Version Control and Product Lines in Model-Driven Software Engineering

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von

Felix Schwägerl

aus Nürnberg

1. Gutachter: Prof. Dr. Bernhard Westfechtel
2. Gutachter: Prof. Dr. Sven Apel
3. Gutachter: Prof. Dr. Andy Schürr

Tag der Einreichung: 4. September 2017
Tag des Kolloquiums: 8. Februar 2018

To my family

Versionskontrolle und Produktlinien in der Modellgetriebenen Softwareentwicklung

Kurzfassung

Diese Arbeit behandelt Anforderungen, die formale Ausgestaltung, die Implementierung sowie Anwendungen eines konzeptionellen Rahmenwerks zur Integration der modellgetriebenen Softwareentwicklung (MGSE), der Versionskontrolle (VK) und der Softwareproduktlinienentwicklung (SPLE).

Die allermeisten Softwareprojekte sehen sich mit drei Phänomenen konfrontiert: Abstraktion, Evolution und Variabilität. Abstraktion wird durch Modelle erzielt. Diese beschreiben Softwaresysteme auf einer höheren konzeptionellen Ebene und erleichtern somit die Durchführung und Kommunikation von Entwurfsentscheidungen. MGSE hat ausführbare Modelle und letztendlich eine Reduktion handgeschriebenen Quelltexts zum Ziel. Softwareevolution wird von Versionskontrollsystemen unterstützt, die zum Zwecke der Organisation gemeinschaftlich entwickelter Software unverzichtbar geworden sind. SPLE erfreut sich als Ansatz zur organisierten Verwaltung von Variabilität zunehmender Beliebtheit. Dabei wird das Softwaresystem in so genannte Features dekomponiert. Diese sind wiederum Merkmale, anhand derer sich Mitglieder der Produktlinie unterscheiden.

Gewöhnlich werden Abstraktion, Evolution und Variabilität durch voneinander unabhängige Werkzeuge erzielt. Dies führt einerseits zu unerwünschten Kontextwechseln zwischen Modellierungswerkzeugen, Versionskontrollsystemen und Produktlinienunterstützung. Andererseits wird eine lose Werkzeugkombination nicht den offensichtlichen Überschneidungen zwischen den Disziplinen gerecht. Beispielsweise beschäftigten sich MGSE und SPLE gleichermaßen mit (Domänen- bzw. Feature-) Modellen. Auch verwalten VK und SPLE verschiedene Arten von Versionen, nämlich Revisionen und Varianten.

Im Vorfeld der Ausgestaltung des Rahmenwerks werden seine Anforderungen mit dem Stand der Forschung in den Schnittbereichen modellgetriebene Produktlinienentwicklung, Modellversionierung und Produktlinienversionierung abgeglichen. Die Eigenschaften der verglichenen Systeme werden bei der Identifikation und Auflösung von Entwurfsentscheidungen im Bezug auf das Rahmenwerk bewertet und berücksichtigt.

Den Kern des Rahmenwerks stellt eine Hybridarchitektur dar, die sich aus drei Dimensionen zusammensetzt: Einem Revisionsgraphen, der die Evolution der beiden anderen Dimensionen beschreibt, sowie einem Featuremodell, das die Variabilität des Domänenmodells verwaltet. Letzteres unterliegt also der Evolution genauso wie der Variabilität. Das Rahmenwerk setzt das so genannte gefilterte Edieren ein. Dabei werden die Inhalte des Software-Repositorys vom Endbenutzer nicht direkt, sondern in mehreren Iterationen in einem getrennt verwalteten Einzelvarianten-Arbeitsbereich (Workspace), bearbeitet. Eine Iteration beginnt mit dem Kommando Check-Out. Dieses verlangt eine Benutzerauswahl im Revisionsgraphen und darauf folgend die Konfiguration der gewünschten Workspace-Variante durch eine vollständige Auswahl im Featuremodell. Die Inhalte des Workspace können daraufhin beliebig bearbeitet werden, bevor die Iteration mit dem Kommando Commit abgeschlossen wird. Hierbei gibt der Benutzer eine so genannte Feature-Ambition

an. Diese entspricht einer Menge von Varianten, auf die sich die vollzogenen Änderungen beziehen. Die Inhalte des Repositorys werden automatisch unter Berücksichtigung der historischen und logischen Komponente der Änderung aktualisiert.

Die Ausgestaltung des Rahmenwerks erfolgt auf Basis eines der Literatur entnommenen Formalismus, des Uniform Version Model. Die komplementäre strukturelle Perspektive wird mit modellgetriebenen Mitteln beschrieben. Die Inhalte des Repositorys sind Instanzen mehrerer Metamodelle, die die spezifischen Anforderungen der jeweiligen Dimension berücksichtigen. Beispielsweise darf das Domänenmodell wie in gängigen Modellierungsprojekten aus mehreren miteinander verknüpften modell- oder textbasierten Ressourcen bestehen.

Das in dieser Arbeit entwickelte Bedienmodell zum gefilterten Edieren unterscheidet sich von herkömmlichen Ansätzen durch seine dynamische Natur. Beispielsweise ist es möglich, Features innerhalb ein und derselben Iteration zu definieren und zu realisieren. Auch die mit einer Änderung verknüpfte Feature-Ambition unterliegt der Evolution. Jedoch kann es durch dynamisches gefiltertes Edieren zu neuen Arten von Problemen kommen. Um diesen gerecht zu werden, werden mehrere Konsistenzbedingungen sowie konsistenzerhaltende Algorithmen für Check-Out und Commit definiert. Außerdem wird eine neue Operation namens Migration eingeführt, die wiederholte Check-Outs überflüssig macht und somit einen unaufdringlichen Arbeitsfluss entstehen lässt.

Die gemeinschaftliche Bearbeitung modellgetriebener Produktlinien wird durch eine verteilte Replikationsstrategie ermöglicht. Private Transaktionen, die von Check-Out und Commit umschlossen sind, werden hierzu um öffentliche Transaktionen ergänzt. Letztere werden mit den Operationen Pull und Push gestartet bzw. beendet. Aufgrund gleichzeitiger Bearbeitung können Konflikte auftreten; diese werden zunächst durch nicht-interaktives kontextfreies Drei-Wege-Modellverschmelzen aufgelöst.

Sowohl durch die Zusammenarbeit mehrerer Entwickler als auch durch die Kombination von optionalen Features kann die syntaktische Wohlgeformtheit der gewählten Produktversion beeinträchtigt werden. Im hier betrachteten Rahmenwerk werden solche Situationen mittels einer produktbasierten A-Posteriori-Analyse behandelt. Vor dem Export konfliktbehafteter Inhalte wird deren syntaktische Wohlgeformtheit zunächst durch vorgegebene Auflösungsentscheidungen wiederhergestellt. Im Rahmen der darauf folgenden Iteration hat der Benutzer die Möglichkeit, auch die semantische Korrektheit wiederherzustellen.

Mit dem Werkzeug SuperMod wird eine modellgetriebene Implementierung des Rahmenwerks, basierend auf der Entwicklungsumgebung Eclipse und dessen Modellierungsrahmenwerk EMF, beigetragen. Die Client-Server-Kommunikation, die gemeinschaftliche SPLE ermöglicht, ist mit Hilfe eines REST-basierten Webservice realisiert worden.

Um letztendlich Rückschlüsse auf die Eigenschaften des Rahmenwerks zuzulassen, wurde das Werkzeug in drei akademischen Fallstudien angewendet: zwei Lehrbuch-Produktlinien, darunter eine Graph-Bibliothek und ein Haustechnik-Automatisierungs-System, sowie ein Bootstrapping-Experiment, in dem SuperMod auf Basis einer domänenspezifischen Sprache neuentworfen wird. Wie daraus gewonnene Ergebnisse zeigen, lässt sich eine SPL mit deutlich weniger Aufwand als mit ungefiltertem Edieren erstellen; außerdem liegt der Bedienaufwand des dynamischen Ediermodells unter dem des statischen Gegenstücks.

Insgesamt zeichnet sich das Rahmenwerk durch mehrere praktische Vorzüge aus: einheitliche Versionierung, uneingeschränkte Variabilität, Werkzeugunabhängigkeit und eine

Reduktion der kognitiven Komplexität. Der Ansatz bietet ein unaufdringliches und automatisiertes Versionsmanagement und eignet sich daher für reaktiv adaptierte Softwareproduktlinien sowie für agile SPLE-Prozesse. Jedoch ergeben sich durch das gefilterte Ediermodell auch neue konzeptionelle Einschränkungen. Zum einen zwingt das Konzept der Feature-Ambitionen den Benutzer zu ungewohnt feingranularen Commits. Zum anderen kann das gefilterte Edieren Multivarianten-Architekturentscheidungen erschweren.

Insgesamt trägt diese Arbeit die erste integrierte und automatisierte Lösung zur Kombination von MGSE, SPLE und VK bei. Das konzeptionelle Rahmenwerk wurde vollständig implementiert. Die ersten Experimente vermögen einige wünschenswerte Eigenschaften zu bestätigen, motivieren gleichzeitig aber weitere Forschung an diesem Thema.

Abstract

This thesis addresses the requirements, the formal elaboration, the implementation, and applications of a conceptual framework for the integration of model-driven software engineering (MDSE), version control (VC), and software product line engineering (SPLE).

The majority of software engineering projects are faced with three phenomena: abstraction, evolution, and variability. Abstraction is achieved through models, which provide a higher-level description of a software system that facilitates the enforcement and the communication of design decisions. MDSE aims at making models executable, reducing the amount of manually written source code. Software evolution is addressed by version control systems, which have become indispensable for the organization of collaboratively developed software. Last, an increasingly popular approach dedicated to the organized management of variability is SPLE. Corresponding approaches assume a decomposition of the software system into features, by which different members of the product line are distinguished.

Traditionally, abstraction, evolution, and variability are addressed by independent tools. This, on the one hand, causes undesirable context switches between modeling tools, version control systems, and product line technology. On the other hand, a combination of mutually unaware tools ignores overlaps between the disciplines. For instance, both MDSE and SPLE deal with models—domain models and variability models, respectively. Likewise, VC and SPLE deal with the management of different kinds of versions—revisions and variants.

In advance to the elaboration of the integrating conceptual framework, its requirements are aligned with the current state of research in model-driven product line engineering, model version control, and product line version control. The properties of existing systems are taken into consideration during the exploration of the design choices and decisions.

The core of the framework is a hybrid architecture consisting of three dimensions: a revision graph that controls the evolution of the other two dimensions, and a feature model that manages the variability of the domain model, which is subject to both evolution and variability. The framework relies on filtered editing; repository contents are not modified directly by the designated user, but in a single-version workspace in several iterations. An iteration is begun by the operation check-out, which requests a selection in the revision graph, before the desired variant to be presented in the workspace is defined as a configuration of the feature model. Then, workspace contents may be modified arbitrarily. The operation commit concludes an iteration, requesting a so called feature ambition from the user. This corresponds to the definition of a set of variants to which the performed modifications are relevant. Repository contents are updated automatically such that they consider both the historical and the logical scope of the change.

For elaborating the conceptual framework, we instrumentalize an established theoretical formalism, the Uniform Version Model. The structural perspective is designed by model-driven utilities; repository contents conform to several metamodels that consider the specific requirements of its three dimensions. E.g., the domain model may comprise heterogeneous interconnected model or text-based resources as usual in realistic model-driven projects.

The filtered editing model contributed here distinguishes from related approaches by a high amount of dynamism. For instance, features can be defined and realized within the same iteration, and the logical scope of a change may evolve. Dynamic filtered editing may, however, cause new kinds of problems. To this end, several consistency constraints

and consistency-preserving definitions of the operations check-out and commit are formally provided. Moreover, a new operation, migrate, is introduced, which obviates the need for the majority of check-outs and therefore enables an unobtrusive workflow.

Collaborative editing of model-driven product lines is achieved by a distributed replication strategy. Private transactions, embraced by check-out and commit, are complemented by public transactions, started with the operation pull and finalized with push. Conflicts, which may occur due to concurrent modifications, are preliminarily resolved by means of a context-free non-interactive three-way model merging strategy.

Both collaboration and the combination of optional features may cause product well-formedness violations. In the here considered conceptual framework, these are resolved by an a-posteriori product-based analysis strategy. Before being exported to the workspace, default resolution strategies are applied to conflicting contents. To restore semantical correctness, the user may revise the effects in a subsequent iteration.

With SuperMod, a model-driven implementation of the conceptual approach is contributed. The tool relies on the development environment Eclipse and its modeling framework EMF. Client/server communication, which enables collaboration, has been realized as a REST-based web service.

In order to allow for conclusions referring to the underlying framework, the tool has been applied to three academic case studies: two textbook product lines, a Graph Library and a Home Automation System, as well as a bootstrapping experiment, where SuperMod is re-engineered based on a domain-specific language. Results indicate a lower development effort when compared to unfiltered editing; moreover, the amount of user interactions is lower than the one implied by the static counterpart.

Altogether, the framework offers benefits such as uniform versioning, unconstrained variability, tool independence, and reduced cognitive complexity. By offering lightweight and automated version management, the approach is suitable for reactive adoption paths and for agile SPLE processes. On the downside, the filtered editing model creates new conceptual limitations. First, the concept of feature ambition forces potential users into unusually fine-grained commits. Second, filtered editing reduces awareness of other variants, which potentially hampers multi-variant architectural decisions.

Overall, the thesis contributes the first integrated and automated solution to the combination of MDSE, SPLE, and VC. The conceptual framework has been fully implemented. Initial experiments confirm many desirable properties, but also motivate future research.

Contents

Part I

Introduction

The entire history of software engineering is that of the rise in levels of abstraction.

GRADY BOOCH (2003)

Chapter 1

Context and Contributions

Abstract

The initial chapter of this thesis outlines the scientific background – represented by tools that manage the abstraction, the evolution, and the variability of software engineering artifacts – and portrays three early visions – stepwise refinement, feature-driven development, and filtered editing – that have influenced the research presented here. The central achievement presented in this thesis constitutes the integration of model-driven software engineering, version control, and software product line engineering, in a single support tool. Here, we provide a summary of the conceptual and technical contributions as well as experimental results, reflecting the added value of the approach. Last, the role of previously published scientific papers and the structure of the whole thesis are explained.

Contents

1.1 Revolutions of Software Engineering

The way computers are used is evolving constantly, and so are the methodologies of computer programming revolving. Until the 1960s, computers were space-consuming and expensive machines capable of only a limited set of instructions. Executing a computer program involved encoding both the program and the input using a punch card and bringing them to a counter, where they were manually dispatched and queued. The program's execution took hours to days, which was a multiple of the time required for developing the executed program; see [Wir08] in the bibliography.

Evidently, hardware capacity has grown significantly since then, and so has the effort for computer programming. The *software crisis* [Dij72] revealed that, rather than instantaneously allowing for more efficient and more reliable software, the increasing hardware capacity lead to a deterioration of the quality of software. "Our limitations in designing complex systems [were] no longer determined by slow hardware, but by our own intellectual capacity" [Wir08, p. 32].

In response to this observation, the term *software engineering* was coined during a NATO-sponsored conference in 1968 [BBH69]. The attendees agreed that software should no longer be created as by-product by the electrical engineers who design the hardware; instead, programs should be carefully designed on the basis of well-established formalisms, i.e., *abstractions* that better match the problem domain. Furthermore, well-defined *software development processes* should manage the increasing size of both software projects and development teams.

1.1.1 Increasing the Abstraction of Software Development

The history of software engineering can be told as the story of creating and mastering gradually increasing levels of abstraction. Conversely, the *level of detail* decreased, the higher the level of abstraction was becoming. Figure 1.1 recapitulates four milestones.

Computer processors run programs encoded in a binary format, which is difficult to comprehend and write for human beings. *Assembly languages* [PH08], which have been developed at the end of the 1940s, provide a more readable syntax for instructions, which are transformed into machine code by *assemblers*.

High-level programming languages soon appeared as abstractions for assembly languages, providing features such as *structured programming* and removing the one-to-one correspondence between instructions written by the programmer and instructions executed by the

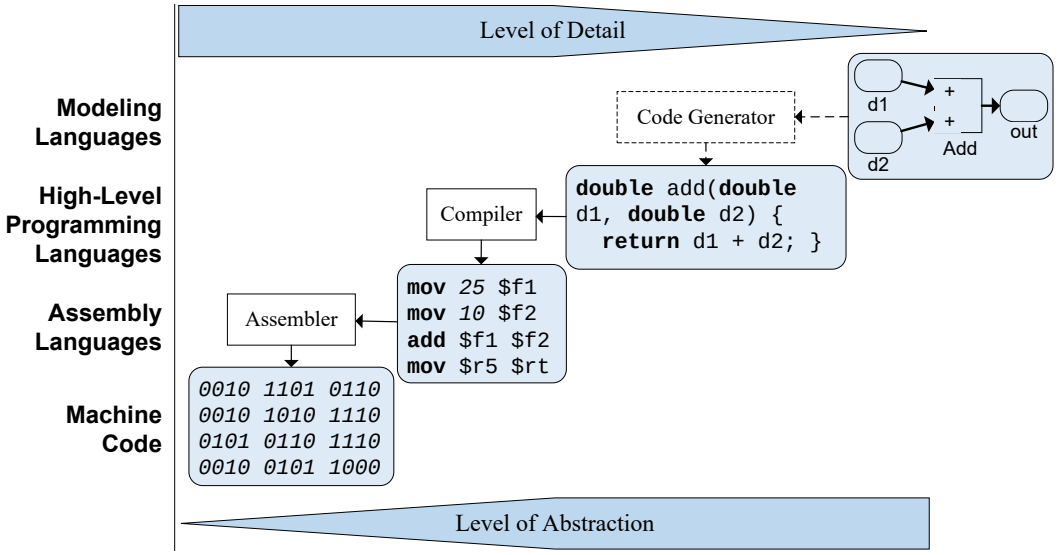


Figure 1.1: Levels of abstraction from machine code to modeling languages.

machine. High-level programs can be transformed into behaviorally equivalent assembly programs running on different platforms by *compilers* [Aho+06]. The advent of *object-oriented programming* (*OOP*) during the 1990s marks a prominent milestone of high-level programming. An *object* encapsulates the state of a part of the executed program that may be modified using *operations*. OOP languages such as *Java* [Gos+15] have become the preferred choice for the development of desktop, remote, and mobile applications. Albeit, it has soon turned out that OOP by itself was "not abstract enough".

From the beginning of the era of programming, computer scientists invented formalisms helping them reflect about the problem to be solved. Prominent examples include *EBNF* (the *Extended Backus-Naur Form*) [Aho+06] for the description of the syntax of *formal languages* as well as *flow diagrams* for the informal definition of algorithms. Nowadays, there exist a multitude of *modeling languages* having a well-defined syntax and more or less formally defined semantics. For instance, the model depicted in Figure 1.1 conforms to the modeling language *MATLAB Simulink* [Beu06], which is frequently used in the domain of numerical simulation.

1.1.2 From the Waterfall Model to Agile Software Development

As another result of the 1968 NATO software engineering conference [BBH69], it has been realized that software development is more than just programming, but implies systematically collecting requirements from the later users, carefully designing the software, and dedicated quality management. This lead to the definition of *core processes* [Som06], namely *requirements engineering*, *analysis*, *design*, *implementation*, and *validation*. Orthogonal to core processes are *support processes* such as *project management*, *deployment*, and particularly, *software configuration management*.

The *waterfall model*, initially described in [Roy70], is the simplest form of combining all core processes in a strictly sequential, i.e., *phase-structured* way. Each phase produces

an *artifact* that is used as input for the subsequent phase. While informal descriptions are suitable for *requirements specification*, *conceptual model*, and *architecture*, the only artifact actually required to run the final application is *code*.

The waterfall model was soon extended by *iterations* allowing to repeat a phase or to step back into previous phases. Redesign and modification of the waterfall model gave birth to a multitude of *plan-driven development processes* such as the *Rational Unified Process* (*RUP*) [Kru03], which enabled the development of software in a large scale.

In the 1990s, a revolution in the field of software development processes was triggered by the growing documentation overhead caused by plan-driven processes, which became increasingly heavyweight. *Agile development processes* such as *Extreme Programming* (*XP*) [BA04] and *Scrum* [BS02] aim at reducing the planning effort to a minimum by giving individuals more responsibilities, while maintaining essential engineering practices. In the *Agile Manifesto*, pioneers delimit themselves from defenders of plan-driven software development by defining twelve *agile principles* including, among others, "continuous delivery", "preparation for changing requirements", and "customer collaboration" [Bec+01].

On the one hand, agile processes are designed for small development teams, which closely *collaborate* with each other. On the other hand, they enforce that software is created in an *incremental* way by gradually adding customer-relevant units of functionality.

1.2 Evolution vs. Variability

Two everyday-life phenomena, *evolution* and *variability* affect almost every kind of software that is being developed nowadays. Their difference is portrayed in an abstract way in Figure 1.2. Together with the *abstraction* gained from software models, it is these two terms that motivate the contributions to be presented in this thesis.

Software developers are humans and therefore make mistakes. Furthermore, it is almost impossible to foresee changing internal or external requirements, or new functionalities demanded by users, at the time of software deployment. These premises make software *evolution* ineluctable. Most approaches covering this phenomenon assume that different historical states of the evolving artifact are arranged in a time-line being made up of an ordered sequence of *revisions* that mutually supersede each other [Cha09; Key07].

In contrast, *variability* denotes the intentional co-existence of different software *variants* at the same point in time [AK09]. These variants typically differ from each other in terms

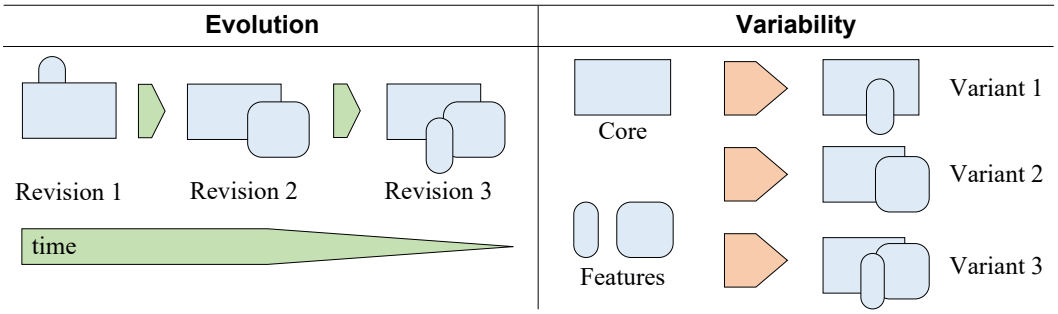


Figure 1.2: Evolution and variability of software.

of the configuration of internal properties – for instance, the underlying operating system – or of user-visible *units of functionality* [BSR04]. In [Kan+90, p. 3], the term *feature* is introduced for "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system [...]". Different co-existing *variants* of a software application distinguish with respect to their individual selection of features.

At first glance, evolution and variability management are orthogonal; in this thesis, we are particularly interested in situations where they overlap. The term *version*, to this end, is a generalization for *revisions* and *variants*.

1.3 Relevant Software Engineering Sub-Disciplines

The aforementioned phenomena, evolution, abstraction, and variability are relevant to many software projects conducted nowadays. This thesis makes contributions to the three related sub-disciplines of software engineering, namely *model-driven software engineering*, *version control*, and *software product line engineering*. These are presented in detail and delineated from each other later; this section gives an introductory top-down overview. [1]

1.3.1 Model-Driven Software Engineering

Model-driven software engineering (MDSE) [Völ+06] is focused on the development of *models* as first-class artifacts in order to describe software systems at a higher level of abstraction (cf. Figure 1.1). In this way, MDSE promises to increase the productivity of software engineers, who may focus on creative and intellectually challenging modeling tasks rather than on repeated activities at source code level. Models are typically expressed in well-defined languages such as the *Unified Modeling Language* (UML) or in *domain-specific languages* (DSLs). These languages define the structure as well as the behavior of model elements. Different forms of notation, ranging between textual and graphical, exist.

A key element of *model-driven architecture* (MDA) [Mel+04] is *model transformations*, which transform, e.g., *platform-independent models* into *platform-specific models*, and the latter into executable source code, respectively. Therefore, the level of abstraction expressed by different models may differ significantly.

1.3.2 Version Control

Version control (VC) is a part of the support process *software configuration management* (SCM); it has become indispensable for software engineers to control software evolution and to coordinate software changes inside a team. *Version control systems* (VCS) such as *Subversion* [CFP04] follow an iterative three-stage editing model (cf. Figure 1.3):

1. A developer *checks out* a *revision* of a software project from a *repository*. A copy of the selected revision of the project is loaded into the private *workspace*.
2. In the workspace, the developer *modifies* the project implementing new functionality.

1 The section is based on the introductions pre-published in [Schwä+15; SBW15].

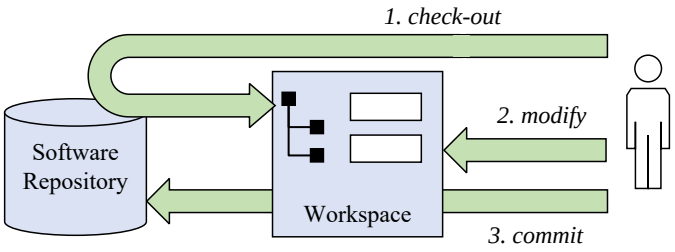


Figure 1.3: The iterative three-stage editing model offered by version control systems. Based on [SBW15, Figure 1].

3. To make these modifications visible for others, the developer *commits* his/her changes to the repository. A new *revision* is created transparently.

From a fine-grained view, version control enforces an *iterative* style of development. Within each iteration, the developer may focus on a change referring to a specific increment of the overall software without having to deal with earlier versions, details of which are transparent in the workspace. Furthermore, *collaboration* is enabled, e.g., by *locks* on currently modified resources, or by *merging* changes concurrently applied by several developers [CW98].

1.3.3 Software Product Line Engineering

Software Product Line Engineering (SPLE) aims at the systematic development of a family of software products by exploiting the *variability* among members thereof [CN01]. *Mass customization* is achieved by *organized reuse*, such that software development becomes more economic for both developers and users. Core assets of different products are provided as a *platform*. Commonalities and differences among products are captured in *variability models*, e.g., *feature models* [Kan+90].

In [PBL05], a two-stage SPLE process has been proposed (cf. Figure 1.4):

1. During *domain engineering* (*DE*), platform and variability model are defined. The platform contains *variation points*, representing architectural elements whose realization may diversify, and *variants*, referring to concrete realizations belonging to specific features. To connect the platform with the variability model, a *mapping* is established, e.g., in the form of *traceability links* attached to platform artifacts.
2. In the repeated activity *application engineering* (*AE*), variability is resolved, e.g., by specification of a *feature configuration*, and a *product* with the desired features is derived in a preferably automated way. In each variation point, the variant corresponding to the selected feature(s) is instantiated.

By moving repeated programming and design activities necessary for product development to DE, the effort required for AE may be reduced significantly. Though, having to anticipate variation points in the platform complicates SPLE. Furthermore, product derivation is usually a “one-way road”: After having derived a product, its connection to the platform gets lost; product-specific modifications must be propagated back to the platform manually in case they are relevant to the entire product line.

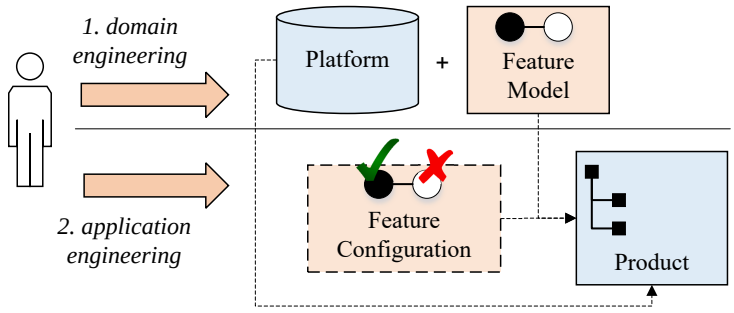


Figure 1.4: The established two-stage SPLE process. Based on [SBW15, Figure 2].

1.4 Early Visions

In this section, let us consider three visions shaped relatively early in software engineering history: *stepwise refinement*, *feature-driven software development*, and *filtered editing*. The principles and requirements discussed there have influenced the approach contributed in this thesis. The research prototype *SuperMod* presented in Chapter 2, provides tool support materializing these three visions. As demonstrated in this thesis, the tool allows to develop software in a *stepwise* and *feature-driven* way, relying on a *filtered editing* model.

1.4.1 Program Development by Stepwise Refinement

Way ahead of agile software development, Wirth [Wir71] gave rise to a programming paradigm relying on *stepwise refinement*. He suggested a highly iterative style of software development:

> In each step, one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition of refinement of specifications terminates when all instructions are expressed in terms on an underlying computer or programming language [.] [. . .]
>
> As tasks are refined, so the data may have to be refined, decomposed, or structured, and it is natural to refine program and data specifications in parallel. ([Wir71, p. 221])

According to these explanations, an *iteration* of program development by stepwise refinement consists of the following steps:

1. Select a program fragment (instructions and/or data specifications) to be decomposed.
2. Let the user realize the decomposition as an artifact in the target language.
3. During realization, define and use placeholders for fragments to be refined later.

The tools available at that time – essentially, the first generation of high-level programming languages and compilers – were, however, not able to adequately support Wirth's vision. On the one hand, there was no suitable representation for "some placeholder fragment to be refined later", which is nowadays provided by suitable abstraction mechanisms. On the other

hand, the management of software being developed in an iterative and would have required sophisticated *software configuration management*, for which convenient tool support was not provided back then.

Not at least due to the missing tool support, waterfall-like development processes had suppressed the idea of stepwise refinement until agile software development became popular. [Raj06] argues that the peculiarities of modern software projects, in particular volatility of requirements, make a paradigm shift from waterfall-oriented to iterative development processes indispensable. At the heart of the new paradigm should be *anticipation* of changing requirements and *prototyping*, which allows for early customer feedback.

1.4.2 Feature-Driven Software Development

Feature-oriented software development (FOSD) "is a paradigm for the construction, customization, and synthesis of large-scale software systems" [AK09, p. 49], aiming at structuring a software system along its *features*, i.e., its "increment[s] in program functionality" [Bat05, p. 7]. Thus, FOSD can be interpreted as a collection of tools and methods for SPLE.

Prior to FOSD, *feature-driven development* (*FDD*) [PF01] was introduced without explicitly referring to features as manifestation of variability, but rather, as mandatory increments of functionality. FDD is also based on an iterative and incremental style of software development; the whole repertoire of development activities (analysis, design, etc.) is repeated feature by feature rather than with a global scope. A development iteration is structured as follows:

1. Select a new feature to be realized.
2. Let the user realize the feature's software increment using the target language.
3. For documentation purposes, associate the program fragments affected by the performed change with the new feature.

As shown in subsequent chapters, this strictly iterative and incremental style of FDD has hardly been realized — one reason is the lack of a suitable variability management approach or tool that should provide adequate mechanisms to narrow down the context of an increment.

1.4.3 Filtered Editing of Multi-Variant Programs

The third vision to be presented here is *filtered editing* as provided by *multi-version editors* such as the *Multi-Version Personal Editor* (MVPE) [SBK88] or *P-Edit* [Kru84]. Here, it is assumed that specific parts of a source document to be edited is surrounded with annotations, e.g., *preprocessor directives* in *conditional compilation* scenarios.

