atkinson, Erik Burger, Ralf Reussner, and Andreas Winter. “Single Underlying Models for Projectional, Multi-View Environments”. In: *International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 2019, pp. 117–128. DOI: `10.5220/0007396401170128`.

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

[158] Willian D. F. Mendonça, Wesley K. G. Assunção, and Lukas Linsbauer. “Multi-Objective Optimization for Reverse Engineering of Apo-Games Feature Models”. In: *International Systems and Software Product Line Conference (SPLC) - Volume 1*. ACM, 2018, 279–283. DOI: `10.1145/3233027.3236397`.

[159] Gabriela K. Michelon, Bruno Sotto-Mayor, Jabier Martinez, Aitor Arrieta, Rui Abreu, and Wesley K. G. Assunção. “Spectrum-Based Feature Localization: A Case Study Using ArgoUML”. In: *International Systems and Software Product Line Conference (SPLC) - Volume A*. ACM, 2021, 126–130. DOI: `10.1145/3461001.3473065`.

[160] Gabriela Karoline Michelon, David Obermann, Lukas Linsbauer, Wesley Klewerton Guez Assunção, Paul Grünbacher, and Alexander Egyed. “Locating feature revisions in software systems evolving in space and time”. In: *International Systems and Software Product Line Conference (SPLC) - Volume A*. ACM, 2020, 14:1–14:11. DOI: `10.1145/3382025.3414954`.

[161] Leticia Montalvillo and Oscar Díaz. “Tuning GitHub for SPL Development: Branching Models & Repository Operations for Product Engineers”. In: *International Conference on Software Product Line (SPLC)*. ACM, 2015, pp. 111–120. DOI: `10.1145/2791060.2791083`.

[162] Mukelabai Mukelabai, Benjamin Behringer, Moritz Fey, Jochen Palz, Jacob Krüger, and Thorsten Berger. “Multi-View Editing of Software Product Lines with PEoPL”. In: *International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 81–84. DOI: `10.1145/3183440.3183499`.

[163] Stephan Murer and Bruno Bonati. *Managed Evolution: A Strategy for Very Large Information Systems*. Springer, 2011. DOI: `10.1007/978-3-642-37521-7`.

[164] Dirk Muthig and Colin Atkinson. “Model-Driven Product Line Architectures”. In: *International Conference on Software Product Lines (SPLC)*. Springer, 2002, pp. 110–129. DOI: `10.1007/3-540-45652-X_8`.

[165] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. “Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study”. In: *Transactions on Software Engineering* 41.8 (2015), pp. 820–841. DOI: `10.1109/TSE.2015.2415793`.

[166] Elisa Negri, Luca Fumagalli, and Marco Macchi. "A Review of the Roles of Digital Twin in CPS-based Production Systems". In: *Procedia Manufacturing* 11 (2017), pp. 939–948. DOI: `https://doi.org/10.1016/j.promfg.2017.07.198`.

[167] Damir Nešić, Jacob Krüger, Ștefan Stănciulescu, and Thorsten Berger. "Principles of Feature Modeling". In: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE*. ACM, 2019, pp. 62–73. DOI: `10.1145/3338906.3338974`.

[168] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulesza, and Paulo Borba. "Investigating the Safe Evolution of Software Product Lines". In: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2011, 33–42. ISBN: 9781450306898. DOI: `10.1145/2047862.2047869`.

[169] Michael Nieke. "Consistent Feature-Model Driven Software Product Line Evolution". PhD thesis. Braunschweig University of Technology, Germany, 2021. URL: `https://publikationsserver.tu-braunschweig.de/receive/dbbs\_mods\_00069399`.

[170] Michael Nieke, Gil Engel, and Christoph Seidl. "DarwinSPL: An Integrated Tool Suite for Modeling Evolving Context-Aware Software Product Lines". In: *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2017, pp. 92–99. DOI: `10.1145/3023956.3023962`.

[171] Michael Nieke, Adrian Hoff, and Christoph Seidl. "Automated Metamodel Augmentation for Seamless Model Evolution Tracking and Planning". In: *International Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 2019, 68–80. DOI: `10.1145/3357765.3359526`.

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

[173] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. “Anomaly Analyses for Feature-Model Evolution”. In: *International Conference on Generative Programming: Concepts and Experiences (GPCE).* ACM, 2018, 188–201. DOI: 10.1145/3278122.3278123.