To ease multi-variant editing, a *view* is created from the source based on a *read filter*—a complete selection of configuration options. Furthermore, a *write filter* – a partial selection of configuration options – is specified. After performing a modification in the filtered view, the multi-version source document is updated transparently, such that the changes are connected to the options selected in the write filter. An iteration of filtered editing may be summarized as follows:

1. Based upon the *read filter*, fade out parts of the software irrelevant for the change.

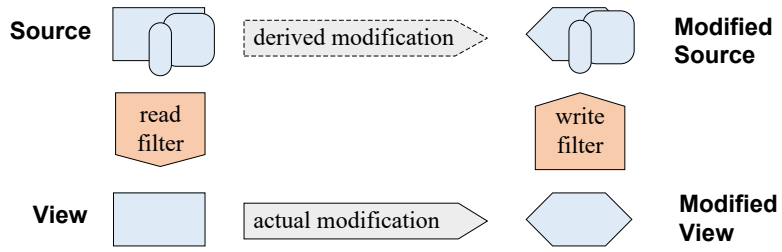


Figure 1.5: Principle and terminology of filtered editing.

2. Let the user perform a *modification* in the filtered view.
3. Based upon the *write filter*, apply a corresponding *derived* modification in the source transparently.

Figure 1.5 illustrates the relationship between source, view, read filter, and write filter.

Filtered editing is recently undergoing a revival in the context of *variation control systems* [WC09; WO14; Stă+16; LBG17], a new SPLE research branch.

1.4.4 The Quintessence

Stepwise refinement and FDD are built around the concepts of *iterations* and *increments*, respectively. Albeit, their notion is mutually different. In stepwise refinement, an increment is a mandatory part of a software system, implementing a placeholder defined in a previous iteration in the sense of continuous advancement of the software, i.e., *evolution*. Contrastingly, the FOSD interpretation of FDD [AK09] considers increments (i.e., features) as manifestation of *variability*. Furthermore, once the concept to be realized has been selected, the complexity of the second realization step may be drastically reduced by applying filtered editing. Abstracting from the peculiarities of the three visions, their *iterations* may be reconciled as follows:

1. Select some concept to be realized. Based on a read filter, fade out parts of the software that are irrelevant for the modification.
2. Let the user realize the concept as an *increment* to the existing software.
3. Manage the increment's integration into the software repository based on a write filter.

This observation forms a central argument of this thesis and is reflected by the three-stage *check-out/modify/commit* workflow, which is borrowed from version control systems and transferred to SPLE on top of a filtered editing model. Moreover, both stepwise refinement and feature-driven development rely on *abstraction* as an important principle for reducing complexity and for enabling placeholder objects. *Models* constitute a universal source of abstraction and are therefore assumed as primary artifacts in this thesis.

1.5 Key Contributions and Results

The approach elaborated in this thesis combines the three visions presented above by integrating the management of *historical* and *logical* versions, i.e., *evolution* and *variability*,

based on abstraction mechanisms provided by *models*. The key contributions and results presented throughout this thesis are anticipated in this section.

1.5.1 Conceptual Contributions

A *conceptual framework* for the integration of SPLE and VC based on MDSE is formally developed. The framework organizes the versioning of models and text files with respect to both their historical evolution and their variability.

At the heart of the framework is a *hybrid* repository architecture, consisting of a a *revision graph*, a *feature model* and a *domain model*. The revision graph manages the historical evolution of both the feature model and the domain model; the domain model is in addition versioned by the feature model. As theoretical foundation of the version space, the *Uniform Version Model* (*UVM*) [WMC01] has been selected. UVM is based on *set theory* and *propositional logic*. In this thesis, the mapping of high-level version models (revision graphs and feature models) to an extended form of UVM is contributed.

The product space, which consists of the domain model and the feature model, is managed in the repository transparently. We contribute metamodels and transformations for a multi-version *extrinsic* representation; in the workspace, the standard (*intrinsic*) single-version representation is still used for compatibility with external tools.

Rather than directly manipulating the repository, the user indirectly accesses it by filtered editing. To this end, UVM's static filtered editing model is advanced inasmuch as it supports the *dynamic* co-evolution of feature model and platform. This additional flexibility, however, is paid with a new class of consistency problems. These are addressed by evolution-aware consistency constraints and by precise definitions of the workspace operations CHECK-OUT and COMMIT, as well as of a new operation, MIGRATE, which prepares the current workspace content for the subsequent iteration.

In addition, conceptual requirements and challenges connected to *multi-user operation* are explored. To this end, the VC metaphors CHECKOUT and COMMIT are extended by PULL and PUSH, which synchronize different copies of a repository and thus provide a distributed architecture for collaborative versioning of model-driven software product lines.

Both variability and collaboration may cause violations of the syntactical well-formedness of the product version to be checked-out. This is addressed by an *a-posteriori product-based well-formedness analysis* strategy that relies on *default resolution actions*, which may be revised in the single-version workspace. To this end, we define individual *product conflict conditions* for the feature model and the domain model, respectively.

1.5.2 Technical Contributions

With *SuperMod*, a full implementation of the conceptual framework is provided. The tool has been built upon the integrated development environment *Eclipse*. The aforementioned "domain model" may comprise arbitrary non-model or model-based artifacts; in the latter case, the *Eclipse Modeling Framework* (*EMF*) is universally supported. Multi-user operation has been realized as a *web service* compliant with the architectural style *Representational State Transfer* (REST) [Fie00].

1.5.3 Experimental Validation and Results

In order to evaluate the properties of the conceptual framework, *SuperMod* has been applied to three case studies, whose set-up and outcome are described. The case studies include the well-known *Graph Library* standard example, a product line for *Home Automation Systems* (HAS), and a bootstrapping example where SuperMod's core is redesigned as a software product line based on a domain-specific modeling language. As far as quantitative results are concerned, the user effort saved as well as the manual overhead in terms of required editing steps in comparison with state-of-the-art tools are investigated.

Results suggest that the dynamic filtered editing model implies significant benefits over both unfiltered and static filtered editing. Furthermore, the effects of product well-formedness analysis go beyond the variant available in the workspace, which advances the state of the art in product-based product line well-formedness analysis.

A second goal of the evaluation was to collect initial experiences with respect to secondary properties of the approach. Regarding this, the case studies demonstrate that the approach enables collaborative SPLE, that it is compatible with both general purpose and domain-specific languages, and that it particularly supports a reactive style of SPLE. Moreover, both models and source code are supported at an adequately fine product granularity.

1.5.4 Benefits and Limitations

Based on both theoretical and experimental considerations, we may assign four central benefits to the approach:

First, historical and logical versioning are supported *uniformly*, while the user may tie on familiar version control and SPLE metaphors. Second, since the workspace representation relies on single-version views, *tool independence* is guaranteed. Third, variability manifested in the multi-variant domain model is *unconstrained* in the sense that every detail of a model may vary. Last, since the repository, containing multi-variant artifacts as well as version membership information, is hidden from the user, *cognitive complexity* is *reduced* when compared to state-of-the-art approaches.

Conversely, almost every scientific contribution implies conceptual limitations, and so does the approach at hand:

By forcing the user(s) into operating in a single-version view, the *awareness of other versions* is limited, which impedes explicit modeling decisions referring to the realization of variation points and variants. Another potential drawback is related to the definition of the concept *ambition*; during an iteration, all changes performed in the workspace are supposed to be connected to an *equal logical scope*, which does not necessarily reflect the user's intent. Moreover, the comparison-based versioning strategy implies a *limited precision of matching*, which may result in falsely detected differences at commit. Last, *product well-formedness control* is applied in a filtered way, which complicates the identification and resolution of collaboration-related conflicts not visible in the workspace.

1.5.5 Added Value Seen from Different Perspectives

For the end user, the added value of the conceptual framework and its implementation depends on his/her specific requirements. Therefore, the contributions must be regarded

from at least three different perspectives.

(Model-Driven) Software Engineering. SuperMod and its underlying conceptual framework contribute to the support discipline *software configuration management* by offering integrated historical and logical version control support. Although specifically designed for model-driven development, the approach is generic enough to harmonize with source code as well as configuration files.

Version Control. Most state-of-the-art version control systems are limited to historical versioning of text files, which are interpreted merely as sequences of lines. The approach at hand raises the level of abstraction of the versioned product. In addition, *logical* versioning allows for the combination of mutually independent changes in order to compose new variants during check-out. From SPLE, *feature models* have been imported as adequate abstraction for variability.

(Model-Driven) Software Product Line Engineering. By providing an iterative and incremental editing model, the conceptual framework is designed for a *dynamic* development style for (model-driven) SPL; feature model and platform may co-evolve. By following *filtered* editing, the distinction between domain engineering and application engineering is blurred intentionally, enabling *product-based product line development* (which can be summarized under the slogan: "perform application engineering, and get domain engineering for free"). Furthermore, *collaborative* (MD)SPLE is enabled.

Notice the optionality of the term "model-driven" above. Throughout the thesis, we consider text files as a special case of models, such that the approach is transferable to source code based scenarios natively.

Altogether, SuperMod can be considered both as a fully-fledged version control system and as a software product line management tool, relying on model-driven abstractions both for its implementation and for the artifacts it is applied to.

1.6 Thesis-Related Publications

The following scientific publications contain material partly reproduced in this thesis; references are provided in the corresponding sections. Notice that all publications listed here have been peer-reviewed and indexed by established computer science bibliographies such as *DBLP*[2]. The list is ordered by publication date. Bibliographic details are provided in the back matter (see page 403).

– The first version of the conceptual framework has been published in [Schwä+15], which also includes a tentative literature review with respect to related approaches. Furthermore, the *flow chart* example is introduced here; it is re-used in this thesis on several occasions.

– In [SBW15], the requirements and design decisions underlying the tool *SuperMod* are presented. Furthermore, a comparative analysis of the domains SPLE and VC is provided. Here, the running *Graph Library* example has been adapted first.

2 http://dblp.uni-trier.de/

– [Schwä+16] presents the model-driven realization of the conceptual framework and introduces several pieces of optimization both to the framework and to the tool. Also here, the *core metamodel* has been introduced.

– In [SBW16a], the filtered editing model underlying the conceptual framework is characterized and assessed with a focus on the process perspective. To this end, a larger case study, the *Home Automation System* (HAS) product line, is conducted and analyzed.

– [SW16a] extends the conceptual framework towards multi-user operation by providing collaborative revision graphs and the multi-user operations PULL and PUSH.

– SuperMod has been presented in a tool demonstration format in [SW16b]. The demonstration is complemented by a video. See Section 14.9 on page 322.

– The tool-centric survey published in [SW17c] contains excerpts of Chapter 6, where pairwise mutual integration of MDSE, SPLE, and VC is conceptually investigated.

– *Dynamic* consistency constraints and consistency-preserving algorithms, which are also a subject of Chapter 11, were originally published in [SW17b]. Also here, the operation MIGRATE, an important part of the dynamic filtered editing model, has been introduced. An extended version of the paper is being published in [SW17a].

– Among others, the topic of *product well-formedness analysis* and the *evaluation* of the overall approach are covered by [SW18], whose publication is still pending. The article reproduces selected excerpts of this thesis, including Chapters 13 and 15.

1.7 Outline

This section concludes the introduction by outlining the remainder of this thesis part by part.

Part I. The remainder of the first part consists of Chapter 2, where the central problem statement of this thesis is phrased and requirements for an integrated solution are collected. In advance, a top-down comparison with related approaches is given. Furthermore, we provide a fast-paced and tool-centric description of the contributed framework.

Part II. The second part of this thesis has two purposes. On the one hand, it presents the state of the art referring to the three software engineering sub-disciplines to which the thesis makes contributions. Chapters 3, 4, and 5 are dedicated to *model-driven software engineering*, *software configuration management* (with a focus on *version control*), and *software product line engineering*, respectively. On the other hand, the ambiguities implied by the sub-disciplines and the combinations thereof are eliminated by introducing a common terminology.

This part is considered as an optional read, making the thesis self-contained. Readers familiar with these topics may skip or cross-read the corresponding chapters, or use them for looking up specific terminology when studying subsequent chapters.

Part III. This part is dedicated to the trains of thought and formalisms underlying the actual contributions. In Chapter 6, the mutual pair-wise combinations of SCM, SPLE, and MDSPLE are portrayed, materializing in the integrating disciplines *model-driven product line engineering*, *model version control*, and *software product line version control*. Complementarily, *integrated historical and logical versioning* is considered from an SCM-centric perspective. In Chapter 7, the *design choices* and *decisions* of the framework are explored. To this end, several *challenges* connected to state-of-the-art integrating solutions are discussed in advance. Chapter 8 introduces mathematical *formalisms* on which the contributions are based, including *multi-version data structures*, *three-valued propositional logic*, and finally, the *Uniform Version Model* (UVM).

Part IV. Here, the actual theoretical contributions of this thesis are explained. The part begins with the introduction of the *conceptual framework*. Chapter 9 introduces a *hybrid version model*, providing formal definitions for revision graphs and feature models covering both revision and variant management.

In Chapter 10, a flexible *extrinsic product model*, integrating feature models as well as the "domain model", which in turn consists of model and non-model resources, is presented.

Chapter 11 deals with *consistency constraints*, whose introduction has become necessary due to the dynamism of the editing model. Algorithms for the operations CHECKOUT, COMMIT, and MIGRATE, preserving those constraints, are provided.

In Chapter 12, the conceptual framework is extended to *multi-user operation* by formalizing *collaborative revision graphs* as well as synchronizing operations PULL and PUSH.

The subsequent Chapter 13 deals with *metadata management* as well as with the question of *product well-formedness*, which arises due to the support of optimistic collaborative versioning as well as due to the possibility of combining optional features.

All chapters of Part IV are concluded by a discussion of related work with a focus on conceptual commonalities and differences, and with a summary referring back to the design decisions deduced in Section 7.

Part V. The implementation and evaluation of the whole conceptual framework are the subject of Part V, beginning with Chapter 14, where the design and realization of the tool *SuperMod* is explained. Related work is recapitulated, now with a focus on technical implementation.

Subsequently, Chapter 15 provides an overview of *case studies* conducted with SuperMod in order to assess the conceptual framework as well as its implementation. In addition to an experimental *evaluation* of the contributed concepts based on thoroughly defined research goals, questions, and metrics, subjective observations are assessed.

Part VI. In Chapter 16, the thesis is concluded by a top-down *summary*, a critical discussion of the *benefits* and *limitations*, an outlook referring to *future work*, as well as a discussion of the *academic* and *industrial relevance* of the approach.

In addition to lists of figures, tables, and code listings, as well as the mandatory bibliography, the back matter includes a list of abbreviations and an alphabetical index.

Without requirements or design,
programming is the art
of adding bugs to an empty text file.

LOUIS SRYGLEY

Chapter 2

Requirements and Benefits of an Integrated Approach

Abstract

This chapter begins with the central problem statement referring to the feasibility and the added value of an integrated conceptual framework for the combination of MDSE, SPLE, and version control. A brief literature review reveals that there exists currently no tool addressing a list of identified requirements in a satisfactory way. The combination of existing tools in turn implies significant drawbacks. Later on in this chapter, a top-down introduction to the tool SuperMod, which implements an integrated conceptual framework, is given. Then, an example of an iteration of SuperMod's filtered editing workflow is provided. Further aspects to be covered later in this thesis are listed, before we make explicit four decisive benefits supposed to be offered by the integrated solution.

Contents

2.1 Central Problem Statement

This thesis addresses requirements, the formal elaboration, the implementation, and applications of a conceptual framework for the integration of MDSE, VC, and SPLE.

The central problem statement underlying the contributions presented in the thesis thesis is the following:

> Is it possible to design and implement a development support tool that integrates the disciplines model-driven software engineering, version control, and software product line engineering?
>
> If the answer to the first question is yes, which is the added value of such an integration over a combination of state-of-the-art tools, and which are the limitations caused by it?

Before attending to these questions, existing solutions for pair-wise integration of MDSE, VC, and SPLE are briefly reviewed in Section 2.2. Then, we recapitulate the individual requirements of the considered sub-disciplines and align them with the surveyed categories of approaches (see Section 2.3). Section 2.4 sketches a consecutive tool integration scenario. Beginning with Section 2.5, the framework and tool contributed in this thesis are characterized in a top-down fashion and demonstrated by an introductory example. The benefits expected from this synergy are made explicit in Section 2.8.

2.2 Integrated Tools: A Brief Literature Review

This section provides a brief overview of state-of-the-art approaches and tools towards the pair-wise integration of MDSE, SPLE, and VC [1]. More comprehensive literature reviews follow in Chapter 6.

1 The section is based on the related work sections pre-published in [Schwä+15; SBW15; SW17b].

2.2.1 Model-Driven Software Product Line Engineering

To begin with, *model-driven software product line engineering* (MDSPLE) denotes the combination of MDSE and SPLE with the goal to boost productivity by abstracting from both the variability and the platform. For the realization of (model-driven) software product lines, three distinct approaches exist in the literature: *compositional*, *transformational*, and *annotative variability*. For annotative variability, we may distinguish between *filtered*, *partially filtered*, and *unfiltered* MDSPLE.

Approaches to MDSPLE based on *compositional variability* require specific tools in order to compose variable realization fragments with the common core, typically using *feature-oriented* [Ap+09a] or *aspect-oriented modeling* techniques [Wim+11]. This way, the core is kept small and concise, but conflicts may arise as soon as transformations or aspects are combined to realize several features.

In contrast to compositional approaches, where product creation is *monotonic* inasmuch as it exclusively adds components to the product in a suitable order, *transformational variability* assumes that the core model may be arbitrarily manipulated by adding, removing, or modifying details of product elements. Representatives are *language-based* [ZJ07] or *delta-oriented* [Zsc+10] tools. A special case of transformational variability is *clone-and-own* approaches, where a collection of variants with a common origin is gradually transformed into a software product line [LELH16].

Approaches based on *annotative variability* assume a *multi-variant domain model* that realizes all features of the product domain in a place; it must be syntactically well-formed. Existing approaches differ in how features are mapped to realization artifacts. On the one hand, visibilities may be stored within the domain model, e.g., using *annotations* provided by the modeling language [Gom05]. On the other hand, visibilities can be made explicit by using a distinct *mapping model* [BS15a].

A common assumption of the mentioned approaches based on annotative variability is that the user operates in a *multi-variant* view. When modifying the superimposition, all variants are visible at a time, and mapping information is added manually. In this way, *unfiltered editing* is realized. *Variation control systems* [Stă+16; LBG17] deviate from this editing model. Like in VCS, the user operates in a *view* (*filtered editing*, [SBK88]) and need not map model elements to features manually.

In [WO14], an approach to *partially filtered* editing of multi-variant programs is described. Since a part of the variability remains unresolved, corresponding variability annotations remain visible in the workspace. The approach is based on abstract syntax and may therefore be transferred to models.

A couple of MDSPLE tools offer *temporarily filtered editing*. For instance, *Feature Mapper* [HŞW08] offers the possibility of *change recording* during domain engineering. All recorded element insertions are associated with a feature expression derived from a provided feature selection.

The tool SuperMod presented in this thesis represents *fully filtered editing* [WC09]; changes are applied in a representative single version view, but may affect a larger set of variants.

2.2.2 Model Version Control

The term *model version control* (*MVC*) subsumes the combination of MDSE with version control [ASW09], with the goal of lifting version control metaphors (*check-out*, *commit*) up to the abstraction level of models. Rather than calculating differences on the low-level physical representation (e.g., lines of the text serialization), existing approaches to model versioning rely on operations such as *object insertion* or *attribute value update*. This improves both accuracy and consistency to a significant extent.

For storing and merging model versions, *change logs* may be used to describe the performed modifications. *Log-based* approaches such as [KH10] record the user's changes, but lack generality because they require a custom editor or at least extensions to existing editors. In case no change log is available or when intentionally following a *comparison-based* approach, changes need to be reconstructed by *matching* and *differencing*, as realized in [BP08]. The quality of the matching can be significantly improved by the use of *universally unique identifiers* (*UUIDs*) for model elements.

An important sub-problem of optimistic MVC is *three-way model merging* [Wes14].

2.2.3 Software Product Line Version Control

Software product line version control (SPLVC) deals with common problems that occur during the management of the life-cycle of software product lines, for instance, propagating changes from the variability model to the platform [SHA12]. Typically, platform and variability model are represented as artifacts on the same conceptual level, i.e., there is no "is versioned by" relationship in the sense of version control.

Many SPLVC systems have been built on top of existing VCS [ME08]; others rely on an implicit clone-and-own strategy [Pfo+16]. The component-based approach presented by [Tha12a] relies on change propagation at the level of derived products; the level of automation is low as manual visibility updates are required.

2.2.4 Integrated Historical and Logical Versioning

With *branches*, traditional version control systems [Cha09; CFP04] offer the management of logical variants to a limited extent; they do not support the creation of new variants based on a free combination of configuration options.

Approaches to *integrated historical and logical versioning* (IHLV) have been developed independently of SPL research and address software configuration management in a broader sense, promising to uniformly support evolution and variability. We distinguish between *asymmetric*, *orthogonal* and *hybrid* solutions.

Asymmetric approaches to IHLV do combine historical and logical versioning, but not at the same conceptual level. For instance, *Adele* [EC94] has logical variants built into its object-oriented data model using variability-aware program constructs; historical versioning is realized by a layer on top, which relies on directed deltas.

Furthermore, in [Rei95], an approach for orthogonal IHLV is proposed. A version cube is formed by product, revision, and variant space. In this way, evolution and variability are treated "equally before the law". Albeit, orthogonal solutions do not consider that the variant space may be subject to historical evolution.

Approaches towards *hybrid* IHLV promise to combine the advantages of the asymmetric and the orthogonal approach. The approaches described in the literature, however, lack implementation. In [ZS97], an approach to *unified versioning* based on *feature logic* [Smo92] is presented. Versions of artifacts (i.e., text files) are stored with selective deltas; visibilities are controlled by feature-logical expressions, which must be managed by the user. The *Uniform Version Model* (*UVM*) presented in [WMC01] generalizes and extends (historical and logical) versioning concepts initially introduced in the context of *change-oriented versioning* (*CoV*) [Mun93]. The version space is, however, represented at an inconvenient level of abstraction (propositional formula and set theory).

2.3 Requirements for a Fully Integrated Solution

The contributions made in this thesis are relevant for software projects that are subject to both evolution and variability.

To this end, let us assume a software project that requires to organize both historical – in a scale which makes collaborative version control facilities inevitable – and logical versioning—to an extent where fine-grained feature variability as provided by SPLE approaches is necessary. Such projects exist in the real world both in academia (e.g., MOD2-SCM, an SPL for VCS [BDW12]) and in industry (e.g., the Linux kernel [Ada+07]). Let us furthermore assume that the artifacts to be versioned include (but are not restricted to) *source code* and *models*, whose compliance to their respective programming languages and metamodels must be ensured at any time while the user manipulates them using his/her preferred modeling languages and tools.

Figure 2.1 illustrates that such a system needs to be organized along three dimensions: the *historical* dimension, which arranges subsequent points in time (*revisions*), and the *variant* dimension, which is organized along *variants*, which denote valid combinations of configuration options (features), and the *product* dimension, which consists of several (model or non-model) artifacts that can be uniquely identified.

Altogether, the assumed scenario deals with the management of requirements in five classes. Three of them conform to the dimensions *historical*, *variant*, and *product*. In addition, *cross-cutting* requirements emerge from the pair-wise integration approaches listed above. Last, *collaborative* requirements, which are typically addressed by VCS, but may also affect the variant and product dimension, must be handled distinctly.

The functional requirements listed below have been collected from an analysis of the state-of-the-art tools presented above; see alignment in Section 2.3.6.

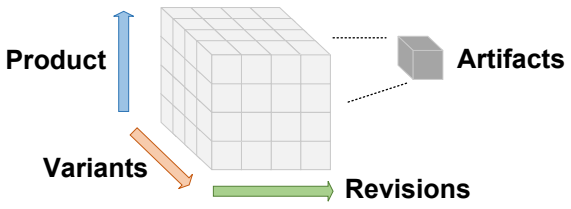


Figure 2.1: Different dimensions to be managed by an integrated approach. From [Rei95].

2.3.1 Historical Dimension

Requirements in the historical dimension coincide with the usual version control needs.

R1. Revision Graphs. The history of the software project should be described by a *revision graph* that, on the one hand, defines the set of revisions committed, and, on the other hand, arranges them in a (partial) order by defining predecessor/successor relationships.

R2. Extensional Revision Selection. A historical version (i.e., revision) is uniquely defined by selecting a single element in the revision set. According to [CW98], this principle is called *extensional versioning*.

R3. Immutability of Revisions. Once committed, a revision must be *permanently* available in the sense that checking-out the same revision should always reproduce exactly the committed state of the workspace. This disallows changes of version membership.

R4. Transparent Multi-Version Storage. The user should never get in touch with the repository-internal representation of multi-version artifacts, such that parts of the product not included in the selected revision are invisible to him/her.

2.3.2 Variant Dimension

Requirements in the variant dimension originate from SPLE tools. Furthermore, *views* shall be supported in order to reduce the complexity of multi-variant editing.

R5. Feature Models. *Feature models* are accepted as the de-facto standard for defining and representing variability in software product lines, so they should be supported as variability mechanism.

R6. Intensional Variant Specification. Based on a feature model, a variant is defined as a *feature configuration*, which binds each feature to a boolean selection state. Therefore, variants are freely compiled on demand based on the specification of their properties (features). In [CW98], this type of version definition is called *intensional*.

R7. Management of Variability Annotations. Variability annotations (also *traceability links* or *presence conditions*) control in which variants a specific part of the software is included. Some mechanism should be provided to directly or indirectly manipulate these annotations.

R8. Views on Product Variants. A variability management tool should offer facilities to temporarily hide parts of the product that are irrelevant for a specific change, and furthermore, to preview single product variants.

2.3.3 Product Dimension

The management of (model and non-model) artifacts being subject to evolution and/or variability should be as convenient and as consistent as possible. Here, we list requirements to complement the pure editing functionality that is provided by core development tools.

R9. Automated Product Derivation. Single-variant products are supposed to be derived in a preferably *automated* way from a multi-version representation by a unique version specification. In VCS, this is achieved by *check-out*; in SPLE, the derivation operation coincides with the starting point of the *application engineering* phase.

R10. Reuse of Existing Editing Tools. Whether or not they apply model-driven techniques, developers typically prefer to utilize existing editors rather than being forced to employ a new tool potentially bringing new limitations with it. This requirement is also valid for the development of evolving and variable software.

R11. Generality. Version control or SPLE approaches should make no or only few assumptions about the structure of the underlying product, such that it is applicable to different programming and modeling languages. In the ideal case, the product space can be specialized and extended towards different types of artifacts, such as text files, models, or databases.

R12. Product Well-Formedness Analysis. Before being presented to the user, single-version products should be checked for *well-formedness*. In the case of conflicts, the user should be notified, such that he/she may repair the product.

2.3.4 Cross-Cutting Requirements

In addition, there are requirements that demand that the three dimensions be mutually aware of each other.

R13. Evolving Feature Model. The feature model has to be aware of evolution in the same way as the underlying product dimension is. To this end, it should be available as an additional artifact in the workspace provided by the VCS.

R14. Overlap of Historical and Logical Versioning. Rather than being treated as two independent dimensions, historical and logical versioning must be allowed to overlap. For example, a change that is initially planned as a purely evolutionary increment might be connected to a new optional feature later.

R15. Uniform Mechanism for Evolution and Variability. When following a separate-tools approach, the management of revision membership – usually achieved by check-out/commit – and variant membership – by manual editing of variability annotations – is performed using different tools relying on distinct formalisms. An integrated solution should use a *uniform* versioning mechanism for evolution and variability.

R16. Fine-Grained Versioning. When considering the connection between the product space and evolution/variability management, *fine-grained versioning* should be applied. For instance, every detail of a model (such as the name of a model element) should be allowed to have different historical or logical versions.

2.3.5 Collaborative Requirements

In addition to cross-cutting requirements, two requirements for multi-user operation can be identified. Collaboration with respect to historical versioning is supported by most VCS.

R17. Multi-User Version Control. A minimal requirement for version control systems is that it can be used simultaneously by several developers, who share the versioned data.

R18. Collaborative Software Product Line Engineering. An integrated approach should also consider the specific needs of *collaborative SPLE*, including the orchestration of concurrent changes to the feature model, to the platform, and to variability annotations.

2.3.6 Alignment with State-of-the-Art Approaches

In order to prepare for a discussion of integration gaps present in the state of the art, we align the requirements listed above with categories of approaches listed in Sections 1.3 and 2.2. The explanations are backed by Table 2.1.

Version Control Systems. By providing immutable revision graphs, version creation by revision selection, and a transparent multi-version repository, VCS naturally address the requirements of the historical dimension in isolation. With respect to the product dimension, they automate the creation of workspace products conforming to selected revisions and they foster the reuse of existing editing tools rather than forcing the user into specific tools. Last, with respect to the collaborative requirements class, multi-user version control is enabled by optimistic or pessimistic synchronization, but collaborative SPLE is not explicitly addressed.

SPLE Support Tools. State-of-the-art SPLE support tools provide for intensional variant specification based on feature models. Furthermore, they allow to (directly or indirectly) manipulate variability annotations (i.e., the *mapping*). A smaller subset of SPLE tools investigated provides dedicated support for views based on filtered editing and/or for well-formedness analysis. Automated product derivation is typically included in the repertoire of functionality offered.

Model-Driven SPLE Support. MDSPLE tools advance the functionalities of general product line support tools by particular aspects belonging to the product dimension and to the cross-cutting class. They typically enforce the usage of a combination of SPLE-specific and independent editing tools. Furthermore, a subset of the surveyed tools is general with respect to the modeling language employed. Many approaches also address product well-formedness management. As far as cross-cutting requirements are concerned, MDSPLE support tools adequately address the hierarchical structure of models, such that fine-grained variability management is achieved.

Table 2.1: Alignment of surveyed approaches with requirements identified. The symbol ✓ denotes that the corresponding category of approaches fully satisfies the requirement, ? is for partial support, and an empty entry means that the requirement is not covered.

	Historical				**Variant**				**Product**				**Cross-Cutting**				**Collab.**	
	R1	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**	**R10**	**R11**	**R12**	**R13**	**R14**	**R15**	**R16**	**R17**	**R18**
VCS	✓	✓	✓	✓					✓	✓							✓	
SPLE					✓	✓	✓	?	✓			?						
MDSPLE					✓	✓	✓	?	?	?	✓	?				✓		
MVC	✓	✓	✓	✓					✓	✓	?	?	?			✓	✓	
SPLVC	✓	✓	✓	✓	✓	✓	✓	?	✓	✓			✓				✓	✓
IHLV		✓	?	?		✓	✓	?	✓	✓				✓	✓		?	

Model Version Control Systems. MVC systems can be considered as a specialization of VCS; therefore, they extend the list of requirements satisfied. In analogy to MDSPLE, some approaches are general with respect to the modeling language employed, and dedicated well-formedness analysis is provided to a limited extent, e.g., by consistency-preserving three-way model merging tools. As feature models can be interpreted as regular model instances by a subset of MVC systems, the cross-cutting requirement of feature model versioning is fulfilled partly. Last, fine-grained (historical) versioning is applied in the product space.

SPL Version Control Systems. Dedicated software product line version control systems combine the functionalities of VCS and SPLE. In addition, feature model evolution (as a cross-cutting requirement) and collaborative SPLE are enabled.