[174] Michael Nieke, Gabriela Sampaio, Thomas Thüm, Christoph Seidl, Leopoldo Teixeira, and Ina Schaefer. “Guiding the evolution of product-line configurations”. In: *Software and Systems Modeling* 21.1 (2021), 225–247. DOI: doi.org/10.1007/s10270-021-00906-w.

[175] Michael Nieke, Gabriela Sampaio, Thomas Thüm, Christoph Seidl, Leopoldo Teixeira, and Ina Schaefer. “GuyDance: Guiding Configuration Updates for Product-Line Evolution”. In: *International Systems and Software Product Line Conference (SPLC) - Volume B.* ACM, 2020, pp. 56–64. DOI: 10.1145/3382026.3425769.

[176] Michael Nieke, Christoph Seidl, and Thomas Thüm. “Back to the Future: Avoiding Paradoxes in Feature-Model Evolution”. In: *International Systems and Software Product Line Conference (SPLC) - Volume 2.* ACM, 2018, 48–51. DOI: 10.1145/3236405.3237201.

[177] Nan Niu, Juha Savolainen, and Yijun Yu. “Variability Modeling for Product Line Viewpoints Integration”. In: *Computer Software and Applications Conference.* IEEE, 2010, pp. 337–346. DOI: 10.1109/COMPSAC.2010.41.

[178] Linda M. Northrop. “SEI’s Software Product Line Tenets”. In: *IEEE* 19.4 (2002), pp. 32–40. DOI: 10.1109/MS.2002.1020285.

[179] Camila Nunes, Alessandro Garcia, Carlos Lucena, and Jaejoon Lee. “History-Sensitive Heuristics for Recovery of Features in Code of Evolving Program Families”. In: *International Software Product Line Conference (SPLC) - Volume 1.* ACM, 2012, 136–145. DOI: 10.1145/2362536.2362556.

[180] Olesia Oliinyk, Kai Petersen, Manfred Schoelzke, Martin Becker, and Soeren Schneickert. “Structuring Automotive Product Lines and Feature Models: An Exploratory Study at Opel”. In: *Requirements Engineering* 22.1 (2017), 105–135. DOI: 10.1007/s00766-015-0237-z.

[181] Object Management Group (OMG). *Object Constraint Language.* Version 2.4. 2014. URL: http://www.omg.org/spec/OCL/2.4/.

[182] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. "Feature-Oriented Software Evolution". In: *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 2013, 1–8. DOI: 10.1145/2430502.2430526.

[183] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. "Coevolution of Variability Models and Related Software Artifacts A Fresh Look at Evolution Patterns in the Linux Kernel". In: *Empirical Software Engineering* 21.4 (2016), pp. 1744–1793. DOI: 10.1007/s10664-015-9364-x.

[184] Mark Paulk, William Curtis, Mary Beth Chrissis, and Charles Weber. *Capability Maturity Model for Software (Version 1.1).* Tech. rep. CMU/SEI-93-TR-024. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1993. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11955.

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

[186] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. "Synchronizing Software Variants with Variantsync". In: *International Systems and Software Product Line Conference (SPLC).* Association for Computing Machinery, 2016, 329–332. DOI: 10.1145/2934466.2962726.

[187] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. "SiPL – A Delta-Based Modeling Framework for Software Product Line Engineering". In: *International Conference on Automated Software Engineering (ASE).* IEEE, 852-857, ASE. DOI: 10.1109/ASE.2015.106.

[188] Christopher Pietsch, Udo Kelter, Timo Kehrer, and Christoph Seidl. "Formal Foundations for Analyzing and Refactoring Delta-Oriented Model-Based Software Product Lines". In: *International Systems and Software Product Line Conference (SPLC).* ACM, 2019, 207–217. DOI: 10.1145/3336294.3336299.

[189] Christopher Pietsch, Christoph Seidl, Michael Nieke, and Timo Kehrer. "Delta-Oriented Development of Model-Based Software Product Lines with DeltaEcore and SiPL: A Comparison". In: *Model Management and Analytics for Large Scale Systems.* Elsevier, 2020, pp. 167–201. DOI: 10.1016/B978-0-12-816649-9.00017-X.

[190] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion: Next Generation Open Source Version Control.* O'Reilly, 2008. ISBN: 0-596-51033-0.

[191] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, 2005. DOI: 10.1007/3-540-28901-1.

[192] Clément Quinton, Andreas Pleuss, Daniel Le Berre, Laurence Duchien, and Goetz Botterweck. "Consistency Checking for the Evolution of Cardinality-Based Feature Models". In: *International Software Product Line Conference (SPLC) - Volume 1.* ACM, 2014, 122–131. DOI: 10.1145/2648511.2648524.

[193] Clément Quinton, Daniel Romero, and Laurence Duchien. "Cardinality-Based Feature Models with Constraints: A Pragmatic Approach". In: *International Software Product Line Conference (SPLC).* ACM, 2013, 162–166. DOI: 10.1145/2491627.2491638.

[194] Daniela Rabiser, Herbert Prähofer, Paul Grünbacher, Michael Petruzelka, Klaus Eder, Florian Angerer, Mario Kromoser, and Andreas Grimmer. "Multi-Purpose, Multi-Level Feature Modeling of Large-Scale Industrial Software Systems". In: *Software and Systems Model* 17.3 (2018), 913–938. DOI: 10.1007/s10270-016-0564-7.

[195] Michael Ratzenböck, Paul Grünbacher, Wesley K. G. Assunçao, Alexander Egyed, and Lukas Linsbauer. "Refactoring Product Lines by Replaying Version Histories". In: *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 2022. DOI: 10.1145/3510466.3510484.

[196] Alexander Reder and Alexander Egyed. "Incremental Consistency Checking for Complex Design Rules and Larger Model Changes". In: *International Conference on Model Driven Engineering Languages and Systems (MODELS).* Ed. by Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson. Springer, 2012, pp. 202–218. DOI: 10.1007/978-3-642-33666-9_14.

[197] Jan Reineke, Christos Stergiou, and Stavros Tripakis. "Basic Problems in Multi-View Modeling". In: *Software and Systems Modeling* 18.3 (2019), pp. 1577–1611. DOI: 10.1007/s10270-017-0638-1.

[198] Julia Rubin and Marsha Chechik. "A Framework for Managing Cloned Product Variants". In: *International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1233–1236. DOI: 10.1109/ICSE.2013.6606686.

[199] Julia Rubin and Marsha Chechik. "A Survey of Feature Location Techniques". In: *Domain Engineering, Product Lines, Languages, and Conceptual Models*. Springer, 2013, pp. 29–58. DOI: 10.1007/978-3-642-36654-3_2.

[200] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. "Cloned Product Variants: From Ad-Hoc to Managed Software Product Lines". In: *Software Tools for Technology Transfer* (2015), pp. 627–646. DOI: 10.1007/s10009-014-0347-9.

[201] Nayan B. Ruparelia. "The History of Version Control". In: *Software Engineering Notes* 35.1 (2010), pp. 5–9. DOI: 10.1145/1668862.1668876.

[202] Alcemir Rodrigues Santos, Raphael Pereira de Oliveira, and Eduardo Santana de Almeida. "Strategies for Consistency Checking on Software Product Lines: A Mapping Study". In: *International Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM, 2015, 1–14. DOI: 10.1145/2745802.2745806.

[203] Tonny Kurniadi Satyananda, Danhyung Lee, and Sungwon Kang. "Formal Verification of Consistency between Feature Model and Software Architecture in Software Product Line". In: *International Conference on Software Engineering Advances (ICSEA)*. 2007, p. 10. DOI: 10.1109/ICSEA.2007.33.

[204] J. Savolainen and J. Kuusela. "Consistency management of product line requirements". In: *International Symposium on Requirements Engineering*. 2001, pp. 40–47. DOI: 10.1109/ISRE.2001.948542.

[205] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. "Delta-Oriented Programming of Software Product Lines". In: *International Conference on Software Product Lines (SPLC)*. Springer, 2010, 77–91. DOI: 10.1007/978-3-642-15579-6_6.

[206] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. "Software Diversity: State of the Art and Perspectives". In: *Software Tools for Technology Transfer* 14.5 (2012), 477–495. DOI: 10.1007/s10009-012-0253-y.

[207] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. "Generic Semantics of Feature Diagrams". In: *Computer Networks* 51.2 (2007), 456–479. DOI: 10.1016/j.comnet.2006.08.008.

[208] J. Schröpfer, F. Schwägerl, and B. Westfechtel. "Consistency Control for Model Versions in Evolving Model-Driven Software Product Lines". In: *International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 2019, pp. 268–277. DOI: 10.1109/MODELS-C.2019.00043.

[209] Reimar Schröter, Thomas Thüm, Norbert Siegmund, and Gunter Saake. "Automated Analysis of Dependent Feature Models". In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 2013. DOI: 10.1145/2430502.2430515.