Integrated Historical/Logical Versioning. The diverse approaches to integrated historical and logical versioning (regardless of their classification into asymmetric, orthogonal, or hybrid), combine the definition of historical revisions and logical variants, but they do not typically rely on higher-level version abstractions (revision graphs and feature models). Some of the approaches discussed utilize a transparent multi-version storage, which can be indirectly accessed by views. The degree of automation of variability annotation management varies among the different approaches. In the product dimension, the VCS properties of automated product derivation and tool reuse are shared. As far as cross-cutting requirements are concerned, this category of tools exclusively addresses the overlap of historical and logical versioning by providing a uniform or unified versioning mechanism. Collaboration is supported partly.

Taken together, all requirements have been – at least partly – addressed by state-of-the-art approaches in isolation. Albeit, there is no tool satisfying them all at once, such that a combination of at least three tools must be employed (see discussion below).

The here presented conceptual framework and tool aim at satisfying all functional requirements mentioned here. We refer back to specific requirements in the conclusions of the respective chapters of Part IV.

2.4 Towards the Full Integration of MDSE, VC, and SPLE

To motivate an integrated approach that fulfills the requirements of all five classes, we explain the drawbacks of an "off-the-shelf" tool combination. Then, we gradually transition from a loosely-coupled into a fully integrated approach.

2.4.1 Drawbacks of the Separate-Tools Approach

Typically, the three dimensions are managed by disjoint tools: An editing tool – e.g., a text editor or a *computer-aided software engineering* (CASE) tool – is used for creating and modifying elements of the product dimension (i.e., both models and source code). The historical and variant dimensions is organized by VCS and SPLE tools, respectively. This strategy, here denoted as separate-tools, has several drawbacks:

Context Switches. Developers have to deal with at least three different tools, or families of tools, in order to create and maintain an evolving product line consisting of models. Each and every tool comes with its individual modes of usage and metaphors; e.g., VCS offer an update/modify/commit workflow whereas SPLE tools require to maintain feature models and multi-version artifacts.

Limited Support for Cross-Cutting Requirements. Even when using tools for pair-wise tool integration, listed in the brief literature review above, several cross-cutting requirements remain unsatisfied (see below). This may result in additional revision/variant management overhead and, even worse, in consistency problems.

Late Relevance of the Variability Dimension. In many (model-driven) software projects, support for variability is not planned from the beginning, but becomes relevant at a later phase in the product life-cycle. Retrospectively, it pays off to have a tool that allows to introduce variability on demand, without requiring heavyweight tooling set-up effort in advance.

2.4.2 Consecutive Integration

By step-wisely adopting tools for pair-wise integration (see Section 2.2), the separate-tools approach can be gradually transformed into a partially integrated, and finally, into a fully integrated, approach. Here, the added value of different stages of expansion are discussed in terms of requirements fulfilled. See Figure 2.2 for an overview of the consecutive integration steps. [2]

Separate Tools. When using appropriate software, the separate-tools approach may satisfy most requirements without any additional effort. In particular, this holds for requirements individual to the historical (**R1** – **R4**) and to the variant (**R5** – **R7**) dimension, respectively. The automated derivation of products (**R9**) and the reuse of editing tools (**R10**) are supported,

2 Here, we follow a specific integration path, consisting of MDSPLE; MVC, and last, SPLVC/IHLV. Notice that this order might be permuted arbitrarily, which produces different intermediate results in terms of dimensions (not) satisfied. The overall conclusion, namely that an integrated tool outperforms loosely coupled tool combinations, is independent of the order.

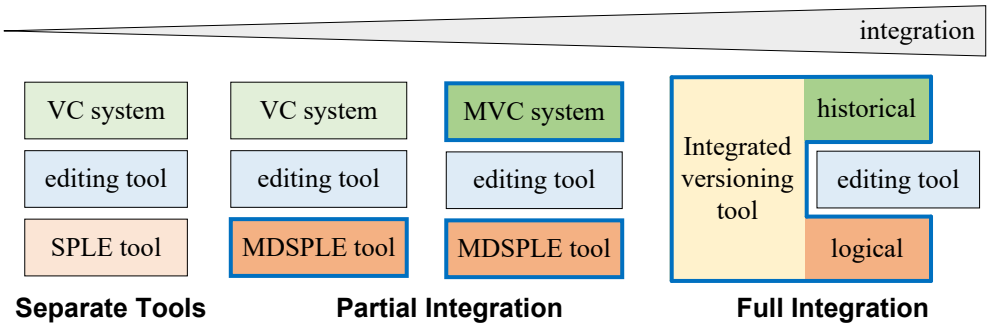


Figure 2.2: Consecutive integration of MDSE, SPLE, and VC.

too. Furthermore, **R17** (multi-user version control) is fulfilled by contemporary VCS. On the downside, product requirement **R12** (well-formedness analysis) is partly ceded to the editing tool, which is not aware of the multi-version context. Generality (**R11**), further cross-cutting requirements (**R13** – **R16**) as well as **R18** (collaborative SPLE) remain completely unsatisfied.

Partial Integration. In a first attempt towards tighter integration, the source code centric SPLE tool might be replaced by a model-driven substitute, which is aware of the syntactic structure of the artifacts part of the platform. As mentioned before, the mapping between the variability model and the platform is expressed by adequate model-driven abstractions such as deltas in the case of compositional or presence conditions in the case of annotative variability. This way, requirements **R11** (generality) and **R16** (fine-grained versioning) are fulfilled when confining to the variability management perspective.

Secondly, the conventional version control system might be abandoned in favor of a specific model VCS. Since the underlying MDSPLE tool represents both the product dimension and the variant dimension as models, they may be handled adequately by the MVC system, such that requirements **R11** and **R16** are now satisfied also from the evolution perspective. Furthermore, **R13** (evolving feature model) is partly fulfilled.

Full Integration. In order to meet the remaining cross-cutting requirements, namely **R14** and **R15**, it is necessary that one and the same tool controls both evolution and variability. This enables dedicated support for views (**R8**), product well-formedness analysis (**R12**), and collaborative SPLE (see **R18**).

Candidate tools to provide this type of integration have been categorized into SPLVC and IHLV. Albeit, the requirements that they satisfy are mutually disjoint. Furthermore, a loosely coupled solution consisting of a combination of pair-wisely integrated tool is still affected by the drawback of context switches. For this reason, we conclude that a single tool that offers *full integration* is indispensable.

Notice that the integration strategy presented here does not aim at eliminating the editing tool. In contrast, we strive for a co-existence of existing (model or source code) editing tools with a support tool, which complements rather than replaces the former.

2.5 SuperMod: Top-Down Tool Description

As core contribution of this thesis, we present a conceptual framework that realizes a fully integrated MDSE/VC/SPLE solution, satisfying all requirements stated above. The functional and architectural considerations underlying the conceptual framework are best explained by means of an informal tool introduction. An example for the usage of the tool is provided thereafter.

SuperMod [SBW15] is a model-driven tool that combines metaphors of VC and SPLE. From VC, *check-out* – for importing a specific version from a software repository into a workspace – and *commit* – for transferring changes performed in the workspace back to the repository – have been borrowed; SPLE has influenced SuperMod in terms of *feature models* – hierarchical decompositions of mandatory and optional features – and *feature configurations*—unique selections in the feature model describing the characteristics of specific product variants. Complementarily, *feature ambitions* are newly introduced as partial selections in the feature model delineating the set of variants to which a change, representatively performed in the workspace, applies.

2.5.1 Architectural Overview

Like ordinary VCS, SuperMod distinguishes between a *workspace* and a *repository*.

The workspace is a region in the file system where a *working copy*, consisting of both model and non-model artifacts, is made available to the user, who uses his/her preferred tool(s) to apply modifications. Specifically in SuperMod, the workspace contains an additional artifact: the *feature model*, which can be edited by the user contemporaneously.

The repository in turn is a persistent, multi-version storage whose contents are transparent to the user. Its contents are schematically sketched in Figure 2.3. The *version space* contains the elements from which version specifications are constructed—here, revisions from a *revision graph* and features from a *feature model*. In contrast, the *product space* contains elements that are combined to specific product versions made available in the workspace. Here, the product space consists of the *feature model* and of the primary versioned artifacts, the *domain model*. [3]

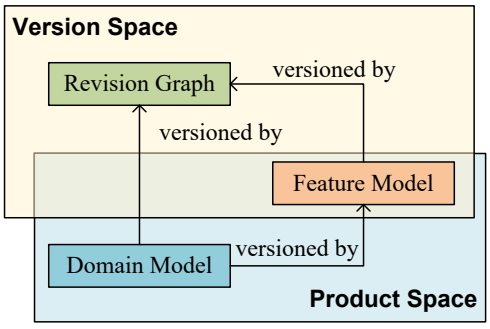


Figure 2.3: Sketch of the contents of a SuperMod repository.

3 Although the term "model" is used here, this may comprise multi-version representations of several model and non-model artifacts conforming to different metamodels and languages.

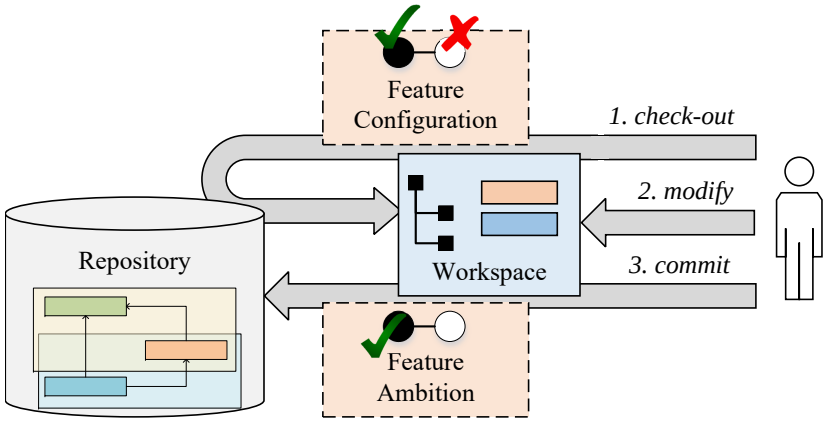


Figure 2.4: A compact illustration of SuperMod's dynamic filtered editing model.

2.5.2 Dynamic Filtered Editing Model

To combine these concepts, the tool provides a *filtered editing model* (cf. Figure 2.4), an iteration of which consists of three essential steps, CHECKOUT, MODIFY, and COMMIT.

1. Select a revision in the revision graph. Define a *feature configuration* as a unique selection in the (selected revision of the) *feature model*. The specified variant of the *domain model* is made available in the *workspace*.
2. Let the user modify the model in the workspace. These changes should reflect an *increment* being connected to a subset of the selected features.
3. Specify a *feature ambition* as a partial selection in the *feature model*. The increment is transparently connected to the selected features, and the change is made persistent in the *repository*. A new *revision* is introduced transparently.

The editing model is called *dynamic* for two reasons. First, the feature model may co-evolve with the domain model during the modify step. Second, the ambition is defined at commit and it may refer to newly introduced features.

2.6 Fast-Forward Example: Graph Library Product Line

This section gives a brief insight into the usage of the tool *SuperMod*.

As a running example, a product line for a *Graph Library* [LHB01] has been chosen. Graphs are a meaningful abstraction in computer science helping to analyze and solve complex problems by means of *graph algorithms*. In general, a graph consists of a set of *vertices* and a set of *edges*, each connecting two vertices. Depending on the concrete problem statement, graphs may be *directed* (i.e., an edge has a dedicated source and target vertex), *weighted* (i.e., there is a numerical value assigned to each edge), or *labeled* (i.e., vertices and/or edges have a unique label assigned). In addition, specific graph algorithms assume that each vertex has a specific *color* assigned. Here, the Graph Library is represented as a UML-compliant class diagram, from which concrete graphs may be instantiated.

2.6.1 Example Iteration Seen from the End User's Perspective

The editing model in action is exemplified by an iteration that addresses the realization of the optional feature weighted, distinguishing weighted from unweighted graphs. Four preceding revisions have already been realized.

In the top left corner of Figure 2.5, the current revision 4 of the *feature model* before executing the iteration is shown. By definition, the feature model is hierarchical, starting with the root feature Graph. This contains two mandatory (circles) sub-features Vertices and Edges, which in turn contain optional features (empty circles) colored and labeled.

During check-out, a *feature configuration* is requested from the user. In the example, all features are selected, except for color, which is deselected. As a consequence, an uncolored, labeled, and weighted graph is exported into the workspace. The realization of the selected feature labeled (realized in a preceding iteration) is visible in form of the attribute label.

Now, the workspace may be modified arbitrarily. In our example, the fictional user adds an attribute weight of type double to the class Edge. Correspondingly, it is this attribute that constitutes the *increment* realized in the presented iteration.

The change shall be connected to a new feature weighted, which is added to the feature model below Edges also during the modify step. The bottom left corner of Figure 2.5 depicts

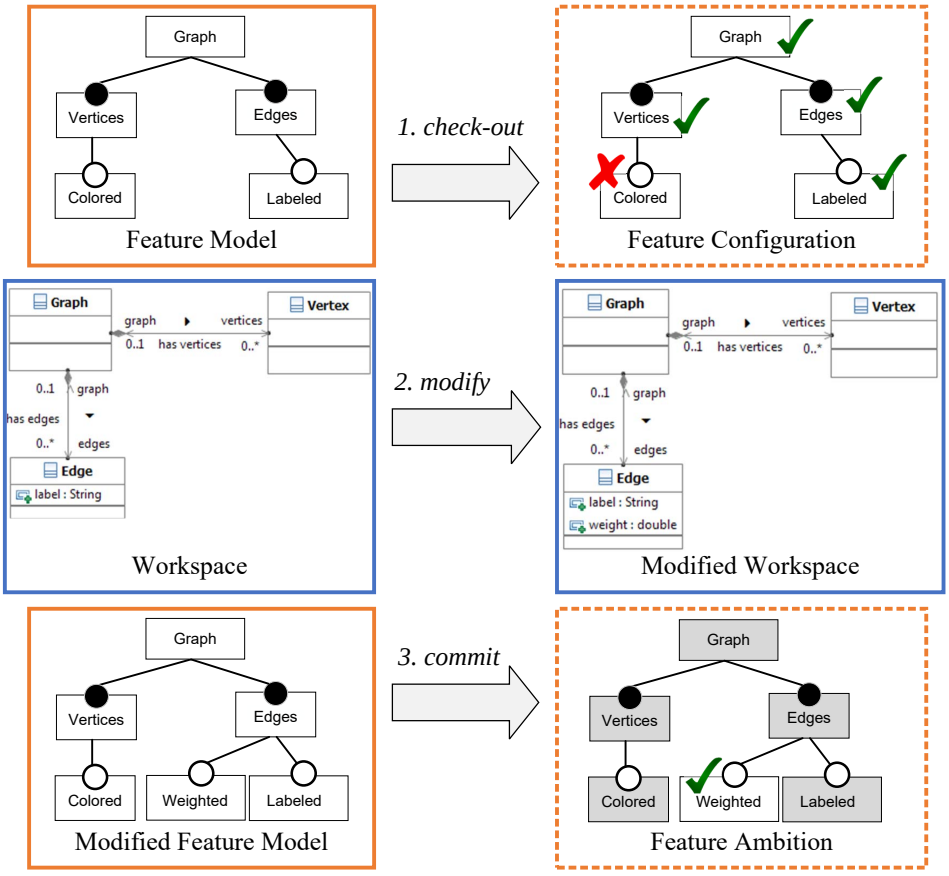


Figure 2.5: SuperMod example iteration: realization of the feature weighted.

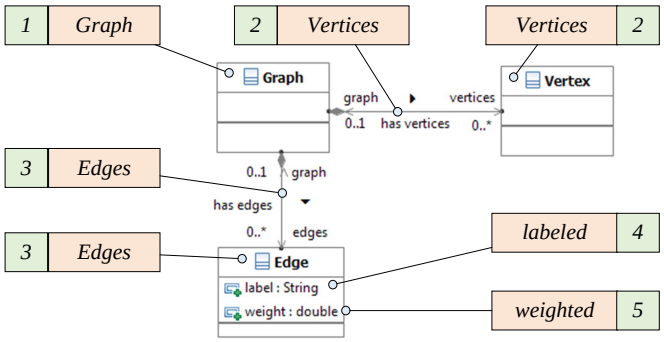


Figure 2.6: Example: internal representation of the superimposition. Visibilities, each including a historical and a logical component, are illustrated as attached comments.

the modified state of the feature model.

The connection between the change to the model and the newly introduced feature is established during commit where a *feature ambition* is requested from the user. In contrast to feature configurations, feature ambitions are *partial*, i.e., they may leave features unbound (gray background). In our case, exclusively weighted is selected positively since the change is immaterial to all other features. The changes are, furthermore, associated with the new revision 5, which is transparently introduced in the revision graph.

As a consequence of the performed iteration, attribute weight will be checked-out into the workspace if a revision greater or equal than 5 is selected and the specified feature configuration includes a positive selection for the feature weighted.

2.6.2 A Glance Behind the Curtains

A greater part of this thesis deals with what happens inside the repository transparently when the designated end user issues a check-out or commit. An intuitive idea of the underlying internal concepts and mechanisms is provided here.

Internally in the repository, the entirety of all versions of the domain model is represented in the form of a *superimposition*, i.e., a multi-variant domain model; this can be understood as a model instance having *visibilities* assigned to its objects. A visibility in turn refers to features of the feature model and to revisions of the revision graph.

The superimposition belonging to the Graph Library example in its state after the iteration performed above is depicted in Figure 2.6. The attribute weight added during the modify step has been associated with the feature weighted and with revision 5 as expected. Further visibilities have emerged from previous commits in an analogous way.

2.7 Further Aspects

This informal top-down example has illustrated some core principles of the formal approach developed in this thesis. More sophisticated use cases are considered in the subsequent chapters. They deal, among others, with the following items.

Revision Graph Management. In contrast to the feature model, the revision graph is not available for modification in the workspace, but managed automatically. The underlying mechanisms need to be further explained.

Dual Role of the Feature Model. Figure 2.3 illustrates the dual role of the feature model, which is part of both the version space and the product space (as it is subject to evolution). This design decision is of central importance and implies a multitude of consequences, which are thoroughly discussed in the remainder.

Product Space. The Graph example's product space consists of a single UML class diagram only. Yet, a multitude of modeling languages as well as text files must be supported in order to make the tool and approach applicable to realistic model-driven scenarios. Furthermore, fine-grained management of the product space is required.

Commit Semantics. In the example, the visibilities of inserted elements are associated with a feature selected in the ambition as well as with a new revision created during commit. Though, deletions and modifications of elements have not been considered. Furthermore, how are visibilities shaped when multiple features are selected positively and/or negatively in the ambition?

Feature Model Editing. In the example iteration, the feature model was only slightly modified. More advanced facilities for feature model editing are provided by the approach and tool. To this end, the semantics of feature models with respect to valid feature configurations must be clearly defined.

Dynamic Filtered Editing. In the informal tool introduction, a specific *dynamic* editing model has been applied. In contrast to other approaches to filtered editing described in the literature, SuperMod allows for parallel feature model and domain model editing, and requests to fix the scope of a change – i.e., the ambition – not before commit. This increases the flexibility of MDSPLE, but also raises new consistency problems, which remain to be examined and solved.

User Assistance. It has been shown that the contributed editing model automates variability management to a large extent. Though, specifying feature configurations and feature ambitions is still tedious, requiring dedicated user assistance in order to avoid, e.g., repeated check-outs with equivalent feature configurations.

Multi-User Operation. SuperMod is a fully-fledged version control system enabling collaborative (SPL) development. Questions such as synchronization of concurrent changes remain to be answered.

Product Well-Formedness Analysis. Likewise, single or multiple users may cause well-formedness violations referring to both the domain model and the feature model. Such situations must be adequately reported to the user, who resolves conflicts preferably in a single-version view.

Processes and Adoption Paths. The thesis at hand also discusses how both the tool and its underlying approach match different development processes such as plan-driven or agile, as well as different SPL adoption paths such as proactive, reactive, or extractive.

2.8 Benefits of the Integrated Approach

Taking into account the remarks given in the previous sections, the here presented fully integrated solution for MDSE, VC, and SPLE may advance the state of the art of tool support by the following items: [4]

B1. Uniform Versioning. The integrated solution provides a uniform concept for the internal representation of logical and historical versions. Until writing back a change to the repository, the user may decide if it incorporates a new revision of an existing variant, or a new variant that co-exists in parallel.

B2. Reduction of Cognitive Complexity. By the adoption of the version control metaphors *check-out* and *commit* in connection with filtered editing, the integrated solution reduces the cognitive complexity for the development of a model-driven software product line. The user applies changes in a single-version view, and variability annotations are updated *automatically*. The user is neither faced with difficult architectural decisions related to a multi-variant domain model, nor with the necessity to describe feature realizations by means of model transformations or composition rules.

B3. Unconstrained Variability. State-of-the-art SPLE support tools require that the multi-variant domain model artifacts part of the platform be valid with respect to its metamodel. The integrated approach allows models to vary arbitrarily within the multi-version repository; this property is inherited from version control systems. Likewise, single-version workspace artifacts should still be constrained by their respective programming languages or metamodels.

B4. Tool Independence. The workspace, where the dedicated end users modify the contents of the product line, contains artifacts that comply to standardized representations, such that they may be edited with existing (single-version) model or non-model tools. No technical adaptations are necessary to integrate exiting tools with SuperMod.

This list is revisited in the conclusion of this thesis in order to confirm that the benefits are actually fulfilled by the contributed conceptual framework.

2.9 Summary and Outlook

In this chapter, the requirements for an integrated solution towards MDSE, SPLE, and VC have been identified. A brief literature review has revealed that many requirements can be fulfilled by combining tools that apply a pair-wise integration of the relevant disciplines, but there exists currently no tool to match them all at a time. Cross-cutting requirements can be satisfied optimally only by a tool that explicitly combines properties of SPLE and VC and, furthermore, is aware of the model-driven structure of the versioned product.

The core contribution of this thesis is a conceptual framework to integrate MDSE, SPLE, and VC in face of these requirements. Alongside of a first introduction, the key functionalities of the tool *SuperMod*, which implements the framework, have been demonstrated.

4 An extended version of this list has originally been published in [Schwä+15].

The framework and tool are tailored towards *cross-cutting* requirements such as dedicated support for feature model evolution (cf. **R13**), awareness of the overlap of historical and logical versioning (**R14**), a uniform mechanism for evolution and variability management (**R15**), and fine-grained versioning (**R16**) of structured artifacts.

The integrated solution profits from four benefits: *uniform versioning*, *reduction of cognitive complexity*, *unconstrained variability*, and *tool independence*.

In a fast-forward tool introduction, we have already demonstrated the unique characteristics of SuperMod when compared to other tools described in the literature. First, the here contributed conceptual framework assumes *fully filtered editing*. Multi-version artifacts, i.e., variation points, variants, and version membership information, are never exposed to end users, who apply changes *representatively* in a product variant instead. The integration of the performed changes into the SPL platform is *automated* to an extent that goes beyond the state of the art in SPLE research. Second, the *integrated hybrid repository architecture*, which is part of the conceptual framework, is unique in the literature. The *feature model* plays a dual role, providing both a product versioned by the revision graph and a variability model that versions the software project.

The requirements list collected here is recaptured in Chapter 7, where the central design decisions for the conceptual framework are recorded. The decisions build upon a deeper analysis of the disciplines MDSE, SPLE, and VC, and of pair-wisely integrated approaches. The results of a more extensive literature review are presented in the subsequent four chapters.

Part II

Three Software Engineering Sub-Disciplines

Note to the Reader. The three chapters of this part provide general introductions into the topics *model-driven software engineering*, *software configuration management*, and *software product lines*. Readers familiar with these topics may skip the corresponding chapters and resume on page 97.

The creative activity of programming
– to be distinguished from coding –
is [. . .] considered as a sequence
of design decisions
concerning the decomposition
of tasks into subtasks
and of data into data structures.

NIKOLAUS WIRTH (1971)

Chapter 3

Model-Driven Software Engineering

Abstract

This chapter provides the first portion of the background of this thesis by surveying the first of the three relevant sub-disciplines in which the contributions are made. Model-driven software engineering aims at automatically deriving executable source code from models, which serve as high-level specifications and are expressed by well-defined modeling languages such as UML, or DSLs. The chapter does not only explore the theoretical foundations of models and metamodels; it also includes a survey of technical solutions. The technological ecosystem is provided by the Eclipse Modeling Framework, whose language definition facilities are outlined before the chapter concludes with an overview of the key element of model-driven software engineering: model transformations.

Contents

3.1 Model-Driven Engineering and Model-Driven Architecture

As anticipated in the introduction, the term *model* is used in software engineering for more abstract but less detailed descriptions of software systems. Stachowiak's *general model theory* [Sta73], which has its origins in philosophy, is often taken into account. According to his definition, a *model* exposes three essential *features* to its stakeholder(s):

Mapping. A model must be based on a concrete artifact, the *original* it reflects.

Reduction. The model incorporates only a relevant subset of the original's properties.

Pragmatism. The model must be capable of replacing the original for a specific purpose.

The purpose of models is manifold—they may be used to analyze existing software, to communicate and understand customer requirements, or to design an implementation template that will eventually be taken into consideration when coding, deploying, or maintaining the actual software application. *Model-driven software engineering* (MDSE) [SV06] makes systematical use of the idea of models being implementation templates and aims at automating the majority of tasks necessary to obtain the executable source code.

Model-driven architecture (MDA) [KWB03; Mel+04] is a conceptual framework from which a set of standards has been derived [MM03]. Essentially, it describes the disciplined use of models, as classified in the next section, and model transformations, which are outlined in Section 3.7, with the aim to significantly increase the productivity of software engineering by replacing coding by modeling to the greatest possible extent. Another important principle is the separation of *functionality* (as exposed to the end user) and *technical platform*. Thus, MDA may be considered as a specialization of MDSE.

3.2 Classification Dimensions for Models

Apart from the commonalities reflected by the essential model features listed above, there are several dimensions along which models may be distinguished when used in the context of software engineering. Figure 3.1 summarizes the explanations given below.

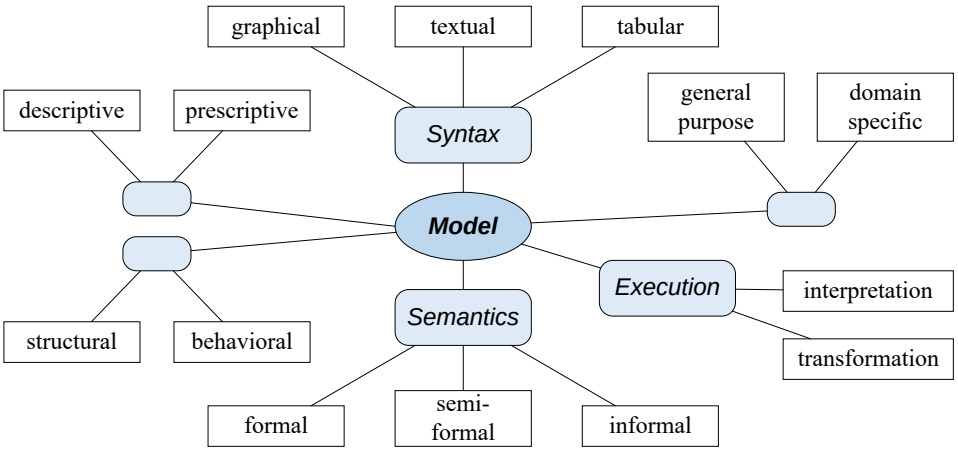


Figure 3.1: Classification dimensions of models used in software engineering.

Descriptive vs. Prescriptive Models. In experimental sciences, the term *model* is frequently used for something that describes a cut-out of reality and allows to make predictions for future. Such models are called *descriptive*—they describe an original that actually exists in the real world *before* a model is derived from it. Descriptive models also exist in software engineering, especially when applying *reverse engineering* [Mül+00]. In this case, a model is derived from the legacy source code of an existing software system and thereafter used for analyzing the underlying design principles.

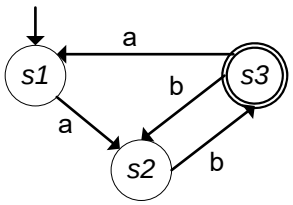
Being an engineering discipline, software development makes extensive use of *prescriptive models*, which are created in advance to an original—the executable software [Béz05]. The main reason for following this approach is the complexity of the system to be developed. Well-defined modeling languages allow to document and to communicate design decisions in a more comprehensible way by abstracting from implementation details irrelevant for the architecture of the system. Prescriptive models are at the heart of MDSE and MDA.

Concrete Syntax. In software engineering, the term *model* is often implicitly connected to a *diagram* representation. The notation of the most prominent modeling language, UML (see Section 3.4), is almost entirely graphical. The reason why graphical notations are considered as suitable abstractions is that they illustrate elements (e.g., classes or states) and their relationships (e.g., associations or transitions) in an intuitive way.

Albeit, modeling languages are not confined to graphical notations. Using the role model of programming languages, many modeling languages, especially *domain-specific languages*, have a textual notation. For technically experienced users, text is easier to write and modify. Furthermore, the development of textual modeling tools tends to be easier. Last, texts scale larger than diagrams.

Conceivably, further forms of concrete syntax exist. For instance, *tables* may be considered as an additional form or model representation. Figure 3.2 illustrates the same model, a state chart, with three kinds of concrete syntax: the well-known graphical syntax, a fictional textual syntax as well as a *transition table* [HMU06].

Graphical



Textual

```
state machine {
  initial state s1 {
    'a' -> s2
  }
  state s2 {
    'b' -> s3
  }
  final state s3 {
    'a' -> s1
    'b' -> s3
  }
}
```

Tabular

	a	b
* *s1*	*s2*	-
s2	-	*s3*
s3 *	*s1*	*s2*

Figure 3.2: Different semantically equivalent notations: graphical, textual, and tabular.

General Purpose vs. Domain-Specific Modeling Languages. Programming languages can be distinguished into *general purpose* and *domain-specific* languages. While general purpose programming languages such as C++ [Str13] or Java [Gos+15] can be employed to develop any kind of application, domain specific languages have a restricted field of application, but typically outperform general purpose languages in terms of declarativeness and readability. For instance, the *structured query language* (*SQL*) [HM08] has been designed specifically for database queries.

An analogous distinction exists for modeling languages: General purpose modeling languages such as UML are suitable for the analysis and design of systems being developed in general purpose programming languages. *Domain-specific modeling languages* (DSMLs) may abstract from general purpose modeling languages; e.g., there exists a C code generator for *MATLAB Simulink* [Beu06]. To DSMLs, a large body of research has been dedicated [Fow10; Völ+13].

Structural vs. Behavioral Models. Regardless of whether being of general purpose or domain-specific, there exist languages for structural and for behavioral models, respectively.

A *structural model* describes the entities a software system is composed of, as well as possible connections between these entities. A well-known example is UML class diagrams, where classes define types for objects to be instantiated from the application, as well as attributes and associations to other types of objects.

Class diagrams, however, do not define how and when objects are created and modified. For this purpose, structural models need to be augmented with *behavioral models*, e.g., UML sequence diagrams. A behavioral model describes possible interactions between the entities of a software system, including creation, modification and deletion of them.

The interplay between structural and behavioral models is exemplified in Figure 3.3. A structural model for a cut-out of the running *Graph Library* example is defined as a UML class diagram on the left hand side. On the right hand side, the behavior encapsulated in the operation addEdge is specified using a refining UML activity diagram.

Moreover, the separation between structural and behavioral models is not mandatory. There exist hybrid forms, e.g., structural models augmented with behavioral specifications, such as the *Action Language for Foundational UML* (Alf, see Section 3.4).