[210] Christoph Schulze. "Agile Software-Produktlinienentwicklung im Kontext heterogener Projektlandschaften". PhD thesis. RWTH Aachen University, Germany, 2019. ISBN: 978-3-8440-6683-8.

[211] Sandro Schulze, Michael Schulze, Uwe Ryssel, and Christoph Seidl. "Aligning Coevolving Artifacts Between Software Product Lines and Products". In: *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2016, 9–16. DOI: 10.1145/2866614.2866616.

[212] Felix Schwägerl. "Version Control and Product Lines in Model-Driven Software Engineering". PhD thesis. University of Bayreuth, 2018.

[213] Felix Schwägerl, Thomas Buchmann, Sabrina Uhrig, and Bernhard Westfechtel. "Towards the Integration of Model-Driven Engineering, Software Product Line Engineering, and Software Configuration Management". In: *International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SCITEPRESS, 2015, 5–18. DOI: 10.5220/0005195000050018.

[214] Felix Schwägerl and Bernhard Westfechtel. "Integrated revision and variation control for evolving model-driven software product lines". In: *Software and Systems Modeling* 18.6 (2019), pp. 3373–3420. DOI: `10.1007/s10270-019-00722-3`.

[215] Felix Schwägerl and Bernhard Westfechtel. "SuperMod: Tool Support for Collaborative Filtered Model-driven Software Product Line Engineering". In: *International Conference on Automated Software Engineering (ASE)*. ACM, 2016, 822–827. DOI: `10.1145/2970276.2970288`.

[216] Christoph Seidl. "Integrated Management of Variability in Space and Time in Software Families". PhD thesis. Dresden University of Technology, Germany, 2017.

[217] Christoph Seidl, Florian Heidenreich, and Uwe Aßmann. "Co-evolution of models and feature mapping in software product lines". In: *International Software Product Line Conference (SPLC) - Volume 1*. ACM, 2012, pp. 76–85. DOI: `10.1145/2362536.2362550`.

[218] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. "Capturing Variability in Space and Time with Hyper Feature Models". In: *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2014, 1–8. DOI: `10.1145/2556624.2556625`.

[219] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. "DeltaEcore - A Model-Based Delta Language Generation Framework". In: *Modellierung*. Vol. P-225. LNI. GI, 2014, pp. 81–96.

[220] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. "Integrated Management of Variability in Space and Time in Software Families". In: *International Software Product Line Conference (SPLC)*. ACM, 2014, 22–31. DOI: `10.1145/2648511.2648514`.

[221] Mike Shafto, Mike Conroy, Rich Doyle, Ed Glaessgen, Chris Kemp, Jacqueline LeMoigne, and Lui Wang. "NASA Modeling, Simulation, Information Technology & Processing -TA11". In: *Procedia Manufacturing* (2010). National Aeronautics and Space Administration.

[222] Tze Ying Sim, Fang Li, and Birgit Vogel-Heuser. "Modules, version and variability management in automation engineering of machine and plant manufacturing". In: *International Conference on Emerging Technologies and Factory Automation, (ETFA)*. IEEE, 2008, pp. 46–49. DOI: `10.1109/ETFA.2008.4638369`.

[223] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.

[224] Thomas Stahl and Markus Völter. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006. ISBN: 9780470025703.

[225] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework.* 2nd ed. Addison-Wesley, 2009.

[226] Perdita Stevens. "Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions". In: *International Conference on Model Driven Engineering Languages and Systems (MODELS).* Springer, 2007, 1–15. DOI: 10.5555/2394101.2394103.

[227] Perdita Stevens. "Is Bidirectionality Important?" In: *European Conference on Modelling Foundations and Applications (ECMFA).* Springer, 2018, pp. 1–11. DOI: 10.1007/978-3-319-92997-2_1.

[228] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. "Concepts, Operations, and Feasibility of a Projection-Based Variation Control System". In: *International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2016, pp. 323–333. DOI: 10.1109/ICSME.2016.88.

[229] Ștefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. "Forked and Integrated Variants in an Open-Source Firmware Project". In: *International Conference on Software Maintenance and Evolution.* ICSME. IEEE, 2015, pp. 151–160. DOI: 10.1109/icsm.2015.7332461.

[230] Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. "Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems". In: *International Systems and Software Product Line Conference (SPLC).* ACM, 2019, 177–188. DOI: 10.1145/3336294.3336302.