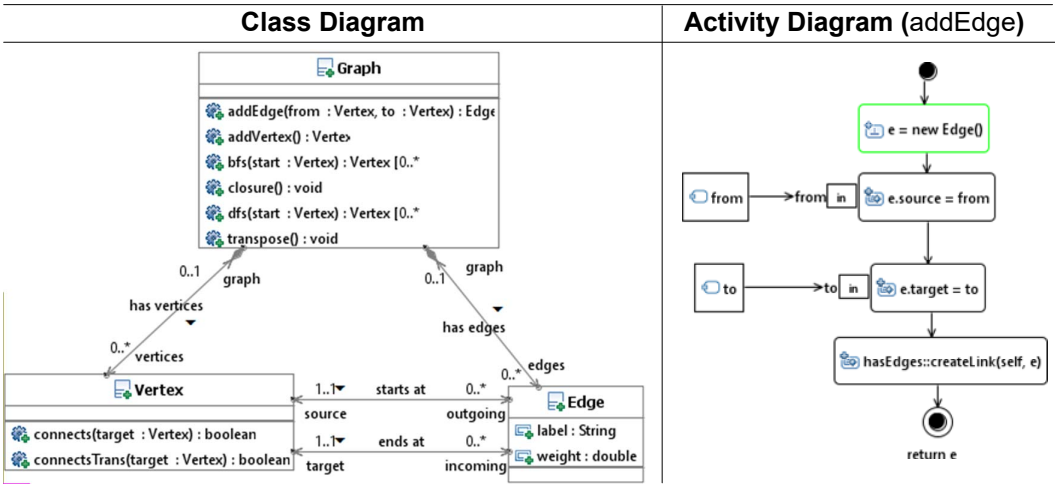


Figure 3.3: Example of a structural UML class diagram being complemented by a behavioral UML activity diagram. Based on [BS16b, Figures 3 and 4].

Semantics. In order to be purposeful to their stakeholder(s), models need not only conform to well-defined syntactical guidelines, but also have a clear *semantics*. There are several degrees of how precise the semantics can be defined.

First, semantics may be defined in an entirely *informal* way. For instance, the mind map depicted in Fig. 3.1, although conforming to the concrete graphical syntax rules defined for mind maps, cannot be interpreted by a machine unambiguously. Models based on informal semantics require human efforts to be convertible into runnable source code.

Semi-formally defined semantics means that there is room for interpretation caused by, e.g., limited expressiveness of the modeling language, or by ambiguity. For instance, UML is defined in a semi-formal way by a standard [OMG15], in which prose is used to define the semantics of specific modeling constructs. The accuracy of definitions varies. For instance, the transformation of class diagrams into runnable source code is more obvious than, e.g., the execution of informal use case diagrams.

In case the specification of a modeling language contains no ambiguities, it provides *formal* semantics to its models. This may be achieved either by using an unambiguous formalism, or by providing a reference transformation that actually implements the semantics by transforming model instances into lower-level entities, e.g., source code.

Execution. Models that have formal semantics may be *executable*, i.e., they may be run on a computer using appropriate tools. Model execution may be realized either by using *model transformations* (see Section 3.7) which convert model instances into executable models or programs in a lower-level language, or by *interpretation*. A *model interpreter* is a program that takes as input a model and directly executes the behavior specified there [ML12; Car+13].

3.3 Modeling Languages, Metamodels, and MOF

The syntax and semantics of models are defined by the respective *modeling languages* they are supposed to conform to. In the field of MDSE, there are different formalisms for the definition of a modeling language. Most of them assume a separation between *concrete syntax* – the visual appearance of model elements – and *abstract syntax* – the internal representation of model instances –, a distinction that has been borrowed from compiler construction [Aho+06]. Technical approaches to the definition of concrete syntax is a subject of Section 3.6.

3.3.1 The OMG Metamodeling Hierarchy

The conventional formalism for describing the abstract syntax is *metamodels* [Küh06]. A metamodel is a model that defines which elements and which relationships are available in a specific modeling language. More precisely, a metamodel defines, e.g., *classes* and *references*, which are instantiated in models as *objects* and *links*.

Being a model itself, the question arises which is the modeling language for metamodels. This, however, depends on the specific *modeling paradigm* used. The *Object Management Group* (OMG) have standardized a three-level metamodeling hierarchy that is used in the majority of MDSE applications nowadays: *Meta Object Facility* (MOF), see Figure 3.4. On level *M1*, models are located; *M2* represents metamodels; *M3* defines a fixed metametamodel (i.e., metamodel for metamodels). This level is where the concepts of *classes* and *references* are defined for usage at M2. In addition, level M3 is *self-defining*: The metametamodel MOF conforms to itself and therefore obviates the need for levels M4 or higher.

Below the three modeling levels, a pseudo level *M0* is defined, which represents everything that is reflected by M1 but that is not part of M1 itself. Depending on the specific context where models are used, M0 may either reflect the real world (e.g., in the case of descriptive analysis models) or run-time instances of the generated source code (e.g., in the case of prescriptive implementation models).

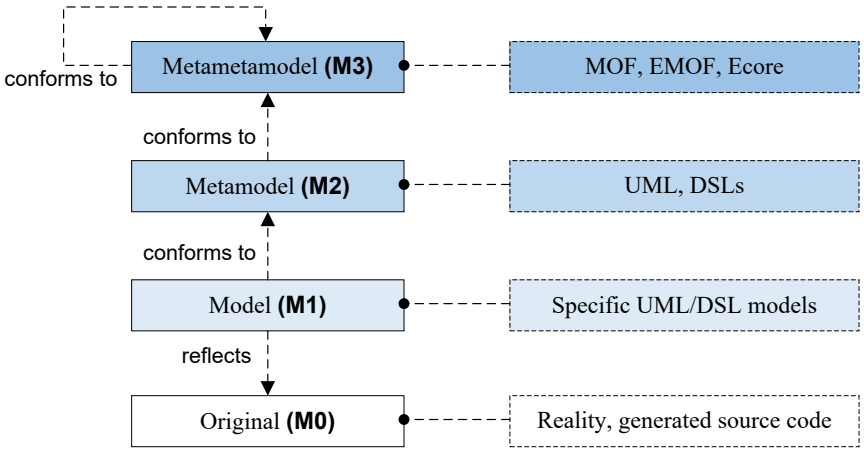


Figure 3.4: Hierarchy of metamodeling levels as proposed by the OMG.

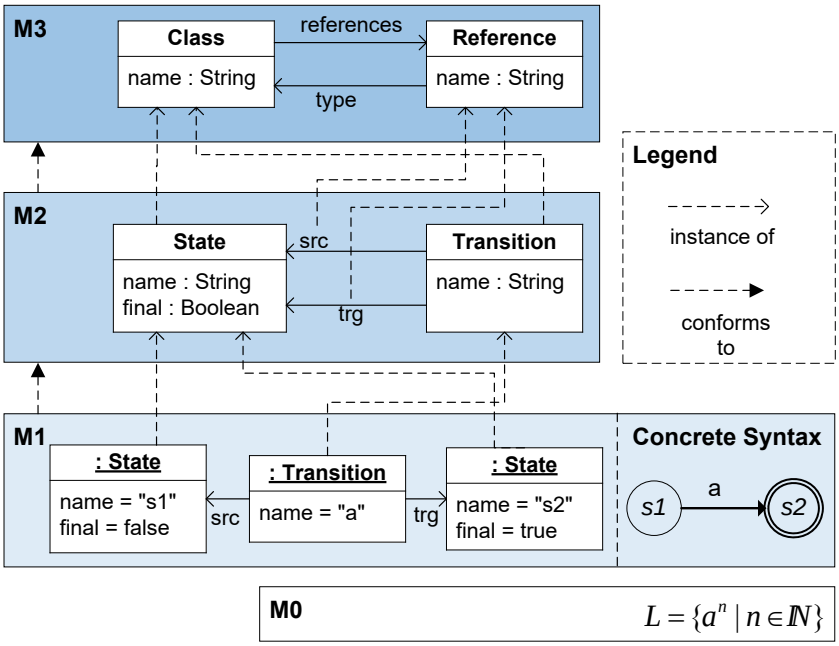


Figure 3.5: Example for the use of the MOF levels: state chart.

3.3.2 Example

Figure 3.5 shows an example involving all meta levels M3 until M0. In the *M3* box, a cut-out of the OMG MOF metametamodel is depicted. It defines the modeling concepts Class and Reference, both carrying a name. Each reference belongs to a specific class and has a specific type. These concepts are instantiated at level *M2*, which defines a metamodel for simple state diagrams that conforms to the M3 model. The classes State and Transition are introduced, both carrying an attribute[1] name. The concept Reference is instantiated twice, defining the source (src) and target (trg) states of a transition. On level *M1*, an example of a state diagram is shown in abstract (represented as object diagram conforming to the M2 class diagram) and in concrete graphical syntax. It instantiates the class State twice and defines a transition a. Furthermore, the references src and trg are instantiated as links. On level *M0*, the "real-world original" for the state diagram is shown: the context-free language described by the state chart.

3.3.3 XML Metadata Interchange

With *XML Metadata Interchange* (XMI), the OMG have defined an XML-based serialization standard for model instances located at M1, M2, or M3 [OMG15a]. XMI is intended as a purely internal representation, i.e., modelers typically use specific modeling tools to edit models based on their concrete syntax, whereas the model instances are persisted in their abstract syntax in the XMI format.

For the remainder of the thesis, it is important to notice that indisciplined text-based

1 The M3 class Attribute instantiated here is not shown in the M3 compartment of the figure.

modifications to XMI files can destroy their well-formedness, making them unreadable for modeling tools.

3.4 UML, OCL, and Alf

The *Unified Modeling Language* (UML) represents the currently most widespread modeling language. Since it has been used and is used in many examples in this thesis, some background shall be provided here. Furthermore, the closely related standards OCL and Alf are characterized.

3.4.1 Unified Modeling Language

In the 1990s, UML has been issued by the OMG with the aim to establish a standardized notation for software models [OMG15]. UML is a *general purpose* language in the sense that it may be used for modeling software systems of any domains. Furthermore, it covers both *structural* and *behavioral* models and may be applied in all phases of software engineering, i.e., its application ranges from requirements engineering over analysis, design, and implementation, to deployment and maintenance. As a consequence, as well as due to the fact that UML has historically grown from several ancestor languages, UML's language extent is enormous, consisting of 14 diagram types having more than 300 different element types[2] available; Figure 3.3 shows two example diagrams.

UML's *concrete graphical syntax* has been defined informally by means of example diagrams in the standard [OMG15]. Moreover, the *abstract syntax* is defined by a metamodel defined at level M2. For each model element available in the UML, there exists a corresponding class in the UML metamodel. Each model element, i.e., each object used in a specific UML diagram, is a M1 instance of a class of the UML metamodel. Unlike sketched in Figure 3.5, UML's language architecture is slightly more complicated [Fow03; HK05]: Rather than conforming to MOF, a small subset of the UML metamodel – the so called *infrastructure* – has been chosen at M3 level. The UML infrastructure in turn corresponds to a subset of MOF, defining core concepts such as classes and associations. The majority of classes of the UML metamodel is contained in the *superstructure*, which can be considered both as an extension to and as conforming to the infrastructure. Therein, all diagram elements, such as use cases and activities, are defined.

3.4.2 Object Constraint Language

Structural UML models such as class diagrams define *types* of objects to be instantiated. Furthermore, *multiplicities* may be specified for attributes and references. Typing and multiplicity rules incorporate necessary preconditions for model instances to be syntactically *well-formed*. However, specific application contexts may require additional conditions that cannot be expressed by a purely static model. This is where the *Object Constraint Language* (OCL) [WK98] steps in. The OCL standard defines a textual language that augments an existing (UML or MOF compliant) class diagram by different kinds of *constraints* [OMG14].

2 The UML 2.4.1 superstructure metamodel contains 331 classes from which elements may be instantiated.

In addition, the language allows to specify the *behavior* of structural models to a certain extent, e.g., by expressions that describe the result of an operation call or a derived attribute. Albeit, OCL has been designed as a *functional* language free of *side effects*, such that the creation, modification, and deletion of objects cannot be directly expressed. As a replacement, OCL allows to define *pre* and *post conditions* for state-changing operations.

3.4.3 Action Language for Foundational UML

Alf has been standardized as a "textual surface representation for UML modeling elements" [OMG13, p. 1], which particularly focuses on the specification of *behavior*. In contrast to OCL, Alf does support the specification of side effects, i.e., object creation, modification, and deletion. Alf, however, does not require a specification of the underlying model as UML class diagram. Rather, Alf expressions are embedded into different diagram types—e.g., the actions specified in the activity diagram in Figure 3.3 are Alf-compliant. Furthermore, Alf code may be embedded in a textual syntax for a subset of UML which is known as *foundational UML* (fUML). Being independent of specific implementation languages, Alf contributes to the goal of separation of functionality and technical platform as demanded by MDA. Despite being textual, Alf specifications are fully-valued *abstractions* of source code, since the language provides higher-level language concepts, e.g., for the instantiation of associations as links.

3.5 Eclipse Modeling Framework

While the OMG standards described above – MDA, MOF, XMI, UML and OCL – aim at providing common conceptual methodologies and defining interchange formats for different tool vendors, the *Eclipse Modeling Framework* (EMF) is a technical modeling solution having widespread use both in industry and in academia [Ste+09]. As its name suggests, EMF is based on the *integrated development environment* (IDE) *Eclipse*[3]. Eclipse is a highly extensible IDE [CR06], whose development support functionalities and user interface may be almost arbitrarily modified and extended by *plug-ins*. EMF is both a plug-in for Eclipse providing basic modeling functionalities and a framework for building tools extending the framework itself, e.g., by allowing to define new modeling languages or support tools.

At the heart of EMF lies the *Ecore metamodel*, which complies to *Essential MOF* (EMOF); this in turn is a subset of the language MOF (see Section 3.3). Ecore provides the metametamodel (M3, cf. Figure 3.4) for EMF-based applications. An editor for Ecore models is provided—instances are represented externally as *class diagrams*. Depending on the mode of usage, Ecore models may define either modeling languages situated at level M2 or domain models residing at level M1.

The framework provides a built-in source code generator for transforming Ecore models to Java source code. Since EMF is a purely structural modeling language, manual adaptations to the generated source code are necessary in order to add behavior, which is encapsulated in Ecore operations that are eventually transformed into Java methods. In particular, the added

3 http://www.eclipse.org/

behavior may describe the manipulation of M1-level model instances, which are internally represented as Java run-time objects, such that the boundary between M1 and M0 is blurred.

3.5.1 Ecore Metamodel

Since it is extensively used in this thesis, the Ecore metamodel is further detailed. Figure 3.6 shows a cut-out reduced to the most relevant elements.

The root element of each Ecore model is an EPackage, which may contain nested packages and instances of subclasses of EClassifier. There are three types of classifiers defined. Class EDataType represents basic types such as EString, EInt, and EBoolean. A special case of data types is *enumerations* (EEnum), which contain a fixed set of literals denoting the valid values of the data type.

Being an object-oriented modeling language, Ecore class diagrams are built around *classes*, which are represented as instances of EClass. The reference eSuperTypes defines a multiple inheritance relationship between classes. If a class is abstract, it must not be instantiated. The attribute interface affects EMF's built-in Java code generator. Normally, a pair of class and interface are generated, but if interface is set to true, no class is generated. Since multiple inheritance is not allowed in Java, it is broken down to multiple interface realization. One non-interface class may be chosen[4] as implementation superclass.

Classes are compound data types; their objects may carry individual values for different *structural features*, which are disjointly divided up into *attributes* – instances of EAttribute – and *references*—instances of EReference. From their common superclasses, structural

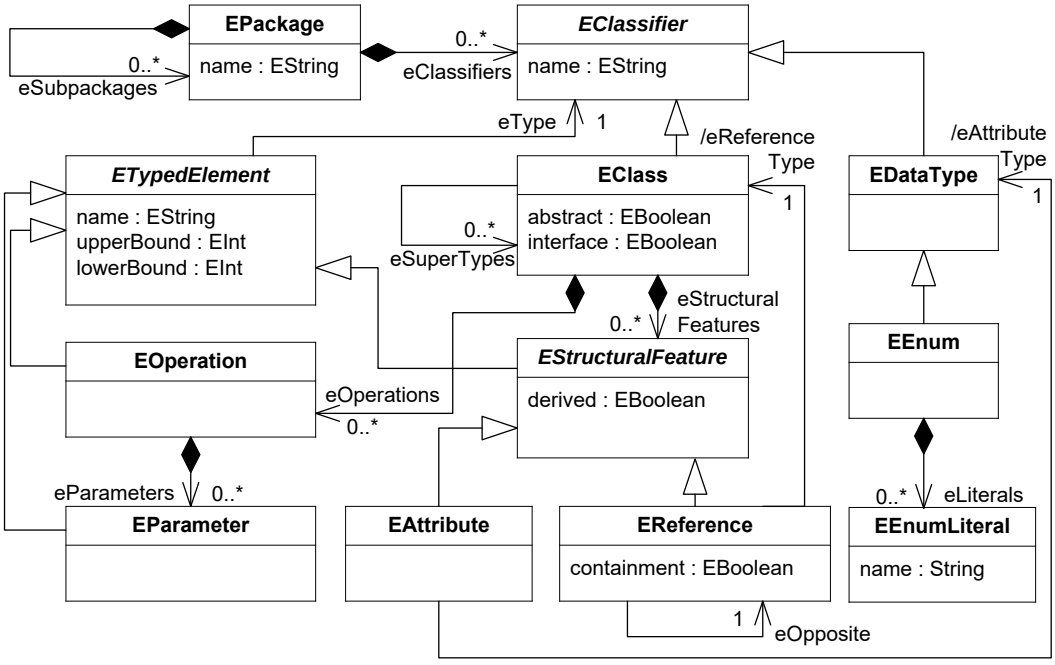


Figure 3.6: A cut-out of the Ecore metamodel represented as class diagram.

4 Actually, the first non-interface element in the collection realizing the reference eSuperTypes is selected.

features inherit a *name*, a *multiplicity* (consisting of lowerBound and upperBound), and a *type*, which is realized as a reference to EClassifier. As hinted by the corresponding subclasses, the eType is restricted as follows: attributes have as type a basic data type or an enumeration, whereas references end in a class. Last, both attributes and references can optionally be derived, i.e., there is a computation rule for values of its instances. In concrete syntax, this is expressed by a leading slash (/).

There are two additional features of references not discussed yet: *containment* and *bidirectionality*. First, references can have the attribute containment set to true. Then, the following properties apply to *instances* of this reference [Ste+09]:

1. *Existential dependency*: The contained object is part of its container. In case the latter is removed from the model, the former must not exist any further.
2. *Unique container*: Each object may have at most one container.
3. *Acyclic containment*: An object must not (transitively) contain itself.

In concrete syntax, containment references are expressed by a black diamond at the container end. Another optional feature of references is *bidirectionality*. Normally in Ecore, references are assumed to be navigable along one direction only. Bidirectional references are realized by a pair of references, mutually referring to each other as eOpposite. An additional constraint requires that the container class and the eReferenceType of opposite references be reciprocally identical.

3.5.2 Code Generation and Implementation of Behavior

The last element of the Ecore metamodel to be explained is *operations*, which are contained by classes. From its superclasses, EOperation inherits its name, its return type, as well as the multiplicity (of its returned value). Operations may also declare a sequence of *parameters* having the same structural features. Ecore does not define classes for the description of the dynamic behavior of operations. Instead, they are converted into method stubs during code generation. Operation bodies must then be implemented manually in Java. In order to control technical code generation properties such as the identifiers of generated Eclipse projects, a *generator model* is provided by EMF as an intermediate artifact.

3.5.3 EMF's Resource Framework

The default interchange format for EMF models and their metamodels is XMI (see Section 3.4). To both end users and application programmers, serialization is made transparent by the *resource framework* [Ste+09], which abstracts from the internal physical representation by offering the concept of *resources*, containers for EMF objects.

Figure 3.7 illustrates different kinds of relationships between objects contained in resources. Each resource corresponds to a file whose location is described by a *uniform resource locator* (URL). Resources may contain any kind of EMF models, i.e., metamodels (instances of Ecore), or models (instances of metamodels). In the example, two metamodels and three models are defined, each being contained by its own resource. Technically, a resource contains one or several *root objects*, which in turn build up a spanning containment

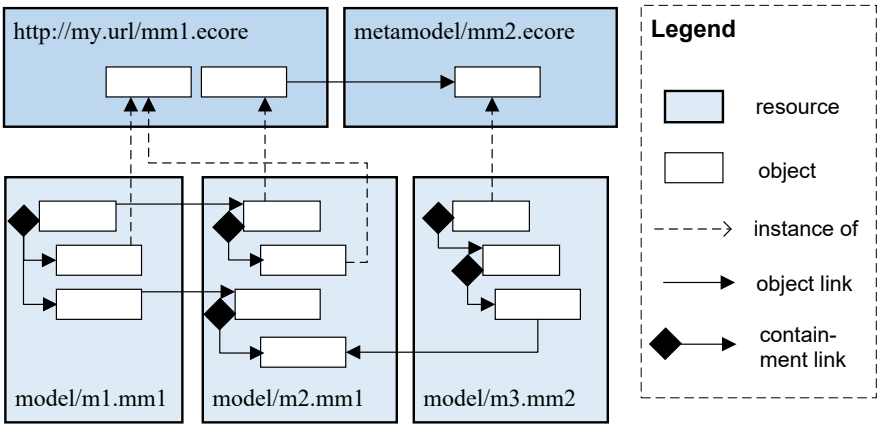


Figure 3.7: Possible relationships between objects located in different EMF resources.

tree of objects. Furthermore, *cross-resource links* are allowed only when having been instantiated from non-containment references.

3.5.4 Dynamic EMF, Reflective API, and Generic Tools

In the remarks above, it was implicitly assumed that Ecore models will eventually be transformed into Java source code. As a lightweight alternative, EMF offers a *dynamic* approach that does not require to generate any code. Rather than this, model objects are represented as instances of a dynamic class, DynamicEObject.

For both dynamic and generated classes, EMF provides a *reflective API* by which the state of objects may be dynamically read and modified. Furthermore, operations (of generated classes) may be dynamically invoked and *metadata*, such as the EClass of an object or the resource of a model, may be queried.

For tool developers, the reflective API constitutes an important point of contact. Tools can provide functionality for any kinds of EMF models without having to know the exact classes used, which are defined in the metamodel. Based upon this, *generic support tools* for general model management may be developed and used in a generic way. SuperMod is an example of such a tool.

3.6 Graphical and Textual Syntax

After having discussed the internal representation of models in general and of those conforming to the Eclipse Modeling Framework in particular, let us consider several different approaches to the *external* representation of models to the user, who wishes to create and modify models based upon their concrete syntax. In the context of EMF, three state-of-the art technologies are surveyed below. First, EMF contains a built-in *tree editor*. Second, the *Graphical Modeling Framework* (GMF) extends EMF by tools for the development of concrete graphical syntax for modeling languages. Last, using the framework *Xtext*, DSML developers may define textual syntax based on parsers which are generated from a grammar.

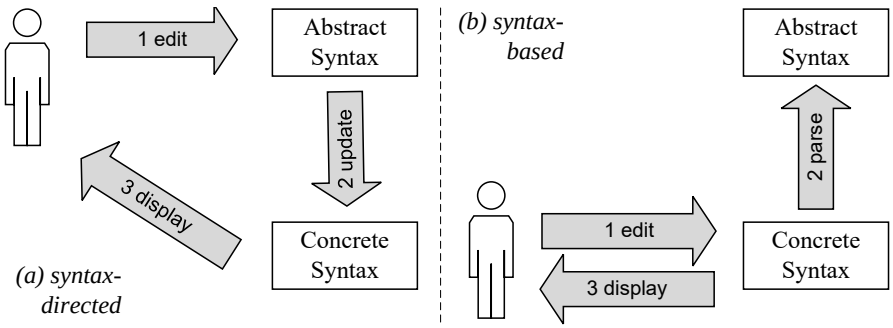


Figure 3.8: Syntax-directed vs. syntax-based editing.

Regardless of the framework used and the type of concrete representation (graphical or textual), there exist two distinct approaches to modification of models presented in concrete syntax (see Figure 3.8). On the one hand, in the case of *syntax-directed* editing, the model is persisted based on its *abstract syntax*. A *projection* of the model is made visible to the user, who may modify the model's abstract syntax using specific *commands* such as "create element". After editing, the projection is updated such that the user immediately sees the performed changes. On the other hand, *syntax-based* editing assumes that the model is persisted in *concrete syntax*. The user may freely edit this model. After editing, it is *parsed* in order to derive the abstract syntax in the background as a temporary data structure.

3.6.1 EMF Edit as Foundation

EMF supports both syntax-based and syntax-directed editing. In order to ease the implementation of different concrete syntax technologies, the *EMF Edit* framework is provided as a model manipulation and representation API independent of concrete user interfaces. The core concept of EMF Edit is *item providers*, which describe in an abstract way which operations are available for the creation and modification of objects and their properties. Furthermore, display properties such as *icon* or *label* can be defined in a reusable way. Modifications to the default appearance and behavior may be made either in the *generator model* or in the *Edit plug-in* source code generated for the respective metamodel. The concrete syntax frameworks presented below reuse the EMF Edit framework.

3.6.2 EMF Tree Editor

EMF includes a built-in default editor for models, the *tree editor*. It directly builds upon the EMF Edit framework. Objects part of a model are represented in a tree reflecting the containment hierarchy. Operations for object creation and manipulation are available as context menu entries. Details of objects may be modified in a *properties* view. When following the generative approach, the EMF tree editor may be customized by extending or restricting the set of available edit operations.

Although the EMF tree editor can be easily generated for existing Ecore models, it is considered to have limited use for concrete syntax editing since the provided tree syntax is still "too abstract". This is especially true for graph-like models, which are better

interpretable in a diagram syntax. On the contrary, for tree-like models, a customized tree editor may be quite comfortable to use. Therefore, this approach is used for SuperMod's *feature model editor*, which is described in Section 14.2.2, as well as for an example DSML in the evaluation (see Section 15.3.3).

3.6.3 Graphical Modeling Framework

The *Graphical Modeling Framework*[5] (*GMF*) is an EMF extension allowing to generatively develop a *syntax-directed* (cf. Figure 3.8) *graphical* editor for instances of a particular Ecore-based metamodel. Technically, it combines EMF with the *Graphical Editing Framework* (*GEF*), a toolkit for the development of diagram editors [Gro09].

By itself, GMF follows a model-driven approach, meaning that tool developers are intended to create different kinds of models to define the graphical syntax, editing facilities, as well as the connection between abstract and concrete syntax. These different tasks are orchestrated by the following models:

- The *domain model* corresponds to the Ecore model that defines the abstract syntax of the graphical modeling language to be developed (cf. Section 3.5). Using EMF's built-in *generator model* facilities, *model* and *edit* code can be derived.
- In a *graphical definition model*, the tool developer defines graphical modeling elements. Their visual appearance is defined in a so called *figure gallery*. There exist two important types of graphical elements: nodes (e.g., boxes or circles) and connections. For both nodes and connections, styling properties such as icon, fonts, colors, or line widths can be adjusted. Furthermore, nesting may be defined for specific node types.
- The *tooling definition model* defines the creation tools available in the tool palette. There is a distinction between node and edge creation tools.
- In the *mapping model*, the editing behavior is defined by combining elements of the domain model, the graphical definition model, and the tooling model, by means of different types of *mappings*. Depending on whether an object or a reference defined in the domain model shall be mapped to the canvas, to a node, or to a connection figure, different types of mappings can be defined. Furthermore, each mapping may refer to a creation tool.
- The *GMF generator model* is an intermediate artifact where generation parameters can be adjusted. The *diagram code* generated by this model depends on the model and edit code generated by the domain generator model.

By intention, the GMF run-time separates concrete from abstract syntax. In particular, for each domain model, represented as an instance of its Ecore-based metamodel, an arbitrary number of diagrams may exist. These diagrams contain references to the domain model, augmenting them with details referring to their graphical representation (e.g., position and size on the diagram canvas). Diagram models are instances of the *GMF runtime* metamodel. Therefore, from the perspective of tool support, GMF-based diagrams must be treated as a set of interconnected model instances conforming to different metamodels.

The UML modeling tool *Valkyrie*, which is used for many examples here, has been developed upon GMF [Buc12].

5 http://www.eclipse.org/modeling/gmp/

3.6.4 Xtext

The Xtext framework[6], which is available as a plug-in for EMF, provides support for defining concrete textual syntax for EMF models relying on a *syntax-based* (cf. Figure 3.8) editor that extends the built-in Eclipse text editor.

The language definition workflow realized by Xtext is centered around an augmented form of *context-free grammars* (CFGs) [HMU06], a formalism borrowed from compiler construction [Aho+06]. An Xtext grammar comprises a set of *grammar rules*, whose left-hand side refers to a class from the Ecore model that defines the abstract syntax. The corresponding right-hand side may contain non-terminals, e.g., keywords or placeholders of a specific data types, or terminals, which correspond to nested rule calls.

In general, a parser analyzes an input text and converts it into an *abstract syntax tree* (AST) [Aho+06]. In the case of Xtext, the AST is an instance of the underlying Ecore model. Therefore, when compared to ordinary CFGs, an Xtext grammar contain references to the metamodel that explain how the parsed contents are mapped to the model instance.

In contrast to GMF, where the concrete and the abstract syntax are persisted independently, Xtext-based models are persisted in their concrete syntax, i.e., in textual form. The abstract syntax is derived on demand by parsing the underlying text file. Nevertheless, Xtext-based models integrate seamlessly with other EMF models since Xtext implements EMF's resource framework (cf. Section 3.5).

3.7 Model Transformations

As introduced so far, models are capable of capturing information, which is exposed to the user based on different kinds of concrete syntax. Using specialized editors, stakeholders may modify and share models. The real potential of MDSE, however, is obtained by *model transformations*, which have been called "the heart and soul of model-driven engineering" [SK03]. On the one hand, they may *convert* models into less abstract representations and, eventually, into source code. On the other hand, model transformations are used in order to describe the intrinsic *behavior* of structural models. Therefore, some model transformations may be considered as behavioral models as well.

3.7.1 Classification

As shown in Figure 3.9, there exist several classification dimensions for model transformations, details of which are outlined below. In the sequel, references to concrete EMF-based technologies are made.

Output. Model transformations take as input a model that conforms to a MOF-like modeling paradigm, e.g., an instance of an Ecore-based metamodel. We can distinguish between *model-to-model* (M2M) and *model-to-text* (M2T) transformations according to the transformation output. In the latter case, the produced text may represent, e.g., source code or configuration files.

6 http://www.eclipse.org/Xtext/

Endogenous vs. Exogenous. A model-to-model transformation is called *endogenous* in case the output model conforms to the same metamodel as the input model. *Exogenous* model transformations produce an artifact of a different type, i.e., either text or a model conforming to a different metamodel.

In-Place vs. Out-Place. In an *in-place* model-to-model transformation, the input model is equivalent to the output model. More precisely, the transformation describes a *side effect* on the input model. In contrast, *out-place* transformations produce a new model (or text), leaving the input model unmodified. In-place transformations are endogenous by definition.

Syntax. Two different approaches for the notation of model transformations exist. On the one hand, a model transformation may be specified in *textual* form. Alternatively, there exist a number of approaches relying on a *graphical* syntax, the greater part of which have their origins in the theory of graph transformations [Roz97].

Paradigm. In analogy to programming languages, which are assigned to different paradigms such as imperative, procedural, or functional, the style of transformation specification dictated by a model transformation language may vary. Two different paradigms have emerged: While *imperative* languages consider a model transformation as a sequence of statements, each mutating the state of the output model, in *declarative* languages, the expected result is formalized using suitable abstractions. Since both approaches imply desirable properties, in practice, mixed forms are often employed.

Unidirectional vs. Bidirectional. Transformation languages can be distinguished into *unidirectional* and *bidirectional*. A unidirectional language exclusively describes a transformation from a dedicated *source* to a *target* model. Bidirectional transformations, in contrast, contain a single specification, from which both execution directions can be derived automatically. Bidirectional transformations are of particular interest in round-trip engineering

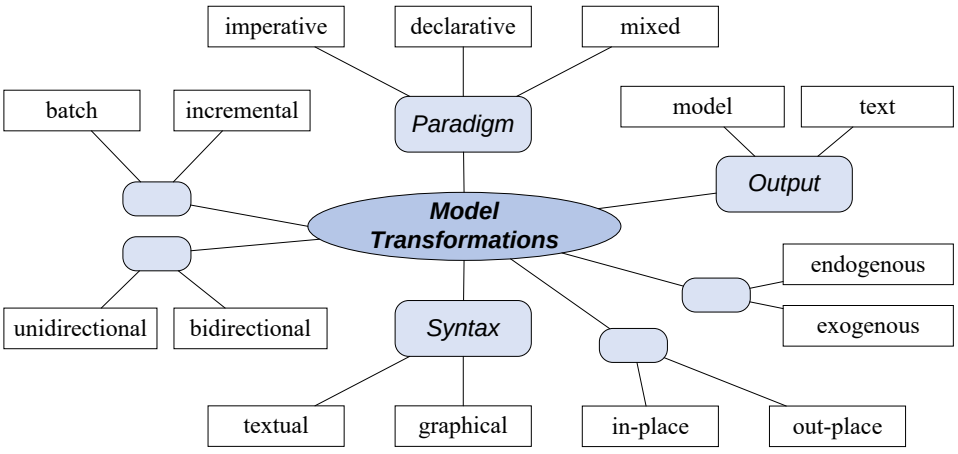


Figure 3.9: Classification dimensions of model transformations.

scenarios, where source and target model co-evolve. Languages that enable bidirectional transformations tend to follow the declarative paradigm.

Batch vs. Incremental. The last classification dimension refers to *incremental* transformations. They assume that the source model is not only transformed once but repeatedly. Furthermore, the target model should not be entirely created anew, but only those parts of the model that changed in the source model should be propagated to the target model. Like bidirectionality, this is a frequent requirement in round-trip scenarios. For instance, modifications to generated source code should be maintained when re-invoking a M2T-based code generator. Incremental transformations usually rely on a *trace* that records which elements have already been created in order to avoid their repeated creation.

Special Kinds of Model Transformation. For the sake of completeness, two special kinds of model transformations are mentioned. First, a *higher-order model transformation* is a model transformation that takes a M2M or M2T specification as input and/or produces such a transformation as output [Tis+09]. Second, *multi-variant model transformations* assume that the input model is enriched with *variability information* that is supposed to be transferred to the output model in an adequate way [SBW16b].

3.7.2 Technical Solutions based on EMF

In the orbit of the Eclipse Modeling Framework, several distinct solutions to model transformations have been invented, each having individual properties and purposes. The following list summarizes only a small subset.

- With *MOFM2T* (MOF Model to Text), the OMG have issued a standard [OMG08] for M2T transformations of models based on MOF. The suggested template-based textual language is imperative and unidirectional. The de-facto standard implementation of MOFM2T is the EMF-based framework *Acceleo*[7]. Using the concept of *protected regions*, incremental transformations can be realized as well.
- The *Atlas Transformation Language*[8] (ATL) is considered as one of the most frequently used M2M languages. It supports a variety of scenarios, including both endogenous and exogenous, in-place and out-place transformations [Jou+08]. ATL also offers a *refining mode* that supports incremental transformations but is limited to in-place transformations—the output model is a refined copy of the input model. ATL provides a textual syntax; the offered modeling constructs mix the imperative and declarative paradigms. Bidirectional transformations are not supported by ATL.
- With *QVT* (Queries/Views/Transformations) [OMG16], a set of standards for M2M transformations has been established by the OMG. QVT includes two languages which are mapped to the same core language that actually executes the transformation(s). First, *QVT operational* is a textual, imperative language supporting unidirectional transformations in

7 https://eclipse.org/acceleo/
8 http://www.eclipse.org/atl/

batch mode. *QVT relational* follows a declarative paradigm, allowing for incremental transformation execution. Furthermore, both a textual and a graphical syntax are defined. At the heart of QVT relational are *relations* that describe correspondences between instances of the models to be mutually converted into each other. From these specifications, both execution directions are derived, thus realizing bidirectional transformations. Both the operational and the relational language variant support in-place and out-place as well as endogenous and exogenous transformations. At present, only partial EMF-based tool support is available: The Eclipse *QVTo* project[9] for QVT operational, and *medini QVT*[10] for QVT relational.

– The theory underlying model transformations has its origins in the field of *graph transformations* and *graph grammars* [Roz97]. Many current research activities are dedicated to the adaptation of graph theory to M2M, resulting in EMF-based tools offering graphical syntax for the declarative specification of different kinds of M2M transformations. For instance, the formalism of *triple graph grammars* [Schü94] has been realized by the tool *eMoflon*[11], which offers bidirectional and incremental execution modes for out-place model transformations.

– Although dedicated M2M and M2T languages allow to specify transformations on an adequate level of abstraction, probably the most widespread implementation language is *Java*. This may be due to the immaturity of the M2M/M2T tools currently available, but also due to performance reasons or specific requirements for transformations not being covered by any available approach. Recently, the Xtext-based language *Xtend*[12] became popular, which offers abstraction mechanisms particularly useful in the M2M context [BG16].

3.8 Bottom Line

EMF, which has been the subject of the former sections of this chapter, realizes a minimalistic and pragmatic approach, based on which many goals formulated under the set of standards of *model-driven architecture* may be achieved.

With the UML, a widely accepted *general purpose modeling language* is available. However, its complexity – which is expressed by 14 diagram types being available and several languages added on top, e.g., OCL and Alf – also complicates modeling. MOF's three-layered language architecture, which has been realized by EMF, enables the definition of *domain-specific modeling languages*. We have also learned that adequate technical solutions exist for the definition of concrete graphical and textual syntax for EMF-based modeling languages. Moreover, EMF's *reflective API* allows to build generic tools applicable to models conforming to arbitrary EMF-based metamodels.

The long-term success of model-driven software engineering depends on good tool support not only for core processes but also for support disciplines such as deployment, validation, and, in particular, *configuration management*. Adequate tools must be aware of both *evolution* and *variability*. These topics are considered in the next two sections independently, before their integration with MDSE is surveyed in Chapter 6.

9 https://wiki.eclipse.org/QVTo
10 http://projects.ikv.de/qvt
11 https://emoflon.github.io/
12 http://www.eclipse.org/xtend/

In software development,
nothing is as persistent as change.

ANDREAS ZELLER (1997)

Chapter 4

Software Configuration Management and Version Control

Abstract

Software configuration management (SCM) is an important support process in software engineering spreading over all development activities. This chapter starts with an introduction of the big picture of SCM, before the sub-discipline version control (VC) is explored with respect to the functionality exposed to the user as well as to the internals of version control systems. Another important aspect of SCM in general and VC in particular is collaboration, which can be orchestrated either in an optimistic or in a pessimistic fashion. In recent years, distributed version control systems have gained popularity. The chapter concludes with an outlook on how far variation control, i.e., the support for developing different product variants based on a decomposition into configuration options, is feasible using state-of-the-art version control approaches.

Contents

4.1 Functionalities of Software Configuration Management

Model-driven software engineering – see previous chapter – and software product line engineering – see next chapter – can be considered as special development approaches which a specific project may or may not follow. On the contrary, *software configuration management* (SCM) has become an indispensable discipline part of almost every development activity from the beginning of software engineering.

In [IEEE05, p. iii], it is stated that "SCM is a formal engineering discipline that [...] provides the methods and tools to identify and control the software throughout its development and use. [...] SCM is the means through which the integrity and traceability of the software system are recorded, communicated, and controlled during both development and maintenance".

With the constantly growing size of software projects and teams, *collaboration* has become a more and more important aspect of SCM. The requirements towards SCM are further detailed by defining the following *functionality areas* [Dar91] (cf. Figure 4.1):

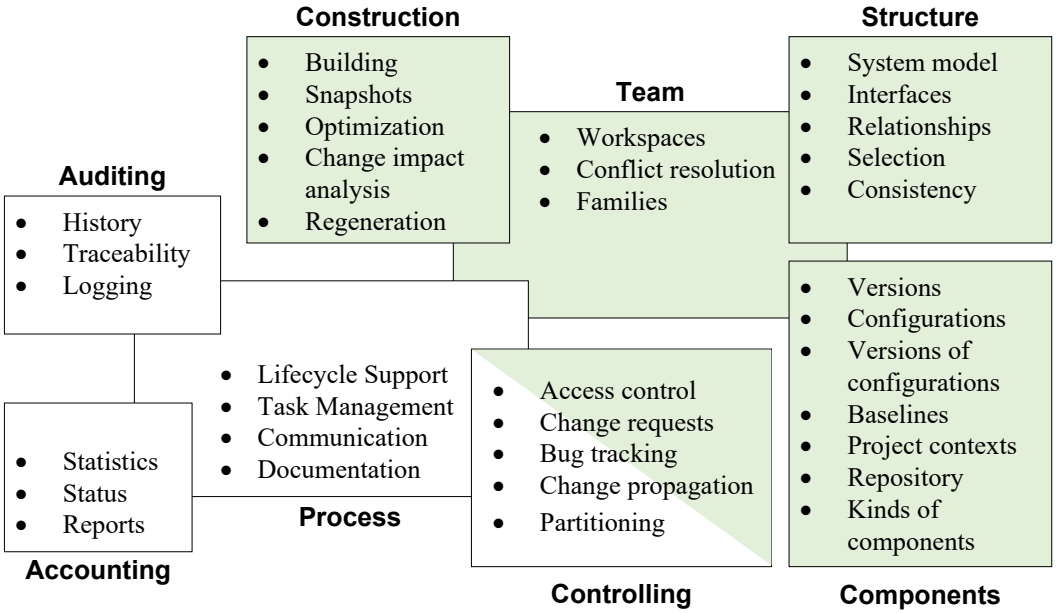


Figure 4.1: SCM functionalities according to [Dar91]. Green parts refer to SCM as *development support* discipline, white parts to its *management* facet.

Components. All components part of the software need to be identified unambiguously. Furthermore, it must be possible to comprehend the differences between multiple versions of the same component, as well as the reasons for the creation of those. This area also covers the storage and the access of software components.

Structure. This functionality area considers the relationships between different components of the system, whose consistency must be ensured.

Construction. Once a configuration has been consistently defined by its structure and components, SCM must support a comprehensible, repeatable and mostly automated compilation procedure.

Team. For collaborative software development, it is indispensable that developers may apply changes to different components concurrently and in isolation. The combination of individual changes must be coordinated.

Process. This functionality area ensures that development activities are aligned with the specific development process used throughout the project. The relationship between specific changes and specific tasks defined in the process must become clear.

Auditing and accounting. Both the development process and the developed software are subject to continuous validation. For this purpose, logs must be written, from which statistical reports may be derived.

Controlling. Access restrictions for individual components and versions thereof must be controlled. Furthermore, SCM requires concrete responsibilities for the management of change requests and bug reports.

Originally, the software engineering community had expected that there will eventually emerge an SCM tool to cover all the functionality areas presented above in an integrated tool. However, fully-featured SCM tools are rare; one representative is *Rational ClearCase*[1]. The reason for this is probably the fact that SCM actually is a hybrid of a *project management* and a *development support discipline* [CW98]. Therefore, different types of tools are needed to fulfill the diverse stakeholders' needs. In many publicly documented software engineering projects, especially in the open source community, we can discover a *triad of support tools* realizing SCM:

– *Project management tools* such as *Redmine*[2] or *Bugzilla*[3] help planning different tasks of a software project and allow to quantify the progress made. Furthermore, they can be used for managing change requests and for bug tracking, such that a subset of *controlling* functionalities is covered. Communication and documentation features, as defined in the *process* area, are supported. Many project management tools include dedicated *auditing* and *accounting* functions.

– *Build tools*, e.g., *Ant*[4] or *Maven*[5], go considerably beyond the functionality of automatic program compilation; they are used for deployment, regression tests, and other functionalities

1 http://www-03.ibm.com/software/products/en/clearcase
2 http://www.redmine.org/
3 https://www.bugzilla.org/
4 http://ant.apache.org/
5 https://maven.apache.org/

of the SCM area *construction*, and to a certain extent, *structure*. The results of regression tests also contribute to the management areas *auditing* and *accounting*.

– *Version control systems* (VCS) including the aforementioned *Git*[6] and *Subversion*[7] cover most functionalities of the areas *structure*, *components*, and *team*. Furthermore, the technical parts of *controlling*, e.g., access control and change propagation, are offered by VCS. Together with build tools, VCS serve SCM's purpose as a *development support discipline* [CW98]. VCS are a subject of the remainder of this chapter.

  As already hinted in Figure 4.1, the functionalities of the area *controlling* are shared between project management and version control tools. Typically, the integration of different development tools is achieved in a loosely coupled way in an integrated development environment.

4.2 Abstractions and Metaphors of Version Control Systems

Here, *version control* (VC) [CFP04] is considered as a development-centric sub-discipline of SCM. Furthermore, the *construction* functionality is faded out in the description below. Therefore, the most important *responsibilities* supported by a VCS are:

– To persistently store and identify different versions of components of the software project in a *repository*.
– To provide developers with temporary *working copies* they can modify and write back to the repository afterwards.
– To help developers *comprehend* the development history and to *plan* future development tasks.
– To *coordinate* changes applied by different developers consecutively or concurrently.
– To ensure the *consistency* of the software subject to evolution by imposing constraints on version selection.

4.2.1 Editing Models and their Metaphors

In order to achieve these tasks, three different *editing models* [Fei91] have been established; they expose individual *metaphors* to the user.

Check-Out/Check-In Model. The metaphors used in this model have already been introduced in Section 1.3.2 and in particular in Figure 1.3: Developers select a specific *version* of the software from the repository (*check-out*). This version is made available in a *workspace*, where the user may apply the intended modifications. Thereafter, changes are written back (*check-in*), which produces a new *version*. The history of consistent versions is immediately apparent to the developer (e.g., in the form of a *revision graph*), thus, *comprehension* and *consistency* are unproblematic. For the *coordination* of concurrent changes, *optimistic* and *pessimistic* strategies exist; see Section 4.4. The check-in/check-out model is applied by most contemporary VCS.

6 https://git-scm.com/
7 https://subversion.apache.org/

Change-Oriented Model. This model assumes that there is a *baseline* version of the versioned software, to which several *change sets* can be applied. Both baseline and change sets are persisted in a repository. For producing a specific version, a consistent collection of change sets are selected and applied to the baseline version. Evolutionary increments are realized by developers as change sets, which are incorporated to the repository upon finalization. Since change sets are developed in isolation, *coordination* is not explicitly supported by this approach. Moreover, *comprehension* of the version history becomes difficult with the growing number of change sets available.

Composition Model. The software repository makes software *components* explicitly available to the developer in the form of a *system model* in the original sense [Dar91]. This requires that the VCS implement dedicated knowledge about the used technologies, e.g., the programming language employed. Developers may evolve the contents of the repository by adding new components or by modifying the system model. Questions of *coordination*, *comprehension* and *planning*, as well as *consistency* depend on the concrete product model realization.

Notice that [Fei91] defines a fourth model, namely *long transactions*. It assumes that different developers may evolve their own copies of the repository, which are *synchronized* at dedicated points in time, in isolation. The support for long transactions is here considered as an orthogonal property that may be implemented upon all of the three version models described above. Nowadays, long transactions have been realized by *distributed version control systems* (e.g., by Git's operations *pull* and *push*; see Section 4.5).

Unless stated otherwise, we assume an editing model oriented towards *check-out/check-in* subsequently. The approach presented in this thesis also builds upon this model and, moreover, supports long transactions.

In Figure 4.2, a simple example for the usage of the check-out/check-in model is provided. A hypothetical user checks out a particular version of a specified software project from the

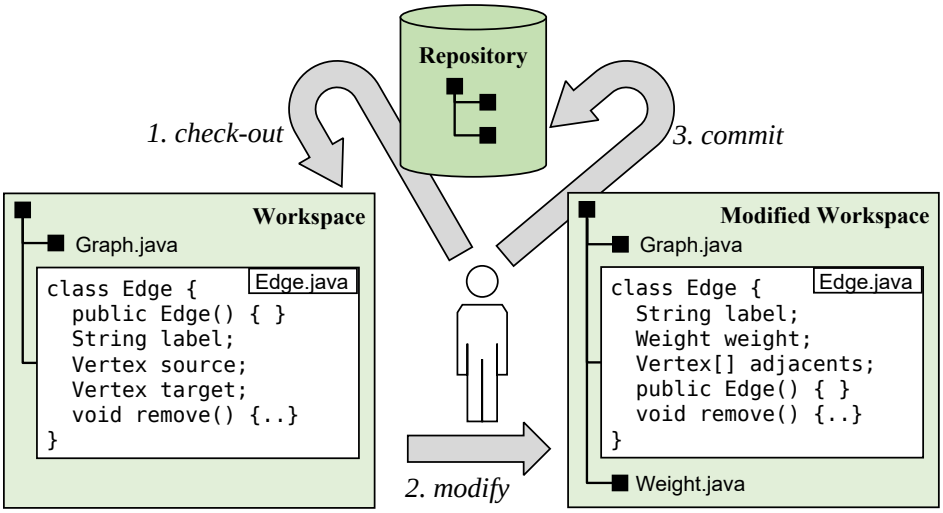


Figure 4.2: Example for the usage of the check-in/check-out editing model based on a Java source code file for the class `Edge`.

repository, resulting in his/her workspace being populated by several files. The user then performs modifications, including the addition of a new file `Weight.java` (whose contents are omitted) and the modification of file `Edge.java`. The modified workspace contents are then *checked-in* (also: *committed*) to the repository. This example is recaptured below for explaining the internals of version control systems.

4.2.2 Revision Graphs

The term *version*, which was referred to above, remains to be clearly delineated. According to [CW98, p. 238], a version is "a state of an evolving item", where *item* covers "anything that may be put under version control". In the literature, version is understood as a generalization of the terms *revision* and *variant*. The fact that software is subject to historical evolution is addressed by *revisions*, which "are ordered, newer revisions supersede previous ones" [Ap+13b, p. 100]. There exist different ways to organize revisions, which shall be discussed below. In contrast, *variants* describe intentionally co-existing instances of the versioned product. Variants are a subject of Section 4.6 and of Chapter 5.

It is natural to describe versions and their predecessor relationships using *directed graphs*, where an edge connects a predecessor to a successor. Since no revision can precede or supersede itself, revision graphs must be *acyclic*. Depending on the specific VCS, the structure of the version history may be constrained by concrete types of *revision graphs* (cf. Figure 4.3).

The most restrictive form of a revision graph is a totally ordered *sequence* of revisions (a). A VCS realizing this approach must ensure that, in case versioned items are modified concurrently, the most recent revision does not override revisions remotely created in the meantime. Subversion follows this approach, applying optimistic synchronization (see below). A *tree* of revisions (b) allows the creation of successors for non-leaf versions. Yet, the number of branches will never be reduced. VCS that support *merging* different versions imply an *acyclic* revision graph (c). This type of graph has been implemented in many contemporary VCS including Git. Moreover, *two-level* revision graphs (d) provide for a better overview by subsuming sequences – or trees, or acyclic graphs – of revisions in explicit *branches*. Different *naming schemes* exist for revisions in two-level revision graphs.

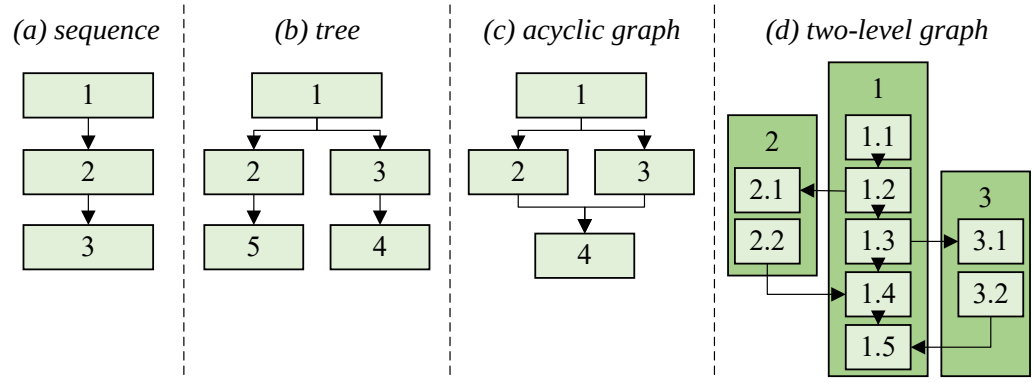


Figure 4.3: Different forms of revision graphs supported by VCS. Based on [CW98, Figures 3, 4].

In the example in Figure 4.3, top-level revisions are enumerated after branch creation.

The purpose of revision graphs goes beyond capturing the relationships between different versions. Typically, several additional *properties* are assigned to specific revisions. These include a *time stamp* of the commit date, the committing *user*, as well as a manually created *commit message*, which should describe the intention behind the change as well as internal details. This way, one part of the *controlling* functionality (cf. Section 4.1) is covered.

4.2.3 Product Space Concepts

Above, we have repeatedly used notions like "software project", "versioned software", or simply "product", in order to denote the entirety of items being under version control. In fact, the structure of the *product space* varies among different representatives of VCS.

Most of the practically established VCS assume the product space to be a hierarchy of directories of folders and files, i.e., simply a subset of a file system. Furthermore, the *software objects* are text files, such that each version of a text file has its individual contents. Differences between different versions of the same file are captured and represented in a *line-oriented* way, such that *editing operations* such as line insertion, line modification, or line deletion are instantiated. This type of product space organization was tacitly assumed in the example in Figure 4.2.

More generally, the organization of the product space, including properties of and possible relationships between different software objects, is dictated by the *product model* employed in the respective VCS. In [CW98], the following properties and relationships are defined:

Object Identifier. Each versioned software object must be uniquely identifiable in the context of the whole product space. A *sameness criterion* can be provided, e.g., by an *object identifier* (OID) attached to each software object. The OID may be system-generated (e.g., an automatically generated *universally unique identifier*), or user-generated (e.g., derived from the file name).

Object Granularity. A software object may be composed of *fine-grained* units internally. For instance, a text file consists of a sequence of text lines. These fine-grained units are typically used in order to provide the user of VCS with the possibility of a detailed comparison of the object contents. Furthermore, memory-optimized storage of multiple object versions can be achieved.

Composition Relationships. *Coarse-grained* composition relationships between different software objects form a hierarchy of objects. For instance, a versioned folder consists of different files. This kind of relationship may be used to define, e.g., *modules*, versions of which consists of different (versions of) source code files. Furthermore, this enables for *product selection*. For instance, in Subversion, the user may decide to check-out only a sub-tree of a versioned software project. In addition, "a composite object may act as a unit with respect to structural operations (e.g., copy or delete), [and] concurrency control" [CW98, p. 236].

Dependency Relationships. Orthogonal to composition relationships are *dependencies* between different software objects. They are used in order to guarantee consistency, in particular between *derived* and source objects. For example, a compiled program depends on its originating source code files. Both Subversion and Git do not explicitly

keep track of dependency relationships, yet it is possible to *ignore* derived resources in order to build them consistently and automatically after each check-out.

4.3 Internal Concepts of Version Control Systems

After having seen VCS through the user's perspective, let us now consider the concepts, algorithms, and data structures realized internally. For the explanation of many details, the systems Git [Cha09] and Subversion [CFP04] have been selected. This is due to the fact that *SuperMod* borrows realization concepts from both of these systems.

4.3.1 Delta Storage

As seen from the user's perspective, a version control system manages different versions of software objects. These versions must be made permanently available, such that the user may recover previous states at any time. When considering the version history of one software object in isolation, a straightforward possibility would be to persist all committed versions in isolation. Albeit, the description above has already pointed out that software objects may be further decomposed into fine-grained units. The overall composition of different versions of a software object may differ in only few units having been added or removed in a single evolution step — this property can be exploited for memory-optimized storage. Below, we assume *text-oriented* versioning, where a software object corresponds to a text file, whose fine-grained units are text lines. Since the line order is crucial, this implies an additional *sequencing problem*.

In practical applications, three different forms of delta storage have emerged: *snapshots*, *symmetric deltas*, and *directed deltas*. Figure 4.4 contrasts these strategies using the text file `Edge.java` of the example introduced in Figure 4.2, assuming that the original version carries the version identifier v_1, whereas the committed version is v_2.

Snapshots. Technically, this is the simplest yet the most storage consuming strategy. On commit, in case the state of a file differs with respect to its predecessor, its entire contents are captured in a *snapshot*, which is transparently written into a new file in the repository. This approach is realized in *Git*, rendering it a "mini file system with some incredibly powerful tools on top of it" [Cha09, p. 5].

Symmetric Deltas. Using this storage strategy, all versions of the text file are stored in a *multi-version* representation, where modifications to a *baseline* version are described in a way similar to *conditional compilation* (cf. Section 5.4.2). Thus, in the text file, content text lines are aggregated in blocks annotated with *insert* or *delete* conditions. These conditions refer to versions and apply transitively to successors. For instance, when referring to Figure 4.4b, the line `public Edge(){}` is deleted, and the line `Weight weight;` is inserted, in case version v_2 or one of its successors are selected. The multi-variant representation determines a fixed order for text lines. Symmetric deltas are seldom used in contemporary text-oriented VCS; an important representative is the *Source Code Control System* (SCCS) [Roc75].

Directed Deltas. The observation that many superseding revisions of text files differ with respect to only small *changes* leads to *directed deltas*. Rather than persisting two files

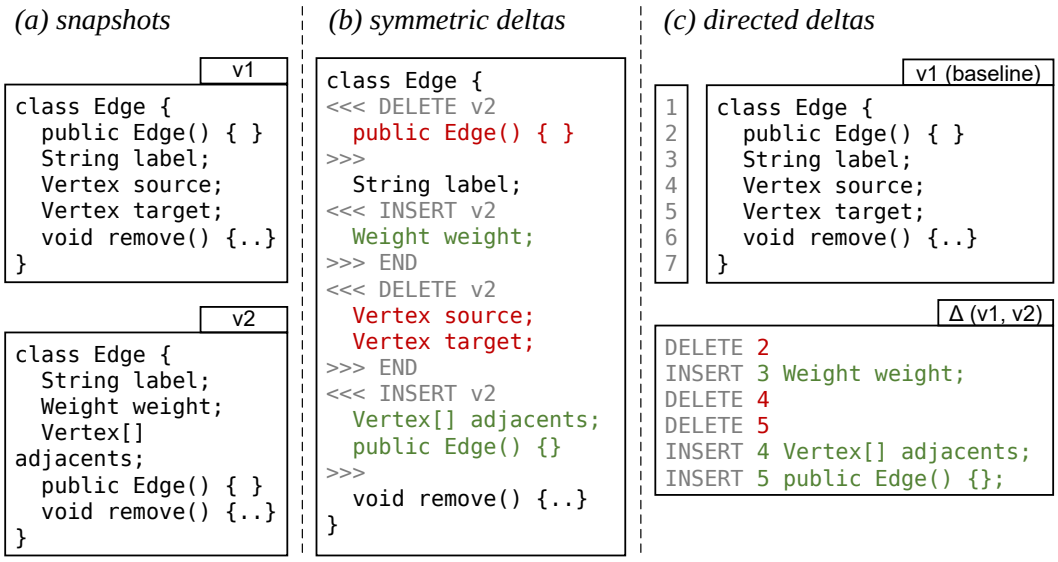


Figure 4.4: Snapshots, symmetric deltas, and directed deltas as storage strategies for versions.

independently, the same information is derived by storing only one *baseline* version along with a *change* that describes a sequence of operations to convert the first file into the second file. In the example in Figure 4.4c, we assumed two operations *insert*, taking the line number and the inserted text, and *delete*, taking only the line number as arguments, respectively. v_2 can be produced by applying the changes defined in $\Delta(v_1, v_2)$ to v_1.

This mechanism can be applied repeatedly to a sequence of n versions, resulting in one single baseline version and $n - 1$ deltas. Obtaining versions located at the end of the chain, though, becomes time consuming. This leads us to a discussion of different subtypes of directed deltas, which are contrasted in Figure 4.5.

(a) *Forward* deltas implement the straightforward approach of treating the initial vision of the history as baseline. When committing a new revision r_i, the delta $\Delta(r_{i-1}, r_i)$ is calculated. Assuming a linear history of n revisions, read access to the latest revision will cost $n - 1$ delta applications.

(b) *Backward* deltas rely on the assumption that more recent versions are accessed more frequently than older revisions. Thus, the *latest revision* serves as baseline. On commit of r_i, the former baseline is replaced by the committed version, and $\Delta(r_i, r_{i-1})$ is appended to the repository's internal storage. In addition, each *branch* obtains an own baseline for its latest revision. *Subversion* follows this approach [Dot11].

(c) *Intertwined* deltas are a mixed form of forward and backward deltas. For instance, in Figure 4.5c, the *trunk* is stored using backward deltas, offering quick access to its latest revision, but *branches*, which are supposed to be shorter, are persisted in a forward fashion. This omits the necessity for multiple baselines to be fully persisted.

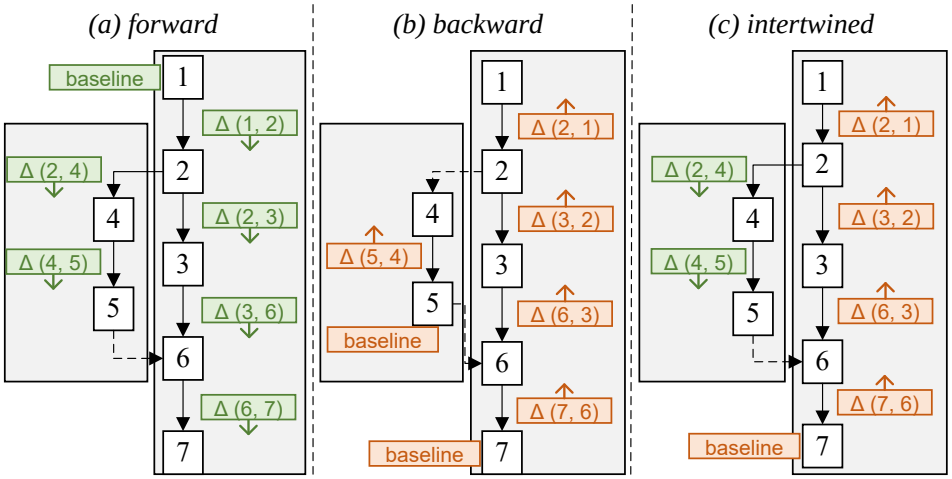


Figure 4.5: Composition of deltas in the revision graph: forward, backward, and intertwined.