[231] Mikael Svahnberg and Jan Bosch. "Evolution in software product lines: two cases". In: *Software Maintenance* 11 (1999), pp. 391–422. DOI: 10.5555/334928.334930.

[232] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. "A Taxonomy of Variability Realization Techniques". In: *Software: Practice and Experience* 35.8 (2005), pp. 705–754. DOI: 10.1002/spe.652.

[233] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. "Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem". In: ACM, 2011, 47–60. DOI: 10.1145/1966445.1966451.

[234] Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “Revealing and Repairing Configuration Inconsistencies in Large-Scale System Software”. In: *Software Tools for Technology Transfer* 14.5 (2012), 531–551. DOI: `10.1007/s10009-012-0225-2`.

[235] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. “Towards Efficient Analysis of Variation in Time and Space”. In: *International Software Product Line Conference (SPLC).* ACM, 2019, 57–64. DOI: `10.1145/3307630.3342414`.

[236] Thomas Thüm, Don S. Batory, and Christian Kästner. “Reasoning about edits to feature models”. In: *International Conference on Software Engineering (ICSE).* IEEE, 2009, pp. 254–264. DOI: `10.1109/ICSE.2009.5070526`.

[237] Michael Vierhauser, Deepak Dhungana, Wolfgang Heider, Rick Rabiser, and Alexander Egyed. “Tool Support for Incremental Consistency Checking on Variability Models”. In: *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS).* Universität Duisburg-Essen, 2010, pp. 171–174.

[238] Michael Vierhauser, Paul Grünbacher, Wolfgang Heider, Gerald Holl, and Daniela Lettner. “Applying a Consistency Checking Framework for Heterogeneous Models and Artifacts in Industrial Product Lines”. In: *International Conference on Model Driven Engineering Languages and Systems (MODELS).* Springer, 2012, 531–545. DOI: `10.1007/978-3-642-33666-9_34`.

[239] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, implementing and using domain-specific languages.* dslbook.org, 2013.

[240] Eric Walkingshaw and Klaus Ostermann. “Projectional Editing of Variational Software”. In: *International Conference on Generative Programming: Concepts and Experiences (GPCE).* ACM, 2014, pp. 29–38. DOI: `10.1145/2658761.2658766`.

[241] Bernhard Westfechtel, Bjørn P. Munch, and Reidar Conradi. “A Layered Architecture for Uniform Version Management”. In: *Transactions on Software Engineering* 27.12 (2001). DOI: `10.1109/32.988710`.

[242] Manuel Wimmer, Nathalie Moreno, and Antonio Vallecillo. “Viewpoint Co-evolution through Coarse-Grained Changes and Coupled Transformations”. In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS)*. Ed. by Carlo A. Furia and Sebastian Nanz. Springer, 2012, pp. 336–352. DOI: 10.1007/978-3-642-30561-0_23.

[243] Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig. “Assessing Merge Potential of Existing Engine Control Systems into a Product Line”. In: *International Workshop on Software Engineering for Automotive Systems*. ACM, 2006, 61–67. DOI: 10.1145/1138474.1138485.

[244] Sai Zhang and Michael D. Ernst. “Which configuration option should I change?” In: *International Conference on Software Engineering (ICSE)*. 2014, pp. 152–163. DOI: 10.1145/2568225.2568251.

[245] Shurui Zhou, Ştefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. “Identifying Features in Forks”. In: *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, 105–116. DOI: 10.1145/3180155.3180205.

The Karlsruhe Series on Software Design and Quality

Edited by Prof. Dr. Ralf Reussner

Developing large and long-living variable systems faces many challenges. Coping with different and changing requirements leads to concurrent product variants with different features (variability in space) and subsequent revisions (variability in time). Moreover, products consist of different artifacts, such as code or diagrams. Dependencies between interrelated artifacts within a product, across products and across their revisions can quickly lead to inconsistencies during evolution. Dealing with both variability dimensions uniformly while preserving consistency of interrelated artifacts is a difficult endeavor.

This work provides a classification and unification of concepts and operations for variability in space and time. Moreover, variability-related inconsistencies, their causes and repairs are identified. Finally, an approach is presented that builds upon the unification and leverages view-based consistency preservation for variable systems comprised of interrelated artifacts.

ISSN 1867-0067
ISBN 978-3-7315-1241-7

Gedruckt auf FSC-zertifiziertem Papier

ISBN 978-3-7315-1241-7
9 783731 512417