4.3.2 Change Detection

All kinds of delta storage require the detection of *modifications* in the workspace in order to propagate them to the repository as soon as a commit is issued. There exist two fundamental approaches, *log-based* versus *comparison-based* versioning.

Log-Based Versioning. Using this approach, all editing commands carried out by the user in his/her workspace are *recorded*, resulting in an *edit log* that precisely captures and allows to reproduce the actual modifications performed in the workspace. From such a log, directed deltas can be easily derived. Recording all modifications requires a deep integration of the editing tool into the workspace. In the case of file-oriented versioning, it is hard to guarantee that every change to the file, regardless of which external editing program causes it, gets noticed. This is why, although promising the highest possible accuracy, log-based versioning is not commonly used in state-of-the-art (text based) VCS. A representative is *Rational ClearCase*[8].

Comparison-Based Versioning. Most of the contemporary VCS, including Git and Subversion, follow this lightweight approach. In contrast to log-based versioning, the edit log is not obtained by recording actually performed modifications, but from an a-posteriori *comparison* between the original and the modified state of each software object when the changes are committed. The applied comparison methods are typically *heuristic* and produce an edit log that does not necessarily reflect the actually carried out editing commands. In the context of text-oriented versioning, *sequence comparison algorithms*, to be discussed subsequently, are employed. By assuming line-wise comparison, their granularity with respect to the versioned object is typically coarse. For example, in case a variable is renamed in a Java program, comparison-based versioning will detect, e.g., the deletion and insertion of a line, since the exact equality of text lines is used as matching criterion.

8 http://www-03.ibm.com/software/products/en/clearcase

4.3.3 Sequence-Oriented Differencing

Let us consider the special case of comparison-based versioning of *sequence-oriented* software objects, i.e., text files as sequences of text lines, in greater detail. We can describe the difference between two versions as a sequence of *operations*, e.g., insertions or deletions, of elements at given positions. This also represents a derived edit log (cf. Section 4.3.2) and corresponds to a *directed delta* (cf. Section 4.3.1). Obtaining the differences involves two distinct steps:

Matching. First, the two versions of the sequence are compared in order to find their *commonalities*, i.e., pairs of elements that match with respect to a given *sameness criterion*. In the case of text files, most VCS assume two lines as matching only in case their contents are identical. There exist a multitude of *sequence comparison algorithms*, each having its individual properties that can be asserted to the produced result. Here, we consider two of them representatively.

– A family of algorithms is based on the *longest common subsequence* (LCS) of the input sequences. A common subsequence is an ordered list of elements that appear in the same order in both sequences. Correspondingly, an LCS is a common subsequence with the maximum number of elements possible. In general, the LCS may be ambiguous, e.g., in the example shown in the top left part of Figure 4.6, we may in fact find two valid LCS.

  An important property of LCS-based matching is that the results do not contain cross-over matches, whose relative position would contradict between the two versions and therefore destroy the common subsequence property. Many algorithms have been designed and implemented, each of which suits with different special cases of matching problems. A prominent example is *Hunt's fast LCS algorithm* [HS77].

– *Heckel's Algorithm* [Hec78] is based on the observation that many source code files contain

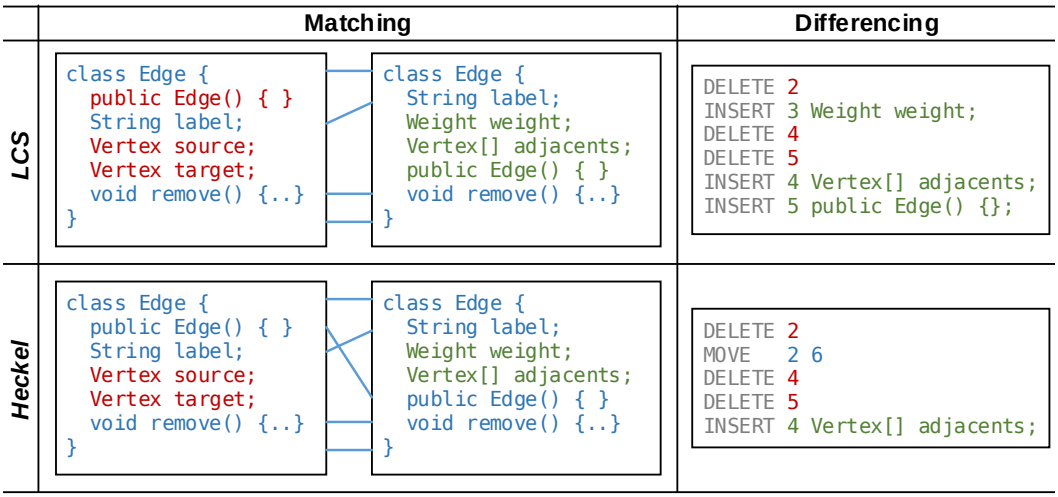


Figure 4.6: Sequence matching and differencing based on the LCS property and Heckel's Algorithm, illustrated using the example of `Edge.java`.

unique lines of text, e.g., method declarations whose signature must be unambiguous within a class. The algorithm follows a two-stage procedure. First, all elements that are *unique* in both input sequences are considered as matching. Second, for each matching element, it is checked whether their direct predecessors and successors are mutually identical. If this is true, the list of found matches is expanded accordingly. The second step is repeated iteratively for each matching until no more new correspondence can be found. Heckel's Algorithm may identify cross-over matches. Furthermore, the *heuristically* produced result may be suboptimal in that they disrespect formal criteria such as the LCS property.

Differencing. The identified matching is used as input for the subsequent difference computation. Intuitively, all elements in the original version that have no corresponding element in the modified version can be considered as deletions — in the opposite case an insertion can be deduced. In addition, the *order* of elements may have changed in case cross-over moves are allowed by the underlying matching algorithm. To be meaningful to both the user and the VCS (e.g., for delta application), the detected operations must be *parameterized* in a suitable way. For instance, the right-hand column of Figure 4.6 encodes deletions using the index of the deleted text line as parameter; insertions require the inserted text in addition. *Moves*, which are deduced from matchings created by Heckel's Algorithm – since cross-over correspondences are allowed there –, require the line indexes indicating the original and the modified location.

The change encoding used in the examples is arbitrary and has been optimized for illustration purposes. In practice, more efficient and less space consuming internal representations are used. In particular, adjacent move operations may be combined to *block moves* [Tic84].

As mentioned before, *Git* uses snapshots for storing different versions of a file. Therefore, differencing is not required internally for delta storage, but merely externally in order to present *graphical difference reports* to the user on demand. For this purpose, Git relies on an implementation that produces an LCS [Cha09]. In contrast, *Subversion* uses an optimized algorithm [Wu+90] for LCS comparison both for storage and for diff presentation. A generalized version of Heckel's Algorithm has been implemented in *SuperMod*—see Section 10.3.4.

4.4 Collaboration

For the description of the conceptual and internal details of VCS provided above, we have tacitly assumed a single-user set-up. According to the here considered SCM functionality areas (cf. Section 4.1), the *team* aspect remains to be considered. We must assume that several developers, each having his/her individual workspace, are involved in a software project, concurrently delivering changes to the code basis organized in the repository. To support multiple users, concurrent changes must be orchestrated and potential conflicts need to be resolved in a meaningful and comprehensible way.

The most important requirement for collaborative version control is a physical separation of repository and workspaces, such that the repository is running on a server, which is connected via network to individual client machines where the workspaces are managed. This immediately leads to a stack of coordination problems [Bab86]:

Duplicate Maintenance. Several copies of each software object need to be maintained. In order to avoid multiple divergent copies, it is required to propagate changes among the copies.

Shared Data. The duplicate maintenance problem may be solved by introducing a *master* copy, e.g., in the repository. Then, however, developers may happen to interfere with each other when accessing and modifying the master copy of some software object.

Simultaneous Update. The solution of the shared data problem is the introduction of individual workspaces. Changes are propagated back (i.e., committed) to the master copy after completion. Then, multiple commits mutually invalidating each other may occur in case transactions overlap.

There are two classes of solutions to the simultaneous update problem, *pessimistic* and *optimistic* synchronization, which are considered in Sections 4.4.1 and 4.4.2, respectively. Optimistic strategies may engage *three-way-merging*, which is a subject of Section 4.4.3.

4.4.1 Pessimistic Synchronization

Pessimistic synchronization is also referred to as the *lock/modify/unlock* approach, which already points out to the main principle: before being capable of modifying a specific software object, a developer must obtain a *lock* from the repository. As soon as an object is locked, no write access on it is granted to other developers — attempts to obtain the same lock will be denied, disallowing a modification. Locked resources may be *unlocked* either by aborting the current modification or regularly in the course of a commit [Pop09].

Figure 4.7(a) demonstrates the pessimistic approach using the example of two hypothetical users. Alice checks-out the software object she intends to modify, and then puts a lock on it. The lock is granted as it does not interfere with existing locks. Then, she starts modifying the object. Concurrently, Bob checks-out the same object and attempts to obtain a write lock. This is denied until Alice releases the lock after committing a new version of the object.

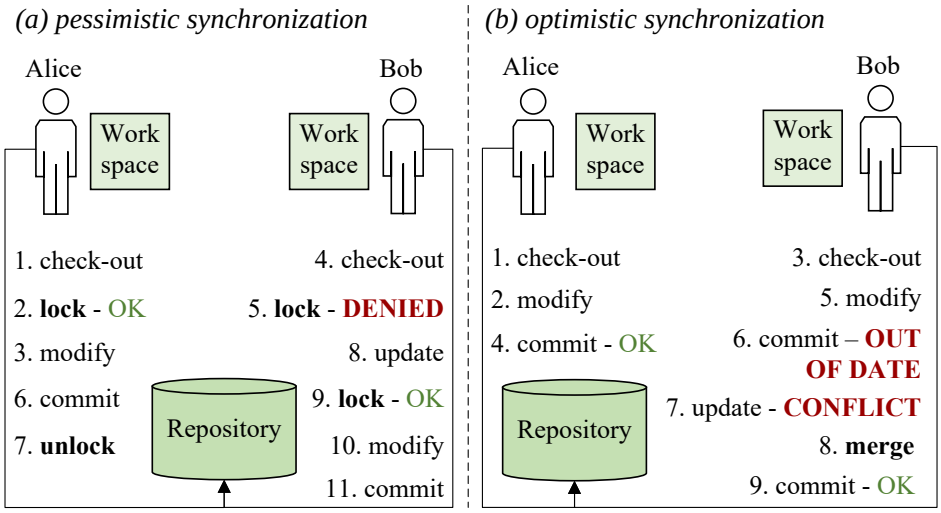


Figure 4.7: Pessimistic vs. optimistic synchronization.

Bob must now update (i.e., check-out the latest version of) the software object. Since Alice has unlocked in the meantime, Bob's lock attempt is now granted and he may carry out his changes before committing.

As with locks in general, the *locking granularity* implied by a pessimistic synchronization strategy is crucial [RS77]. The most coarse-grained lock possible would be the entire repository, which generally disallows concurrent modifications and enforces a linear version history. This is obviously too restrictive. Fine-grained locking would refer to a subset of units of a software object, e.g., a specific range of text lines. In spite of being least restrictive, this would result in a considerable synchronization overhead. As a compromise, locks might be put on specific software objects, or hierarchies thereof (following their composition relationships transitively), for instance, specific text files or folders containing those.

Pessimistic synchronization theoretically avoids conflicts before they can occur. When considering dependency relationships between software objects, though, isolated changes may still destroy the structural consistency of the versioned program. Furthermore, the lock/modify/unlock approach demands developers for maintaining the discipline of locking no more objects than necessary and of unlocking modified objects as soon as possible.

A representative VCS relying on pessimistic synchronization is RCS [Tic85].

4.4.2 Optimistic Synchronization

The *copy/modify/merge* approach realized by VCS relying on *optimistic synchronization* relinquishes locks in favor of *merging* conflicts, which can occur provided that concurrent modifications of the same software object are allowed. This paradigm is typically realized in two steps [CFP04]. First, during commit, it is checked whether another revision has been added to the repository by a different developer in the meantime. If this is the case, an *update* is enforced as a second step. The update may be either *straightforward* – in case the set of locally and remotely modified software objects is disjoint – or *conflicting*. Conflicts may in turn be resolved automatically (e.g., if the modifications to a source code file happened in different isolated regions), semi-automatically (e.g., by three-way merging; see below), or manually (in case the VCS does not support merging explicitly).

An example is provided in Figure 4.7(b). Alice checks-out a software object and immediately starts modifying it. Before she commits, Bob concurrently checks-out the same version, also modifies it, and attempts to commit. This fails due to Alice's write lock established on commit, so an *out of date* situation is signaled to Bob, who enforces an update. Since the concurrent modifications affected the same software object, a *conflict* is detected. Bob now *merges* the local and remote changes using a semi-automatic three-way merge tool, before he commits the combined changes back to the repository.

Obviously, *copy/modify/merge* is more liberal than *lock/modify/unlock*. Furthermore, synchronization overhead is caused only in the case of conflicting modifications. As shown below, however, three-way merging constitutes a complicated task.

Optimistic synchronization is realized by both Subversion and Git. In the case of three-way merge conflicts, their resolution is ceded to third-party tools. Like many VCS, both systems allow to put locks on specific objects, offering pessimistic versioning as an option.

4.4.3 Three-Way Merging

The goal of *three-way merging* is to reconcile changes persisted in two versions, which have a common origin version, in a single merged version, in a preferably consistency-preserving way. Many decisions can be automated by taking into account the origin version.

As contrasted by Figure 4.8, there exist two distinct approaches to three-way merging in general [Wes14]. The categories are orthogonal to log-based versus comparison-based versioning introduced in Section 4.3.2. For instance, log-based versioning might be combined with a state-based merging algorithm.

State-Based Merging. Provided that the elements of the three input versions – the common ancestor b and the alternative versions v_1 and v_2 – are known, the set of elements to be contained by the merged version m can be directly calculated by a function *merge*. This approach suits well with set-theoretic product space models, e.g., the three-way model merging algorithm presented in [Wes14] (cf. Section 6.2.2). When applied to sequential data structures such as text files, the merge function must be *order-preserving* and detect *conflicting insertions* at the same location.

The command line tool *diff3* is popular for three-way merging text files; deep integration with Subversion and Git is provided. Diff3 is classified as a state-based merging tool although its operations are derived from change-based intuitions. A formal investigation of the tool has been provided in [KKP07]. Ensuing from two pair-wise matchings of b and v_1 as well as b and v_2, resulting in two LCS, a sequence of *stable* and *variable chunks*, i.e., subsequences of text lines, is computed. A stable chunk is contained in all three versions, whereas a variable chunk exists in at least two different versions. An *insertion conflict* is present whenever a chunk has different content in the alternative versions v_1 and v_2. In this case, both conflicting versions of the variable chunk are copied into the merged result, and the user has to delete one of the alternatives in order to resolve the conflict.

Change-Based Merging. Approaches belonging to this category do not calculate the merge result m directly, but use *directed deltas* as indirection. First, the differences between the base version and the respective alternative versions are computed, resulting in the operation sequences $\Delta(b, v_1)$ and $\Delta(b, v_2)$. The change-based *merge* function then calculates the *merged delta* $\Delta(b, m)$ as the combination of the two directed deltas. Change-based merging

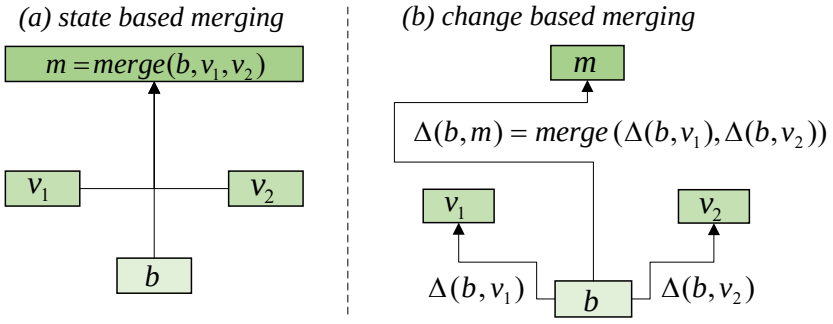


Figure 4.8: State-based vs. change-based three-way merging.

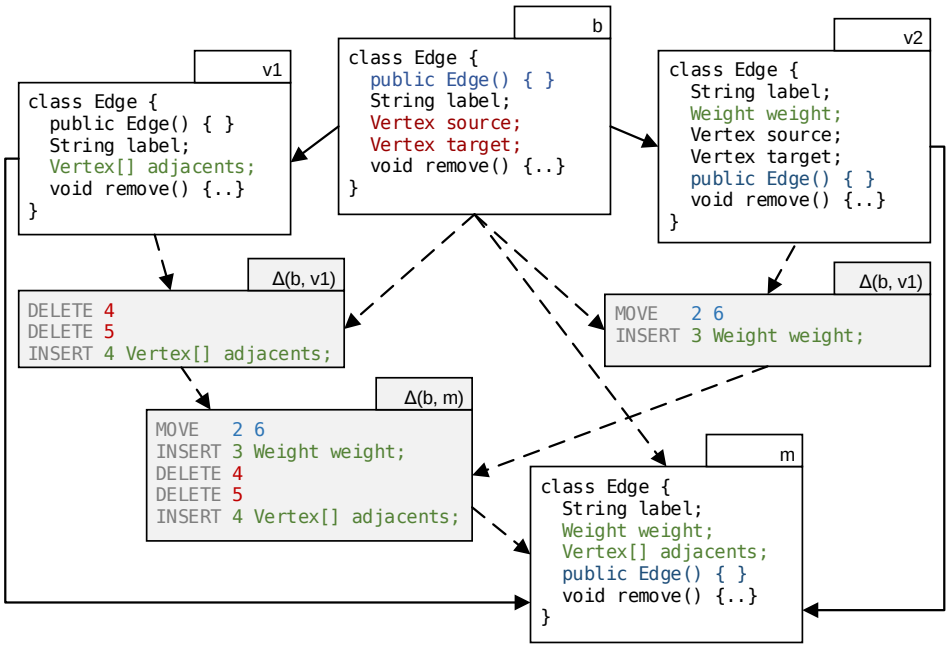


Figure 4.9: Example: change-based three-way merging.

is typically applied to sequential data such as text files. The deltas to be merged contain operations such as *insert* and *delete* as deduced by sequence differentiation algorithms.

Change-based merging requires the combined delta to be *order-preserving* with respect to the operations defined in the changes $c_1 := \Delta(b, v_1)$ and $c_2 := \Delta(b, v_2)$. Furthermore, the function *merge* "has to eliminate *duplicate operations* and has to detect conflicts. Here, a *conflict* between two operations $op_1 \in c_1$ and $op_2 \in c_2$ occurs [...] if either one of the operations *invalidates* the other one [...] or both operations are applicable in sequence, but they *do not commute*" [Wes14, p. 759]. Subversion's predecessor *CVS* [Ves06] applies a change-based three-way merging strategy to realize optimistic synchronization. The way conflicts are reported to the user resembles the strategy employed by diff3 (see above).

An example of change-based merging of text files is depicted in Figure 4.9. Ensuing from a common base version b of the example source file `Edge.java`, two alternative versions v_1 and v_2 have been concurrently developed. The differences between the base version and the alternatives are deduced from matchings obtained from Heckel's Algorithm. Therefore, the deduced deltas contain *move* operations. In $\Delta(b, m)$, the operation sequences are then combined in an order-preserving way without conflicts. The merged version m is created by applying the merged delta to the base version.

4.5 Distributed Version Control

The explanations given above have indicated that version control systems contribute to the *team* functionality area of SCM by providing synchronization operations that allow for concurrent collaborative development. Yet, the *centralized* VCS architecture considered so

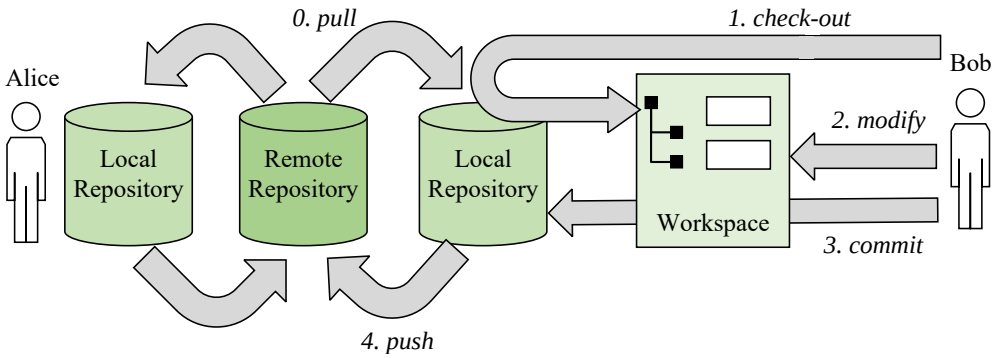


Figure 4.10: Possible usage of distributed version control relying on Git.

far implies the bottleneck of a single repository having to synchronize immediately upon every commit. The synchronization overhead proportionally increases with the number of developers involved. A central VCS is also a bottleneck when seen from the perspective of system reliability, since it provides the sole copy of the entire version history, whereas developers maintain in their workspace only the selected revision of the project.

These shortcomings of centralized VCS have given raise to a new category of systems supporting *distributed version control* [Cha09], which have become especially popular in open source development. These systems rely on an intentional replication of the repository, such that the aforementioned bottleneck is removed. Furthermore, the physical separation of repository and workspace is suspended as "every checkout is really a full backup of the data" [Cha09, p. 3]. As a consequence, multiple *clones* of the repository need to be maintained. This is realized by peer-to-peer synchronization operations, e.g., *pull* and *push* in Git.

Figure 4.10 illustrates the main difference between centralized and distributed VCS (DVCS). The operations CHECKOUT and COMMIT are now used to communicate with a *local copy* of the repository. At dedicated synchronization points, the operations PULL (for fetching incoming changes) and PUSH (for sharing outgoing changes) can be invoked. The pull/push loop added around the check-out/commit loop allows a distinction between locally scoped *short transactions* and globally scoped *long transactions* [Fei91], which have been identified above as a desirable SCM feature (cf. Section 4.2.1).

By repeated *cloning*, a *hierarchy* of repositories may be created, where functionality is implemented in leaf repositories, and inner nodes are responsible for merging changes and propagating them up stepwisely to the root. Several ways of hierarchical organization, such as the "integration-manager" or "dictator and lieutenants" workflows, have been established [Cha09]. The concept of *pull requests* plays an important role especially in open source development, where everybody may clone the repository and contribute to the software project. Yet, the problem of diverging variants of a repository still calls for some superordinate centralized repository management, which is realized, for instance, in hosting platforms such as *GitHub*[9].

9 https://github.com/

4.6 Intensional Versioning and Variation Control

To conclude this section, let us return to the definition of *version* as a generalization of *revisions* – historically ordered versions that supersede each other – and *variants* – co-existing instances of the versioned product differing in logical aspects such as functionality. Correspondingly, we have to distinguish between different *intents of evolution* [CW98].

Most version control systems support variant management by *branches* in the revision graph. Each branch is considered to differ from the *trunk* with respect to a fixed variation property, e.g., an optional customer-visible feature, a performance improvement that is still considered as experimental, or a feature that is only applicable to a certain operating system. The obvious limitation of branches is that each of them may reflect only one variable. Due to combinatorial complexity, in case n independent variables are desired, 2^n branches would be necessary to maintain the entirety of available product variants.

This shortcoming is due to the fact that all VCS approaches discussed so far in this chapter rely on a specific form of version space organization which is called *extensional versioning* [CW98][10]. The set of available versions VER is defined by enumerating its members (regardless of whether they correspond to versions lying on the trunk or on branches):

$$VER_{ext} = \{ver_1, \ldots, ver_n\} \tag{4.1}$$

Contrastingly, *intensional versioning* assumes that versions are constructed by resolving the available configuration decisions. "Instead of enumerating its members, the version set is defined by a predicate" [CW98, p. 239]:

$$VER_{int} = \{ver | cons(ver)\} \tag{4.2}$$

where ver may be any configuration defined in terms of the available properties of the system, and $cons$ is a boolean function defining whether the configuration is *consistent*.

Intensional versioning has not (yet) gained widespread acceptance in industry although there exist several academic prototypes and experiments of *variation control systems* that demonstrate that the approach is feasible [WO14; Stă+16; LBG17].

4.7 Bottom Line

Software configuration management is indispensable; its engineering facet is mostly covered by VCS, which orchestrate identification of components, change management, and team functionalities. Concrete VCS largely differ with respect to their internals. Different approaches exist, e.g., for delta storage, change detection, and synchronization. When allowing for concurrent modifications, three-way merging comes into play.

Intensional versioning has not gained widespread acceptance. As a replacement, developers tend to rely on external tools for the management of configuration options. Several approaches are described in the next chapter in the context of *software product line engineering*. Throughout this thesis, intensional versioning is revisited in several contexts. *SuperMod* supports a combination of extensional and intensional versioning.

10 For consistency with other definitions provided here, some symbols and identifiers were renamed.

Any customer can have a car painted in any color that he wants as long as it is black.

HENRY FORD (1922)

Chapter 5

Software Product Line Engineering

Abstract

Software product line engineering is motivated by the economical advantage of being able to quickly and responsively develop customer-specific applications through organized reuse. Different formalisms exist to document the variability managed by a product line. Here, the emphasis is put on feature models. In the literature, several development processes have been established that intend to match the specific requirements of product line engineering. A multitude of implementation approaches exist, each implying its individual requirements and properties. Furthermore, current challenges in research on well-formedness analysis of software product lines are explained in this chapter.

Contents

5.1 Motivation and General Definitions

Cornerstones of industrial revolution are often described using the example of the automotive domain. Until 1913, cars were manufactured in a batch production mode. Beginning with Henry Ford's T-Model, assembly-line production enabled *mass production*. Nowadays, we live in the era of *mass customization*: customers may freely configure the *features* of their cars, such as color, car body, and motorization, while the assembly of the car still relies on mass production techniques [CE00].

The economical benefits of mass customization are obvious: *time to market* is reduced while the customer receives an *individual* product. When compared to single-product manufacturing, mass customization considerably reduces production costs. This is due to the fact that individual components can be *reused* in an *organized* way. The initial effort for installing a *product line*, however, is considerable, such that a certain number of products, reflected by the *break-even point*, must be in scope in order to be profitable [CN01].

Software product line engineering (SPLE) [PBL05] denotes the adaptation of product line principles to software development. Thus, the product to be individualized is an executable program. When compared to industrial manufacturing, *mass production* is not a real concern in software engineering, since creating exact copies of programs requires almost no expense. In contrast, the principle of *mass customization* applies well, since the exact compilation of desired *features* of the respective software program varies depending on the specific scenario where the program is used. For instance, a text processing program may support multiple input methods or languages, different export formats, and vary with respect to the interfaces offered to external programs.

Time to market is reduced by *organized reuse* of (software) components, whereas customer satisfaction is increased through individualization. Furthermore, *software quality* is intended to grow, since repeated product tests can be reused in a the same organized way as the tested components [CN01]. The creation and maintenance of SPL artifacts, however, is complicated.

SPLE relies on two important concepts, which are defined in an abstract way here and explained in the context of specific SPLE approaches below.

Platform. According to [PBL05, p. 6], "a platform is any base of technologies on which other technologies [...] are built". In SPLE, the platform does neither correspond to the tool that is used to produce reusable components, nor to the technology that is used to assemble customized products. In contrast, the platform *is* the set of reusable components, which is *configured* individually for each product.

Variability. In [Ap+13b, p. 48], variability is characterized as "the ability to derive different products from a common set of artifacts", i.e., from a platform. Conversely speaking, without variability, there is neither a need nor a possibility to configure the platform. Variability is enabled by the presence of different *features* [Bat05]. Accordingly, individual members of a product line differ with respect to their individual selection of features they realize.

When referring to the definition given in [CE00], all decisions made concerning variability form the *problem space*, whereas the platform provides the *solution space* of an SPL. "The problem space comprises concepts that describe the requirements on a software system

and its intended behavior. The solution space comprises concepts that define how the requirements are satisfied and how the intended behavior is implemented" [AK09, p. 51].

Variability can be found anywhere in everyday life. The color of a car is a vivid example. [PBL05, p. 60] introduce two general concepts allowing to abstract from this:

Variability Subject "A *variability subject* is a variable item of the real world or a variable property of such an item", answering the question "what varies?".

Variability Object "A *variability object* is a particular instance of a variability subject", answering "how does it vary?".

For instance, in a product line of cars offering different colors, the color of a car would incorporate a variability subject, for which the concrete colors (red, blue, white) would correspond to mutually exclusive variability objects.

Variation Points and Variants. When referring to how variability is implemented in the product line, the technical terms *variation point* (for the implementation of a variability subject) and *variant* (for the implementation of a variability object) are commonly used. The concrete realization depends on the programming language used. For example, in Java, variation points may be realized by providing an interface that is implemented by different classes representing variants, one of which can be selected as the preferred implementation for specific product line members.

Variability in Time vs. in Space. We may also ask a third question in this context: "why does it vary?". Actually, there exist different kinds of variability, e.g., "different stakeholder needs, different country laws, [or] technical reasons" [PBL05, p. 60], which need to be further distinguished. *Variability in time* is introduced as "the existence of different versions of an [artifact] that are valid at different times". Accordingly, the evolution of the variability subject is described by a sequence of superseding variability objects. In contrast, *variability in space* is defined as "the existence of an [artifact] in different shapes at the same time" [PBL05, p. 65f.]. The distinction is similar to revisions versus variants as introduced in Section 4.2.2 in the context of version control.

External vs. Internal Variability. According to [PBL05, p. 69], *external variability* is "visible to customers", e.g., in the form of available program features, whereas *internal variability* is "hidden from customers", but affects architectural or implementation decisions, e.g., in the form of algorithmic optimization. Similar to this distinction is *requirements variability* versus *design variability* [Gom05]. Typically, external variability is explored and captured during requirements engineering and analysis. The variability model is further extended by internal variability during system design.

Options vs. Alternatives. The relationship between the subject and the object of variability may be established in two different ways [PBL05; Ap+13b]. On the one hand, *optional variation* denotes that there exists exactly one variability object, which may or may not be assigned to the variability subject. Therefore, resolving variability implies a boolean decision whether the corresponding feature shall or shall not be included in a specific product

line member. On the other hand, *alternative variation* is resolved by selecting exactly one element in a set of mutually exclusive variability objects. Generalized forms of alternative variation exist, e.g., allowing the selection of a number of variability objects different from one [CHE05].

Section 5.2 of this chapter explores *feature models* as a formalism for capturing variability. After an overview of SPL development processes in Section 5.3, solution space techniques will be considered in Section 5.4. Consistency-related challenges, referring to both the problem and the solution space, are listed in Section 5.5, before the chapter is concluded.

SPL Adoption Paths. Every SPLE project has its individual conditions, which are due to factors such as the number of product variants to maintain, the complexity and the criticality of the software, the targeted time to market, et cetera. In addition, the question how an SPL is *adopted* may also influence the choice of the process and implementation techniques to employ. According to [Ap+13b], there exist three typical adoption paths.

– In a *proactive* approach, the SPL is initiated from scratch based on a thorough economical and technical analysis, after which the requirements, the discriminating features, and the target customers of the products of the SPL are fixed.
– The *extractive* SPL adoption path, in contrast, assumes a transition from single-product development to organized SPLE. The product line may be initiated based on a set of existing products whose individual properties are known.
– Last, *reactive* SPLE begins with a small (proactively or reactively obtained) product line, which is extended incrementally feature by feature based on new requirements or economic reconsiderations. We can observe several similarities to *agile* software development (cf. Section 1.1.2) in general: early customer feedback, short development cycles, as well as minimal and incremental planning. See also [Ap+13b, p. 41]

The connection between different adoption paths and *SPLE processes* is explained in Section 5.3.

5.2 Feature Models

Several formalisms for the definition of variability have been established, for example the *orthogonal variability model* (OVM) [PBL05]. In the context of this thesis, *feature models* [Kan+90] have been selected as default formalism for documenting variability.

Feature models are hierarchical decompositions of mandatory or optional *features*. Furthermore, cross-tree dependencies between features are allowed. In contrast to other formalisms including OVM, feature models are not only used to model differences, but also *commonalities* [Cza+12]. Due to the hierarchical structure, the role of non-leaf features may become dual, serving both as a *variation point* and as a *variant*.

Until now, there exists no OMG standard for feature models[1] such that the elements introduced in [Kan+90] still serve as the “common sense”. The external representation of a

1 In a request for proposal for the *Common Variability Language* (CVL), the syntactically and semantically similar concept of *variability specifications* (VSpecs) is being introduced [OMG12].

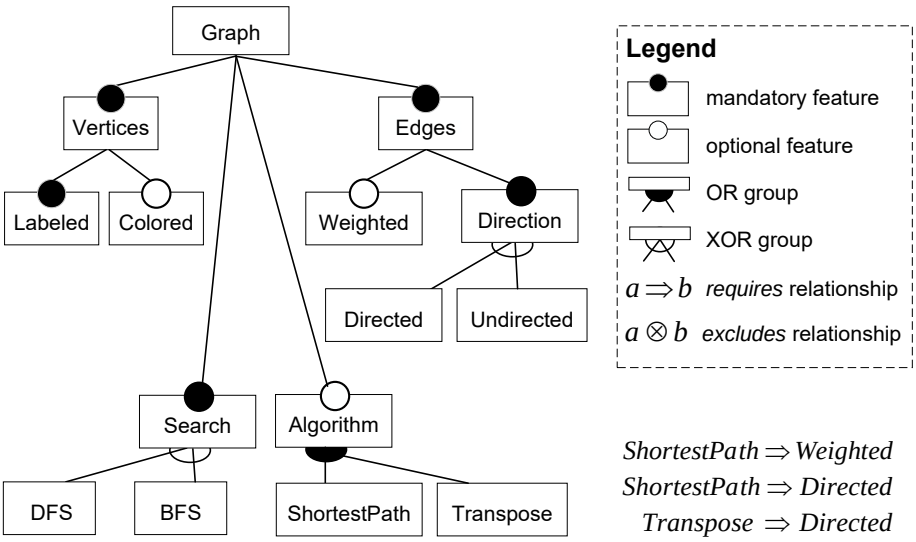


Figure 5.1: Diagram representation of a feature model for the *Graph* example.

feature model is called a *feature diagram* [Ap+13b]. Figure 5.1 depicts a feature diagram for the running *Graph* product line.

The variability defined in a feature model needs to be resolved in order to describe the characteristics of specific products of the product line. A *feature configuration* assigns a boolean value (*true* for selection, *false* for deselection) to each individual feature. Relationships defined in feature models further constrain the validity of feature configurations.

The example of Figure 5.1 is used to explain the *semantics* of feature models with respect to their possible feature configurations:

Root Feature. The root feature of a product line – in this example Graph – must always be selected in every configuration.

Parent-Child Relationship. This may be used in order to define variants (child features) for a variation point (parent feature). The selection of a *child* feature is only allowed in case its parent feature is selected. On feature model level, the parent of a feature must be *unique*.

Mandatory Features. Features may be defined as *mandatory* with respect to their parent feature. Then, the child feature's selection must correspond to the parent feature's selection.

OR Groups. A set of features belonging to the same parent may be arranged in an *OR group*. In a corresponding feature configuration, at least one of these features must be selected in case the parent is selected. E.g., if Algorithm is selected, either ShortestPath or Transpose or both can be selected. OR-grouped features cannot be mandatory.

XOR Groups. Out of a set of features arranged in an *XOR group*, exactly one must be selected in a valid configuration in case the parent is selected. In the example, DFS and BFS are mutually exclusive. XOR-grouped features cannot be mandatory either.

Requires Relationships. This kind of relationship is not explicitly shown in the diagram, but is listed as an external textual constraint. $ShortestPath \Rightarrow Weighted$ states that ShortestPath may only be selected in case Weighted is selected.

Excludes Relationships. In analogy, an *excludes* relationship of the form $a \otimes b$ denotes that a feature a must not be selected if b is selected, and vice versa.

An example of a valid feature configuration is given in Figure 5.2. More specifically, a product variant that corresponds to a graph with labeled, but not colored vertices, as well as weighted and directed edges is described. From the mutually exclusive search algorithms, depth-first search is selected. In addition, the Transpose algorithm is supported.

Not at least due to its unstandardized state, many extensions and generalizations to feature models exist, e.g., *cardinality-based feature modeling* [CA05], which generalizes from OR and XOR groups by introducing group selection ranges. Features may also carry attributes with a defined value range (e.g., string or integer). Furthermore, features can have multiple instances which differ, e.g., in their attribute values or in selection states of its child features. For the remainder of this thesis, unless stated otherwise, we assume that the modeling constructs explained above are sufficient.

For the same reason, a variety of *feature modeling tools* exists, supporting different descendants of the modeling language, e.g., *pure::variants*[2] or *FeatureIDE*[3]. *SuperMod* includes its own tree-based feature model editor for the creation and configuration of feature models (cf. Section 14.2.2).

For analyses such as satisfiability checks, the set of constraints imposed by (non cardinality-based) feature models may be internally expressed by (and mapped to) *propositional logic* [Bat05]; this is further elaborated in Section 9.4.

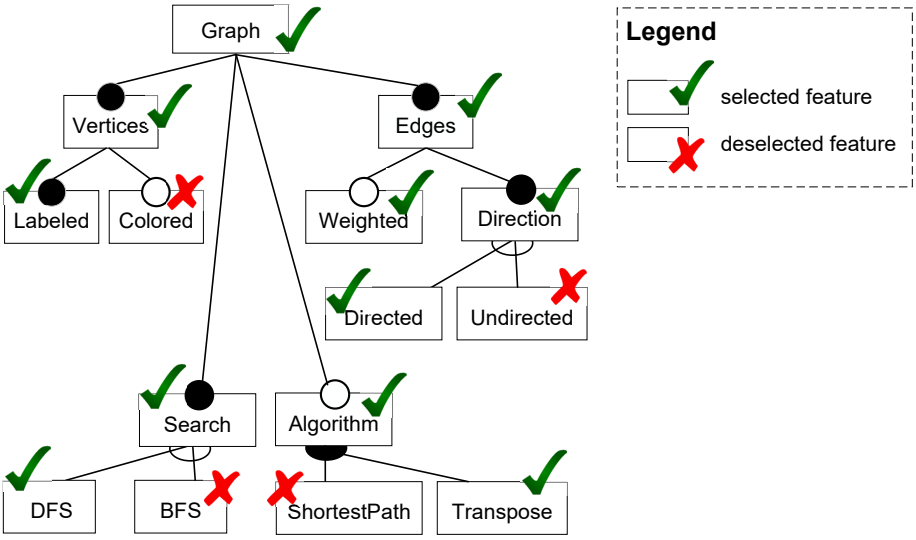


Figure 5.2: An example feature configuration of the *Graph* feature model.

2 http://www.pure-systems.com/products/pure-variants-9.html
3 http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

5.3 The Process Perspective

Before considering tools for the implementation of software product lines that systematically exploit the captured variability, this section gives an overview of *development processes* specific to SPLE. [4]

Most SPLE processes explicitly distinguish between two phases of development activities, *domain engineering* and *application engineering* [PBL05]; see Figure 5.3. Furthermore, the processes found in the literature often assume a *proactive* adoption path, where the SPL is planned in advance and the majority of features is known before they are analyzed, designed, and implemented. Processes explicitly targeting *reactive* or *extractive* approaches are underrepresented. In Section 16.3.2, we retrospectively discuss how the contributed framework may be backed by a reactive SPLE process that is compliant to agile principles.

Domain Engineering. This includes the analysis of the *domain*, i.e., "an area of knowledge that [...] includes a set of concepts and terminology understood by practitioners in that area" [Ap+13b, p. 19]. Domain engineering comprises the analysis of the variability present in the domain, resulting in a *variability model*, e.g., feature model, and the design as well as the implementation of the common software components to solve the domain problem, forming the *platform*.

Application Engineering. Here, artifacts created during domain engineering are reused and configured in order to create custom *applications*. To this end, the captured variability is resolved, e.g., by a feature configuration, and the platform is customized in order to match the application requirements. This should happen in a preferably automated way, but manual product adaptations are necessary in general.

The SPLE Process by Pohl et al. The SPLE process described in [PBL05] provides for a sequential order of DE and AE as well as their activities. Prior to DE stands an additional

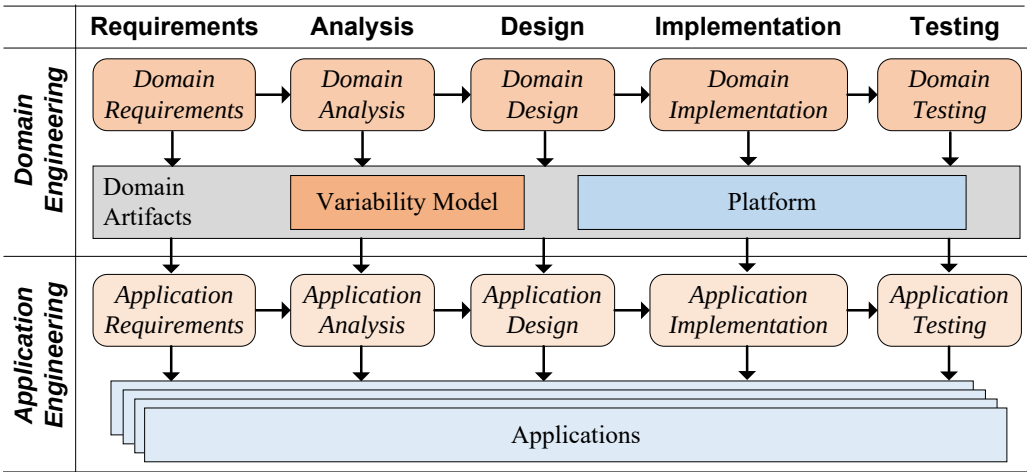


Figure 5.3: General distinction between domain and application engineering.

4 Excerpts of this section have been pre-published in [SBW16a].

activity, *product management*, where the scope of the product line is planned, including economical considerations. AE is applied repeatedly for each product; it strictly recapitulates DE and re-uses artifacts developed there, i.e., the outcomes of domain analysis, design, implementation, and testing. Just like the waterfall model was soon extended by feedback loops, Pohl's process allows for overall *iterations* of DE and AE [PBL05].

Three-Fold Process by Clements and Northrop. Clements and Northrop [CN01] define an iterative SPLE process consisting of three main activities, namely *core asset development*, *product development*, and *management*, which coarsely correspond to DE, AE, and product management, respectively. It is assumed that all three activities are performed in parallel, evolving both the platform and individual products continuously.

Gomaa's Double Spiral Model. Gomaa's *double spiral* development model [Gom05] is a *risk-driven* process that allows for alternations between the activities of DE and AE, which are executed in intertwined spirals, each structured along four management/development activities: (1) "define objectives, alternatives, and constraints", (2) "analyze risks", (3) "develop", (4) "plan next cycle". As a consequence, *product management* happens in each iteration rather than being a prefix activity as suggested in the process by Pohl et al.

Two-Dimensional Process by Apel et al. In [Ap+13b], Apel et al. define a development process for a plan-driven variant of feature-oriented product line development. The process activities are organized along two dimensions. The first dimension corresponds to the classical DE/AE partition, whereas the second dimension makes a distinction between the *problem space* and the *solution space*. The resulting quadrants contain the activities *domain analysis* (for DE in the problem space), *domain implementation*[5] (DE in the solution space), *requirements analysis* (AE in the problem space), and *product derivation* (AE in the solution space). To anticipate new or changing requirements, the two problem space activities are provided with a feedback loop. This allows for both proactive and reactive adoption paths.

5.4 Classification of SPL Implementation Approaches

After having seen the "big picture" of SPL concepts and development processes, let us now have a closer look at implementation techniques used in the *solution space*, where the captured variability is exploited in order to create a variability-aware *platform* from which specific products may be constructed. Section 5.4.1 introduces means to classify different approaches, which are then explained in Sections 5.4.2 until 5.4.5.

In this section, we assume *source code* as the type of artifact to provide the platform. Solution spaces that combine *models* and SPLE are a subject of Section 6.1.

5.4.1 Classification Dimensions

Five dimensions along which SPL implementation approaches may be classified are summarized in Figure 5.4 and further explained below:

5 Here, implementation denotes all activities subsequent to analysis.

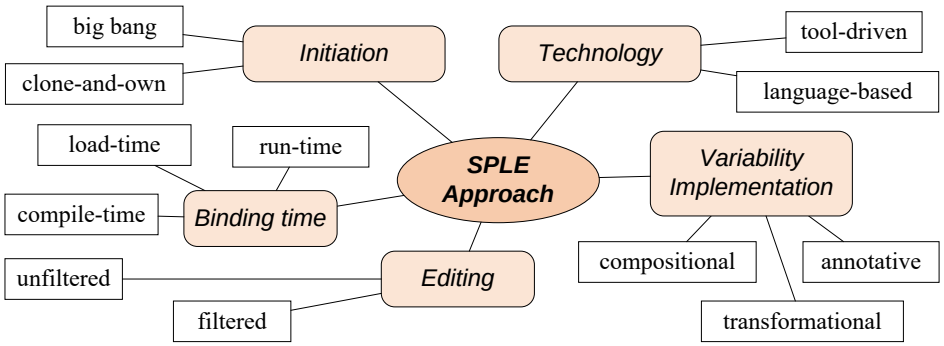


Figure 5.4: Classification dimensions for SPL implementation approaches.

Initiation. A first distinction is made with respect to the point in time when, e.g., economical considerations decide about the introduction of a product line. In *big bang* scenarios [KC14], domain engineering artifacts are all created from scratch, before combining them to individual products as sketched in the idealized process in Figure 5.3. Planning product line support from the beginning implies a significant initial set-up effort, which is reflected in the break-even point. Big bang initiation mostly correlates with a *proactive* SPL adoption path.

In contrast, many SPLs, especially in the open-source world, have been retrospectively defined as such. In this case, domain engineering artifacts are derived in an *extractive* way from a set of cloned variants, which have been developed independently, e.g., by repeatedly forking a publicly available software repository [Fis+15]. This *clone-and-own* approach implies many new challenges such as identifying the differences at product level that correspond to specific features. An exploratory study can be found in [Dub+13].

SPLs following the *reactive* adoption path may be initiated in either of both ways.

Binding Time. In order to resolve the variability defined in the feature model, a set of decisions must be made, each resolving a variation point by selecting an appropriate variant for it. Assuming that the platform has been realized in a general purpose programming language, there are at least three points in time when these decisions can be made effective: (a) at *compile-time*, i.e., before or while the compiler transforms source code into the runnable program, (b), at *load-time*, i.e., when the program is started, or (c) at *run-time*, i.e., during program execution [Ap+13b].

While compile-time variability can ensure that only the exact functionality described by the respective product variant is deployed to the customer(s), run-time variability is more flexible and allows for dynamically reconfiguring product lines. Load-time variability incorporates neither of both advantages but is usually easier to implement and to debug [VG07].

Technology. Different SPLE approaches are furthermore distinguished along the categories *language-based* and *tool-driven* [Ap+13b]. In the case of language-based variability, variation points and variants are defined and resolved by means of mechanisms borrowed from the language the platform is realized with. In contrast, tool-driven

technology assumes that one or several external tools are used to manage variability, while the implementation language remains variability-unaware.

Language-based mechanisms require minimal set-up effort, since no additional tooling is required. Nevertheless, the danger of *scattering* source code with variability management code rises. In contrast, tool-driven variability separates these concerns and is therefore more generic, but requires a more heavyweight set-up.

Variability Implementation. A fourth distinction is made between *compositional*, *transformational*, and *annotative variability* [Ap+13b]. In the case of compositional variability, the platform consists of a minimal common core of source code, to which variable components are added upon product configuration in order to realize specific features. *Transformational* variability generalizes the compositional approach by allowing arbitrary operations, such as modifications and deletions of source code fragments, in addition to the monotonic composition function. Annotative variability, in contrast, ensues from a *multi-variant platform* that initially realizes all possible features at the same time. From this platform, specific products are derived by removing those pieces of code that belong to features not selected for the product. [6]

All three classes of approaches have their individual pros and cons when considering the problem of *product well-formedness*. Both composition and transformation may involve conflicting decisions for the resolution of variation points. Annotative variability may lead to problems such as uses of types without declaration [Thü+14a] or unintended feature interaction [Ap+13a]. Section 5.5 further discusses these issues.

Editing. A last, orthogonal distinction is made based on how the product line is actually edited; typically, SPL editing requires a combination of several tools, covering the problem space, the solution space, and the mapping in between, respectively. *Unfiltered* as well as different forms of *filtered* editing come into question.

The majority of SPL editing tools that have been described in the literature follow an *unfiltered* approach. Here, the developer is faced with multi-variant artifacts and is able to make architectural decisions with respect to the implementation of variability. Typically, the connection between problem space and solution space is managed manually, which gives responsibility to the SPL engineer(s).

The goal of *filtered* (also: *projectional*) editing is to relieve the developer of this responsibility by reducing the complexity of multi-version editing. Here, the developer may edit the product line in a view that filters outs parts of the product not relevant for the intended change. There are different levels of filtered SPL editing such as *partial projections* [Stă+16], *temporary views* [Käs10], or *fully filtered editing*, where a representative product variant is edited. Typically, the mapping between problem space and solution space is managed in a (semi-)automatic way. In this thesis, an approach relying on fully filtered editing is presented.

Unless stated otherwise, the approaches discussed below follow the unfiltered approach.

6 In earlier publications, the terms "positive variability" as a generalization of compositional and transformational, as well as "negative variability" for annotative approaches, were used.

From the five classification dimensions, the distinction between different variability implementation paradigms organizes the representative approaches listed subsequently.

5.4.2 Annotative Variability

Annotative variability can be realized in an ad-hoc way (i.e., neither language-based nor tool-driven) by copying a multi-variant source code and by removing (or commenting out) those artifacts which are not needed for a specific product from the copy before it is compiled. Similarly, conditional statements evaluated at run-time can control the execution of the program in a way that specific parts are never executed. Obviously, the limits of these naive strategies are stretched soon, such that more organized and powerful techniques are employed for SPLE.

Preprocessor Languages. Many programming languages support a built-in *conditional compilation* mechanism based on *preprocessors*, which are therefore classified as a *compile-time* variability mechanism. Besides the well-known preprocessor for *C* [KR88], external approaches such as *Prebop*[7] offer cross-language support. Although preprocessors are embedded into languages, they are classified as *tool-driven* [Ap+13b].

Preprocessor languages extend their host languages by conditional instructions. In C, code wrapped by `#ifdef X` and `#endif` will only be passed to the compiler in case a *compilation option X* is defined by `#define X`. Similarly, `#ifndef X` marks a region of code that is only compiled in case *X* is *not* defined.

This mechanism can be exploited in order to define the platform of an SPL by means of a *multi-variant source code* that surrounds variable parts by corresponding conditional preprocessor directives, where configuration options, e.g., *X*, correspond to features. During conditional compilation, variability is resolved by a corresponding list of `#define` directives

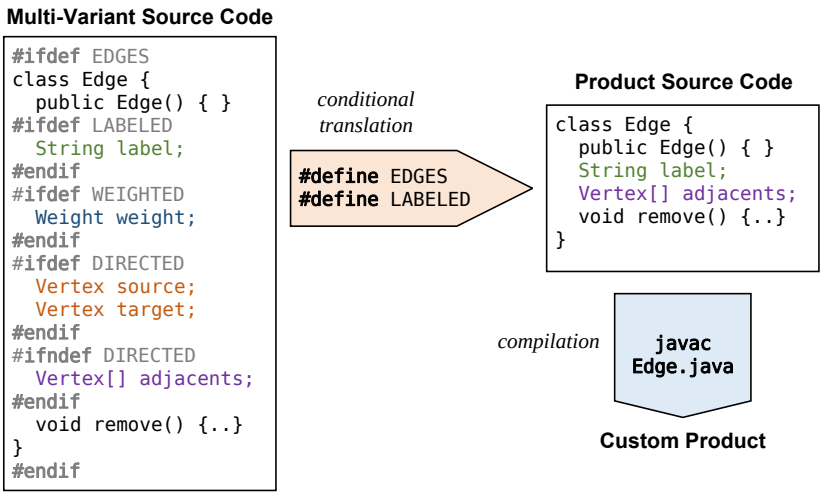


Figure 5.5: Conditional translation using a hypothetical Java preprocessor.

7 http://prebop.sourceforge.net/

for features to be included in the product. In the resulting *product machine code*, variable parts referring to features not realized are not included.

In Figure 5.5, conditional compilation is illustrated using a hypothetical Java preprocessor with a syntax resembling the C preprocessor. In the presented multi-variant source code, variable parts are embraced by corresponding preprocessor directives referring to equally labeled features. The specified configuration represents a labeled, unweighted, and undirected graph. For the reasons of clearness and comprehensibility, the representation distinguishes between conditional translation and compilation, which are technically combined in a single run by many language-internal preprocessors.

Using low-level preprocessor languages for implementing product line variability seems inadequate. This approach is used, however, in the *Linux* kernel[8], which can be considered as a large-scale SPL having more than 9000 features [Lot+10]. Variability is therefore managed by an external tool, the *kernel configurator* [SS08].

Furthermore, preprocessors do not typically take into account syntactical well-formedness problems. For instance, in the example presented in Figure 5.5, the declaration of class `Edge` is conditional (option `EDGES`). Therefore, all *applied occurrences* of this class must be marked with the same preprocessor directive in a disciplined way. Otherwise, compilation problems can occur for variants where `EDGES` is not defined.

Virtual Separation of Concerns. In [Käs10], the tool *Colored IDE* (CIDE) is introduced under the slogan "preprocessors 2.0". Belonging to the categories *tool-driven* and *compile-time*, CIDE extends the preprocessor concept by several properties that make the preprocessor approach more suitable for SPLE. On the one hand, a *colored* representation of source code belonging to different configuration options is used in favor of conditional preprocessor directives, which tend to bloat the source code. Furthermore, developers may temporarily restrict a file containing multi-variant code by views that virtually bind some configuration options and enable *temporarily filtered editing* in the background.

Explicitly Mapping-Based Approaches. When compared to the approaches and tools for annotative variability discussed above, *mapping-based* techniques [CA05] are considered as more general and as more SPLE specific. Approaches belonging to this category are *tool-driven* and are applied at *compile-time*.

A *mapping* is a set of tuples, each connecting an element of the platform to a *presence condition*[9] [CA05], a propositional logical expression on the variables defined in the feature model. In case the expression does not evaluate to *true* given the bindings specified in a configuration, the referenced platform element is removed from the corresponding product.

In Figure 5.6, an example relying on a hypothetical mapping tool for source code elements is provided. The platform is defined by means of *multi-variant source code* that is clear of variability information. The is added by the *mapping*, which is here represented externally as a table, where the source element is referenced by its qualified name. Presence conditions, which encode variability information, are represented in a hypothetical textual syntax. Application engineering is realized by *filtering* the elements with respect to the binding

8 https://kernel.org/

9 Also: *feature expression* or *feature annotation*.

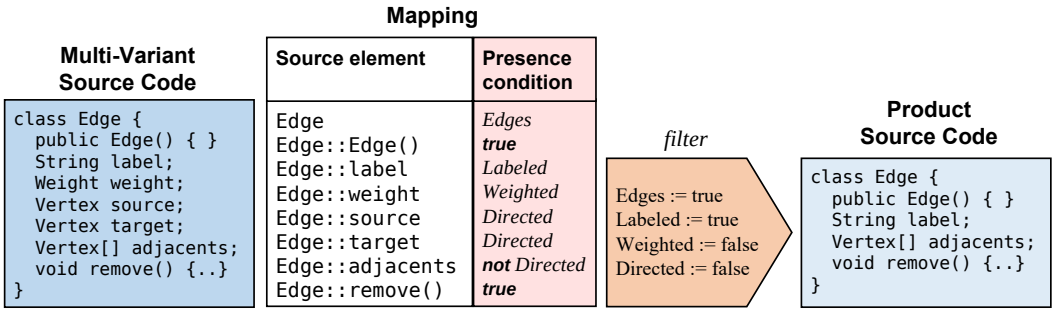


Figure 5.6: Principle of mapping-based SPLE by the *graph* example. Due to nesting, the presence condition of a child element implicitly includes (by conjunction) the parent's condition.

specified in the configuration. In this example, a graph with labeled, unweighted, undirected edges is configured.

In contrast to preprocessors, variability is made explicit by the mapping, such that no scattering of platform code and conditional statements occurs. On the downside, separation of platform and variability information also reduces the comprehensibility of the mapping when compared to preprocessors or CIDE. Furthermore, the question remains how to *identify* elements of the platform referenced in the mapping, and at which object granularity they can be identified. For this purpose, corresponding solutions must be well-integrated with the language in which the multi-variant source code is written. In addition, similar consistency problems as in preprocessor-based approaches may arise, e.g., when dealing with declarations and applied occurrences.

5.4.3 Compositional Variability

Being based on the idea of adding specific program parts (statically or dynamically) to an existing code base, compositional variability can be achieved with small-scale solutions such as *dynamic loading* of class libraries in connection with the use of *behavioral design patterns* [Gam+95]. Albeit, software product lines are not "just a reconfigurable architecture" and more than "just [based on] a component-based development" [CN01, p. 12]. Therefore, we here consider approaches that suit the SPLE principle of organized reuse.

Build Systems. Build systems such as *Ant* or *Maven* contribute to the *construction* functionality area (cf. Section 4.1) of software configuration management. A *build script* automatically manages the compilation of a runnable program based on a collection of source code and configuration files. This mechanism can be exploited in order to manage *tool-driven compile-time* variability. To this end, for each specific product variant, a custom build script, which compiles a product from a subset of the source code files available in the base collection, is defined.

With build systems, variability is managed in a rather coarse-grained way [Ap+13b]. Variation points may be realized by either including or not including a specific source file. The contents of the file are typically not variability-aware, although a combination of build systems and preprocessors is feasible and has been realized for the Linux kernel [Ada+07].

Furthermore, variants of source code files are managed in an *extensional* way (cf. Section 4.6), such that for each valid combination of feature selections, a specific variant of affected source code files must be maintained.

Component Frameworks. *Component frameworks* can be employed as a *language-driven* solution for SPL implementations that resolve variability either at *load-time* or at *run-time*. A commonality of the multitude of frameworks available is that they allow to dynamically load or re-load parts of programs modularized in the form of, e.g., libraries or *plug-ins*.

As a representative, the technology *Open Services Gateway initiative* (OSGi) [MVA10] has been selected for the subsequent explanations. OSGi, on top of which, among others, the *Eclipse* platform has been built [CR06], relies on the metaphors *extension point* and *extension*. A *core module*, which represents the executed program, may define a couple of *extension points* technically represented as a Java interface and an XML schema defining the required metadata. *Plug-ins*, being deployed separately, are containers for *extensions*, which fulfill the contract defined by a particular extension point. During load-time or run-time, the core module may query the available extensions for a particular extension point in order to dynamically load variable portions of source code.

Coming back to the concept of compositional variability, plug-ins offering extensions belonging to the same extension point can realize a variation point having different variants. This is illustrated in Figure 5.7, where extension points (i.e., variation points), are defined for three optional features of the *Graph* example. Different plug-ins define two extensions (i.e., variants) for each variation point, where one extension realizes the presence, the other one the absence of the respective feature. The run-time configuration is finally created by selecting one extension for each extension point (e.g., based on a configuration file).

A similar mechanism is provided by *dependency injection* frameworks, which realize the design pattern *inversion of control* [Gam+95]. For instance, *Google Guice*[10] allows the definition of a set of Java interfaces to abstractly define the contract for *services*. Within service implementations, other service interfaces can be *injected* without declaring their implementation class. In *modules*, concrete implementation classes are ultimately bound to their service interfaces. This way, load-time variability can be achieved [Van08]. SuperMod, which contains run-time variability to a limited extent, has been developed using dependency

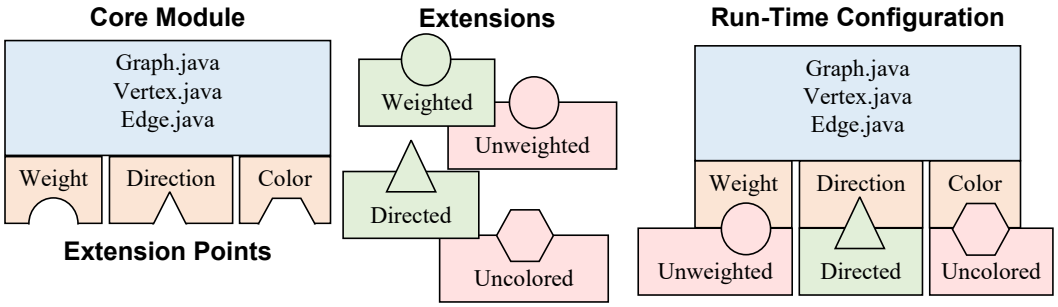


Figure 5.7: Compositional run-time variability using OSGi-like configuration.

10 https://github.com/google/guice

injection; see Section 14.4.2.

The component framework approach naturally enforces highly modular software architectures, which are considered as more reliable and stable [MVA10]. A downside of using frameworks for compositional variability is the lack of static analysis. For instance, binding errors caused by several implementations competing for one interface or by unimplemented interfaces cannot be detected at compile-time.

Aspect-Oriented Programming. The paradigm *aspect-oriented programming* (AOP) has been invented as an extension to procedural or object-oriented languages having the aim to reduce the undesirable phenomena of *code scattering* – code belonging to the same functionality being distributed over multiple fragments of the program – and *code tangling* – fragments of a program containing code belonging to different functionalities [Kic96]. The main principle of AOP is *separation of cross-cutting concerns*, which do not refer to the main functionality of the program, but to (optional and mostly orthogonal) aspects such as logging, tracing, or transaction management.

In the context of AOP, a couple of new language concepts have been introduced. Those are realized by several programming languages such as *AspectJ*[11] for Java. "Dynamic crosscutting in AspectJ is based on a small but powerful set of constructs. *Join points* are well-defined points in the execution of the program; *pointcuts* are a means of referring to collections of join points and certain values at those join points; *advice* are method-like constructs used to define additional behavior at join points; and *aspects* are units of modular crosscutting implementation, composed of pointcuts, advice, and ordinary Java member declarations" [Kic96]. From these specifications, the actual program is composed by a so called *aspect weaver*, whose output is passed to the compiler.

AOP can realize *compile-time*, *language driven* compositional variability. As [VG07, Section 1] explains, "aspect-oriented techniques enable the explicit expression and modularization of crosscutting variability [...]". Following this, the behavior belonging to a mandatory or optional *feature* can be encapsulated in an *aspect*. The decision which aspect to include is determined by a higher-level product configuration specification.

AOP methods allow for fine-grained extensions to an existing code base while reducing the problems of scattering and tangling of code, which are also relevant to SPLE. Though, AOP was not originally designed for SPLE, such that the solved problem is related but different. Not every feature can be considered as a cross-cutting concern—hence, AOP is not the adequate solution for every feature in an SPL [Ap07].

Feature-Oriented Programming. Like AOP, *feature-oriented programming* (FOP) is a compile-time and language-based solution for compositional variability [Ap+09b]. In contrast to AOP, FOP specifically targets software product line engineering. The paradigm is compositional inasmuch as a baseline of software, corresponding to the platform, is *refined*, i.e., extended by feature-specific parts. Two similar approaches to FOP are outlined below.

With *AHEAD*, [BSR04] have proposed a compositional approach internally representing programs as *algebraic hierarchical equations for application design*. Assuming that each program p is defined as a constant value, the program can be extended by mathematical

11 http://www.eclipse.org/aspectj/

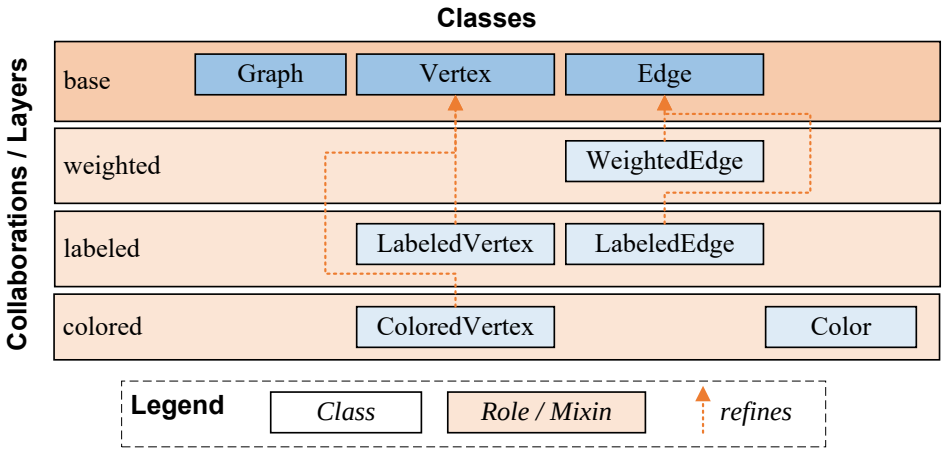


Figure 5.8: Composition of classes and refinements in feature-oriented programming.

function composition, e.g., $f_1 \bullet p$, $f_2 \bullet p$, or $f_2 \bullet f_1 \bullet p$ in order to realize features f_1, f_2, or both of them. Thus, composable features are defined as mathematical functions using the polymorphic operator $\bullet$. The approach has been implemented in a tool suite[12] supporting, among others, language extensions to Java. To this end, so called *refinement* classes or interfaces are made available to the developer. These are organized within different *layers*, each referring to a feature. A refinement may extend a base class using, for instance, a separate *mix-in* mechanism that is orthogonal to Java's built-in inheritance concept.

A similar approach is offered by *FeatureHouse* [Ap+09a]. Here, the terms *collaboration* and *role* are used for units of decomposition, and refinements each assign one collaboration to one source code artifact. There are also several syntactical differences to AHEAD which shall not be discussed here.

Figure 5.8 shows a conceptual generalization of FOP approaches. A product line is considered as a two-dimensional grid of source code artifacts and features. Based upon this orthogonal distinction, products can be derived by selecting those collaborations/layers which are to be included in the respective product configuration.

FOP offers several benefits when compared to other compositional approaches. On the one hand, *traceability* of features is easy since their source code artifacts are organized in layers, or feature modules, respectively. On the other hand, the composition mechanism is simple to understand and to invoke, provided that suitable extensions to the employed programming language exist. Nevertheless, *feature interaction*, which is considered in Section 5.5.1, is hard to resolve. This would require roles/mix-ins to spread over multiple collaborations/layers, which, however, is explicitly disallowed in strictly compositional approaches.

5.4.4 Transformational Variability

In contrast to approaches following the compositional paradigm, transformational variability assumes that a feature is generally defined by means of a *program transformation* that

12 https://www.cs.utexas.edu/users/schwartz/ATS.html

modifies a core program by inserting, altering, and removing fragments of code. These transformations are defined externally, such that approaches listed here belong to the categories *tool-driven* and *compile-time*.

Delta-Oriented Approaches. Several approaches aim at realizing transformational variability based on explicit and user-visible *directed deltas* [Ach+11; Schae+10]. *Deltas* as used in this context are more general, but also more fine-grained than aspects or refinements used in the compositional paradigms AOP or FOP, respectively. For instance, a delta may change the type of a variable used in a program or delete specific program statements.

The connection between deltas and features is summarized in *delta modules*. A delta module consists of a collection of deltas and a *presence condition* (see Section 5.4.2) that controls for which product variants the deltas are applied. Correspondingly, the product line consists of a baseline, a feature model, and several delta modules [Pie+15]. Automatic product derivation is then realized as follows: First, the presence condition for each delta is evaluated in order to detect whether it is relevant for the specified feature configuration. Then, an *execution order* is determined. Last, the deltas are applied to the baseline.

Determining the correct execution order is a key challenge in delta-oriented approaches, since conflicts may involve user interaction. This is in contrast to compositional or annotative approaches, in which product derivation is monotonic – fragments are only added or only removed, respectively –, and therefore not sensitive to the execution order. Taken together, the additional flexibility and power is paid by potential consistency problems and by nondeterminism.

Version Control Systems. As mentioned before, *variants* are a special case of *versions*, such that version control systems can be used for variability management. When speaking in SPLE terms, VCS are primarily designed for managing *variability in time*, e.g., superseding revisions, as opposed to *variability in space*, e.g., co-existing variants of software objects. Albeit, *branches* in the revision graph explicitly allow for co-existence. This immediately raises the idea of organizing features in specific branches. Correspondingly, VCS may provide a *tool-driven compile-time* approach for SPL implementation. Since the *merge* operation by which products are derived can be considered as a special kind of transformation, the solution is here categorized under *transformational variability*.

Notice that we here consider VCS as an implementation technique, without aiming at reasoning about the integration of historical and logical versioning; this is considered in Section 6.4.

A specific VCS-based *reuse and variation approach* is presented in [SH04]. In [Ap+13b], an instantiation of this approach is described, making a distinction between four types of branches in a multi-level revision graph:

– In the *development branch*, the platform of the product line is evolved.
– *Feature branches* are provided to develop specific features of the product line independently, each ensuing from the development branch.
– *Release branches* correspond to customer specific product variants. They are created by merging the desired features' branches into offsprings of the development branch. The configured product may be further customized here in the course of *application engineering*.

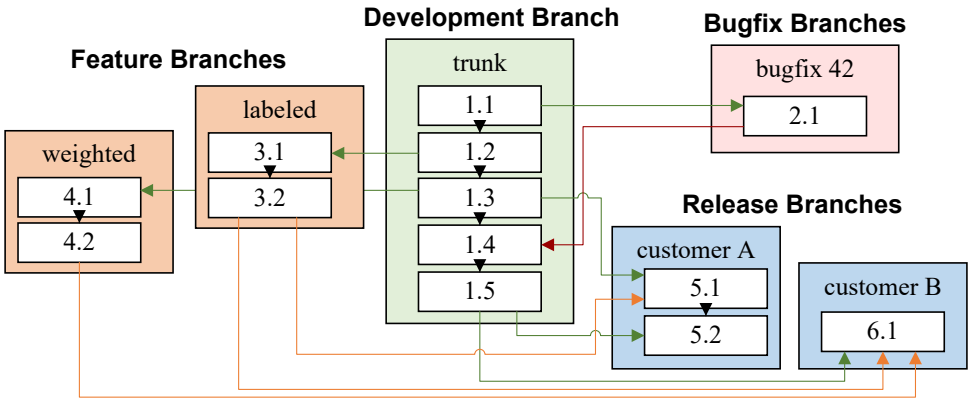


Figure 5.9: Using version control systems as an SPL management mechanism.

– Temporary *bugfix branches* are provided to separate maintenance changes from evolutionary changes. They may either emerge from the development branch, in order to realize a bugfix relevant to all product variants, or from feature branches, when the respective bug is related to a specific feature.

The same distinction is applied in the example presented in Figure 5.9, which shows a revision graph for a hypothetical excerpt of the VCS adaptation of the Graph product line. In the trunk, an initial version of the platform is created, before corresponding branches for features labeled and weighted are forked. Both feature branches are evolved concurrently. After that, the first release is derived for customer A by merging the development branch with labeled. In the meantime, a bugfix is applied to the main development branch, such that the latest revision of the development branch must be merged into the release for costumer A. Last, another product variant is derived, resulting in the release for customer B, a weighted and labeled graph.

This minimal example already demonstrates the downsides of using state-of-the-art VCS for SPLE. On the one hand, it is required to repeatedly merge different variants of the product line, be it for integration of features or for maintenance purposes. On the other hand, release branches contain redundant information that can be derived from the trunk and from respective feature branches. Another disadvantage is shared with build systems: by release branches, the VCS approach ultimately implements *extensional* versioning (although the initial construction of variants is performed by combination of independent branches).

Clone-and-Own Approaches. The variant management platform *ECCO* (Extraction and Composition for Clone-and-Own) presented in [LELH16] follows an extractive approach to gradually transform a set of related product variants with a common origin into an SPL. Once migrated to the platform, new products can be composed based on a specification of their properties. Internally, these specifications are mapped to a *feature algebra*. Rather than actually applying fine-grained variant management in the sense of intensional versioning, the approach relies on *reference variants*, which constitute the variant that "best matches" a selected configuration. On the one hand, this noticeably reduces precision when compared to truly transformational approaches; maintenance problems resembling those VCS-based

approaches (see above) may also occur. On the other hand, the clone-and-own approach has gained widespread acceptance as it is easy to install in a retroactive way [RCC13].

5.4.5 Multi-Paradigmatic Approaches

In the literature, multi-paradigmatic software solutions for SPLE support have been presented. For instance, *FeatureIDE* [Thü+14b] is an extensible FOSD framework that supports different annotative approaches by offering integration with several preprocessor tools. Furthermore, compositional (AHEAD, FeatureHouse, and AspectJ), as well as transformational paradigms (delta-oriented programming) are supported.

Similarly, *pure::variants* offers – besides annotative source code support – several extensions for different kinds of programming and modeling languages forming the platform.

With *PEoPL* (Projectional Editing of Product Lines), a multi-paradigmatic editor for source code centric SPLs is presented in [BPB17]. The tool is capable of representing the same product line in different forms (projections), which allows the developer to switch in between different implementation paradigms for the development of the same product line. Altogether, five projections are supported: textual annotation (using preprocessor-like directives), visual annotation (providing color-coding for different features), module projection (using refinements as in FOP), blending projections (corresponding to partially filtered editing), and variant projections (fully filtered editing).

5.5 Feature Interaction and Product Well-Formedness Analysis

After having explained several concrete SPLE implementation approaches, let us now revisit two important challenges relevant in the context of this thesis, namely *feature interaction* and *product well-formedness analysis*. Intentionally, solutions to the challenges remain underrepresented in this section; these are addressed in the related work sections in the chapters of Part IV.

The here examined challenges may arise for almost all approaches discussed in Section 5.4. For the sake of simplicity, we assume a product line approach based on annotative variability using preprocessors for the explanations below.

5.5.1 Feature Interaction

Interaction between two features occurs whenever the combination of two features leads to unexpected behavior, whereas the activation of both features in isolation exposes the correct functionality. A prominent example is located in the telecommunications domain, where two features – CallWaiting and CallForwarding – may interact [CM00]. In case both features are active, it is not clear whether a waiting queue shall be installed or whether the call shall be forwarded in case the called person is busy.

In [Ap+13a], two types of feature interaction are distinguished:

External Interaction. In analogy to *external variability*, this kind of interaction is apparent to the user of an affected product in terms of conflicting features, such that he/she is able to phrase the unexpected behavior in domain terms.

```
1  public class Edge {
2     #ifdef WEIGHTED
3     private double weight;
4     public Edge() {
5        weight = 0.5;
6     }
7     #endif
8     #ifdef LABELED
9     private static int id = 0;
10    private String label;
11    public Edge() {
12       label = "Edge_" + id++;
13    }
14    #endif
15 }
```

Listing 5.1: An example of static feature interaction.

Internal Interaction. This kind of feature interaction is detected, e.g., in the course of static analysis of a derived product during application engineering. An example is depicted in Listing 5.1: The constructor of class `Edge` is redefined for both weighted and labeled graphs in isolation, but in case both features are selected, the compiler will detect an invalid redefinition, since a constructor with the same signature already exists.

Current research focuses on both the *detection* and the *resolution* of occurrences of external feature interaction, where precise automated detection is considered as the more challenging task. “There is no single strategy that can be claimed as general, scalable, and production-ready, yet” [Ap+13b, p. 217]. Feature interaction detection is complicated by two factors. First, the number of possible occurrences of pair-wise feature interaction grows quadratically with the number of optional features introduced. Second, each possible case of interaction must be tested in a suitable product variant, but it is hard to guarantee that the analysis performed in these candidates is representative.

Concerning the *resolution* of identified feature interaction, there are two possible realms to consider: the problem space and/or the solution space. On the one hand, an identified feature interaction may stem from a missing constraint in the variability model. Returning the telecommunications example, features CallWaiting and CallForwarding might be declared as mutually exclusive. On the other hand, particularly structural interaction often reveals inconsistencies in the multi-variant code. When referring to the example of Listing 5.1, the two constructors should be made mutually exclusive by corresponding `#ifndef` directives. In addition, a new constructor for the combination of both features should be introduced. In approaches based on compositional variability, unintended feature interaction is often a symptom of poor modularization.

5.5.2 Product Well-Formedness Analysis

None of the compile-time approaches discussed in Section 5.4, regardless of whether being based on compositional, transformational, or annotative variability, can guarantee that the compiler will accept derived product variants as syntactically correct. This contradicts with the goal of automating application engineering to the greatest possible extent. The problem of *product well-formedness* is addressed by many current research activities, resulting in a plethora of problem statements and even more solutions based on different analysis strategies. An exploratory survey is provided in [Thü+14a], where four categories of approaches are provided:

Product-Based Analysis. This corresponds to the *brute-force* approach of checking the consistency of each valid product separately for concluding the well-formedness of the overall product line. Since the number of valid products grows exponentially with the number of features, scalability problems soon arise.

Sample-Based Analysis. Rather than considering all valid variants for the checking of product well-formedness, only a small subset is selected. The selection should conform to a criterion such as *pair-wise coverage* [Jay+07].

Feature-Based Analysis. Essentially a special case of sample-based analysis. Feature-based approaches aim at analyzing the well-formedness of artifacts belonging to each feature in isolation. This incorporates a convenient compromise between scalability and coverage. Albeit, by being scoped by one feature only, a-priori detection of possible occurrences of feature interaction is not guaranteed.

Family-Based Analysis. According to [Thü+14a, p. 6:15], family-based analysis "(a) operates only on domain artifacts and (b) incorporates the knowledge about valid feature combinations". Thus, the product line is checked as a whole by variability-aware analysis methods before any application engineering takes place. This requires to adopt the respective validation techniques, but is both scalable (in contrast to product-based) and guarantees the correctness of *all* variants (as opposed to sample-based or feature-based analyses) beforehand. To be able to apply a family-based approach, the corresponding validation technique needs to be lifted to multi-variant checking, which imposes a considerable conceptual and technical obstacle.

Orthogonal to the question above – which artifacts are passed to an analysis – different techniques can be employed for actually performing the analyses. Besides classical verification and validation methods such as symbolic execution or testing [Som06], *model checking* [GLS08], or *automatic theorem proving* [Thü+12] are frequently employed in the SPLE context.

5.5.3 Feature Model Consistency

A related consistency problem addresses the question if the feature model is *satisfiable*, i.e., whether or not there exists at least one feature configuration that accords to the constraints defined in the feature model [Hei09; MWC09].

The problem of feature model consistency is addressed in Sections 9.4 and 13.3.3.

5.6 Bottom Line

SPLE is an increasingly popular software development approach relying on organized reuse of a platform of artifacts in order to quickly and efficiently derive customer-specific software products on demand. For documenting the variability, i.e., the commonalities and differences of product line members, feature models have been established as a precise yet simple to understand formalism. SPL development processes are predominantly oriented toward the proactive rather than to the extractive or reactive adoption paths, and consequently, have been influenced by plan-driven rather than by agile software development processes.

SPL implementation techniques can be distinguished into compositional, transformational, and annotative variability. For the developer, all approaches imply a significant increase in complexity when compared to single-system development. Therefore, we argue that an important aspect of research in SPLE tooling should be a reduction of this complexity.

Last, we have learned that the problem of product well-formedness analysis – which is motivated, among others, by unforeseen feature interaction – cannot be solved in both a precise and a computationally affordable way, such that compromises like sample-based analysis appear to be most feasible.

Part III

Points of Intersection

The promise of large scale software reuse has been a goal of the software industry since the dawn of software engineering in the late sixties. However, it has proved an elusive goal.

HASSAN GOMAA (2005)

Chapter 6

Integrating Disciplines

Abstract

The background provided in the previous chapters is supplemented by a review of pair-wise combinations of MDSE, SPLE, and SCM (see intersection sets in Figure 6.1). Model-driven product line engineering is motivated by the common goal of increased productivity; different approaches relying on compositional, transformational, or annotative variability are compared. Model version control subsumes approaches and techniques to version control that operate on the level of abstraction of models, promising more precise and more consistent collaborative model development. The combination of SPLE and SCM is considered in a twofold way. First, product line version control deals with the co-evolution of variability model and versioned product. Second, state-of-the-art approaches to the integration of historical and logical versioning, having their origins in SCM, are classified.

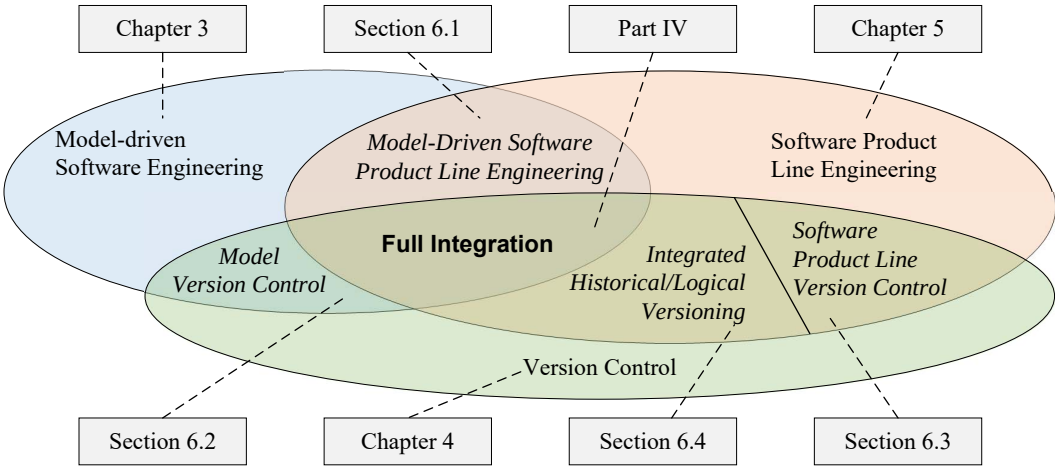


Figure 6.1: Illustration of the integrating disciplines considered in this thesis.

Contents

The contents of this chapter are reproduced in a condensed form in [SW17c].

6.1 Model-Driven Software Product Line Engineering

Combining MDSE and SPLE promises to increase the productivity of software engineers in a twofold way. On the one hand, just like feature models abstract from the problem space, *domain models* are a suitable abstraction for the solution space, such that the platform and the variability model are situated at the same conceptual level. On the other hand, MDSE and SPLE share the goal of highest possible *automation*. When considering the classical distinction of SPLE into domain engineering and application engineering, considerable economical savings can be achieved by moving as many repeated manual development activities as possible from AE to DE [CN01], reducing AE to an automated configuration routine.

Model-driven (software) product line engineering (MD(S)PLE) [Gom05] is becoming a more and more populated research field. This section aims at identifying the major research challenges as well as at outlining the state of the art of currently available technical solutions. Corresponding approaches based on annotative variability are a subject of Section 6.1.1, before compositional and transformational variability are considered in Sections 6.1.2 and 6.1.3, respectively.

6.1.1 Annotative Variability

As explained in Section 5.4.2, annotative variability assumes that the platform is provided in form of a *multi-variant* artifact from which fragments corresponding to deselected features are conditionally removed. When transferring this to MDSPLE, the platform corresponds to a *multi-variant domain model* (MVDM) that is an ordinary instance of a specific metamodel (cf. Section 3.3). In order to perform a fully automated product derivation, it is necessary to establish *traceability links* between elements of the MVDM and the variability model.

Language-Internal Variability. In [Gom05], Gomaa et al. introduce the method *product line UML based software engineering* (PLUS). In [GS04], the corresponding tools for variability management and automatic product derivation are presented.

PLUS is based on an extension of the modeling language UML. To this end, several *stereotypes*[1] have been introduced which can be attached to model elements. These include:

«kernel» The element is mandatorily included in every product.

«optional» The element may be either selected or deselected for specific products.

«default» Indicates that an element is the default variant for a specific variation point, which is selected automatically unless specified otherwise.

«variant» Such elements can be selected to replace the default element for a specific variation point.

Products are derived by defining *views* on the multi-variant domain model, which resolve all configuration decisions by a boolean decision for optional elements and by selecting a representative for each variation point.

Intertwined Domain and Variability Model. A family of approaches to MDSPLE based on annotative variability conceptually separate the feature model from the domain model, however, the connection between problem and solution space is realized within the domain model in an *intertwined* way. This requires corresponding extensions to the modeling language in order to provide for the definition of variation points and variants.

For example, *Clafer* is a "(class-based) meta-modeling language with first-class support for feature modeling" [BCW11, p. 102], e.g., the language mixes feature model with UML class diagram concepts. A feature is not only represented by a class, but also shares its structural features, such that the feature model hierarchy corresponds to the object composition hierarchy defined in the intertwined domain model. Similarly, feature group constraints are equivalent to multiplicities of composition references. For consistency checks, Clafer models are internally mapped to the formal specification language *Alloy* [Jac06].

Explicit Mapping-Based Approaches. MDSPLE has been addressed by several tools and approaches following an *explicit mapping-based* (cf. Section 5.4.2) paradigm. Here, the

1 Stereotypes are language inherent extension mechanisms that dynamically add domain-specific functionality to specific UML elements; see [OMG15].

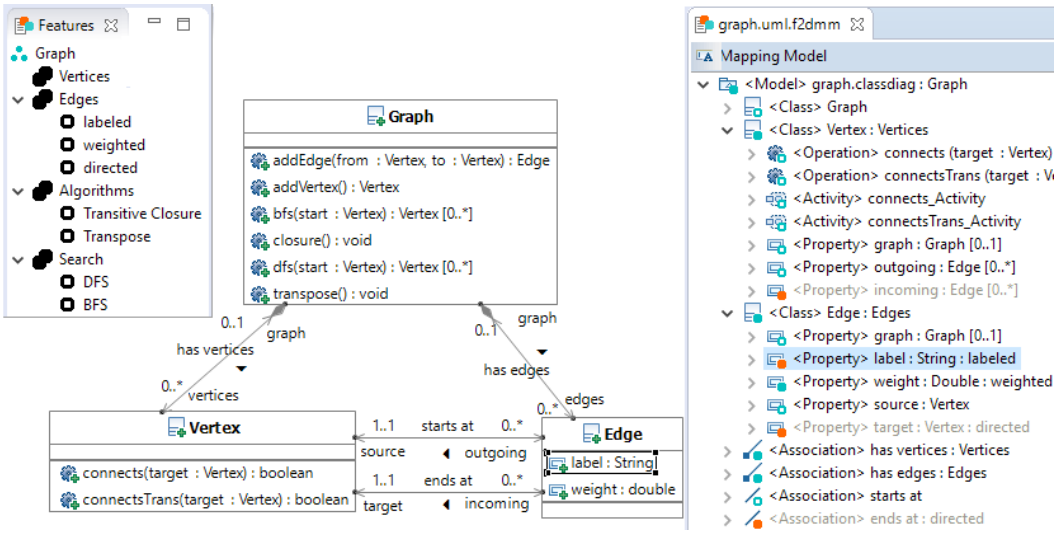


Figure 6.2: Mapping-based annotative variability with FAMILE.

feature model and the MVDM are kept orthogonal by introducing a distinct *mapping model*, which interconnects problem and solution space. In its most general form, the mapping model assigns *presence conditions* – boolean expressions on the variables defined in the feature model, implementing traceability links – to elements of the domain model.

This approach has been realized, among others, by the tools *FeatureMapper* [HKW08], by the *Enterprise Architect* edition of *pure::variants*[2], as well as by *FAMILE* [BS12b] and its UML-based predecessor *MODPL* [BW12].

The workflow implied by mapping-based MDSPLE tools is exemplified by the screenshot shown in Figure 6.2, which was taken from the FAMILE tool while being applied to the running *Graph* example. On the left hand side, the feature model is depicted. In the main window, the MVDM may be edited in concrete syntax. The right hand side contains the *mapping*: Here, the user may arbitrarily edit the presence condition associated with the selected element of the domain model, which is redundantly represented in tree syntax. In this screenshot, a feature configuration was loaded for previewing the product to be derived.

Product derivation during application engineering is performed straightforwardly in mapping-based MDSPLE approaches. Given a specific feature configuration, a *conditional copy* of the MVDM is created. Elements are copied if they do not have any presence condition assigned (implicitly assuming the expression *true*) or if their presence condition evaluates to *true* given the bindings defined in the feature configuration.

Template-based Approaches. In [CA05], Czarnecki et al. propose a *template-based* approach to realize annotative variability based on the connection of feature models and UML activity or class models. Variability is resolved at *compile-time* and in a *tool-driven* way. Being based on *cardinality-based feature modeling* [CHE05], the feature model may also contain *attributes* that serve as input for templates defined in the domain model.

2 http://www.pure-systems.com/products/pure-variants-for-enterprise-architect-286.html

Like in explicit mapping-based approaches, the platform of the product line is based on a multi-variant domain model (MVDM), specific elements of which may be augmented with boolean *presence conditions*. In addition to those, string-valued *meta-expressions* are allowed for individual values of specific model elements, e.g., the return type of a method. Meta-expressions may contain, among others, boolean case differentiations and references to feature attributes. The expressions are evaluated after applying a feature configuration. This potentially raises consistency issues, which are here detected and resolved in a dedicated post-processing step; see [CA05].

6.1.2 Compositional Variability

When transferred from SPLE to MDSPLE, compositional variability can be realized by a *core model* that constitutes the architectural scaffold of applications being part of the product line. This core model is extended by conditional model fragments/components[3] in order to derive specific applications. Several approaches have been established for the definition of these extensions. Unless stated otherwise, all approaches listed below belong to the categories *tool-driven* and *compile-time*.

Aspect-Oriented Modeling. *Aspect oriented modeling* (AOM) [Wim+11] denotes the transfer of concepts of aspect-oriented programming – aspects, join-points, pointcuts, advice – to the abstraction level of models. Specific model elements can be declared as join-points, which serve as extension points for fragments; these in turn are realized by means of aspects that encapsulate several units of advice.

Different approaches exist for specifying transformations that eventually weave the defined aspects into models. In the tool *MATA* [Whi+09], *attributed graph grammars* [Tae04], a graph-based formalism for model transformations, are provided for this purpose. Furthermore, there exist specific model aspect weaving languages such as *XWeave* [GV07].

Superimposition-Based Composition. In [Ap+09b], an approach for product derivation based on compositional variability is described. The concept of transformation is hidden from the user. As a replacement, fragments are merged into the core model, such that during product composition, sets of fragments rather than transformations have to be selected. To this end, *superimposition*[4] is introduced as a model composition technique. "Entities are matched by their name and type. [...] When composing two entities, the union is taken from their members" [Ap+09b, p. 7f.]. This way, the connection between fragments and the core model is established by cloning the parent elements where a fragment is to be inserted.

Layered Models. The tool EASEL [HJH06] introduces the concept of *layered class diagrams*, allowing to intensionally create different versions of UML classes. The approach can be transferred to MDSPLE inasmuch as different versions of a model can be created by merging a selection of *layers*, each realizing another individual optional design decision.

3 By convention, *fragments* are referred to as fine-grained units of models, e.g., UML classes, associations or attributes, whereas *components* denote coarse-grained architectural units that encapsulate a piece of functionality.

4 In approaches based on annotative variability, *superimposition* denotes the multi-variant artifacts. Here in contrast, superimposition denotes the *operation* that composes several models and/or fragments.

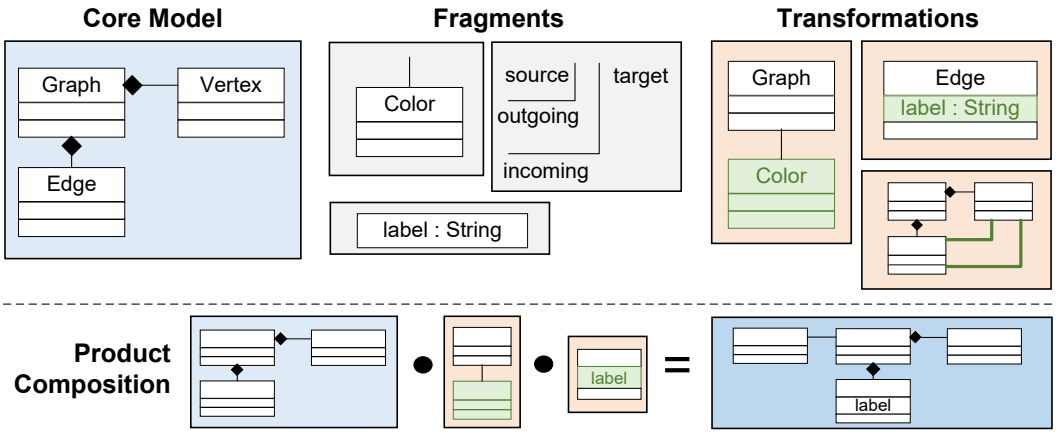


Figure 6.3: Achieving variability in MDSPLE by model transformations.

6.1.3 Transformational Variability

Figure 6.3 sketches how transformational variability can be carried out on top of *model transformations* (see Section 3.7). In this case, the platform comprises the core model, fragments, and model transformations. Fragments are defined in terms of model objects, or a small subset thereof, instantiated from the domain metamodel. Moreover, model transformations define how these fragments are attached to the core model. Product derivation consists in the selection of appropriate model transformations, which are then applied to the core model in a chained way.

General Purpose Model Transformations. A comparably coarse-grained approach to transformational variability based on model transformations is presented in [Hau+04]. In particular, this approach builds on the principles of model-driven architecture (cf. Section 3.1). Variability is resolved already at the level of the *component independent model* (CIM). Model transformations are used not only to compose specific instances of the product line at CIM level, but also to further refine those at PIM (platform independent model) and PSM (platform specific model) levels.

In the literature, several approaches can be found that combine model transformations with other product composition techniques, as well as techniques primarily relying on annotative variability. *Product Line Behavioral Synthesis* (*PLiBS*) [ZHJ04] extends the PLUS approach (see above) in a twofold way. First, behavioral modeling is supported. Second, an additional variability stereotype «virtual» is introduced for placeholder elements, which are bound to specific model fragments in a pre-processing step. The integration is controlled by model transformations. Moreover, in [TP11], a UML profile approach is presented. To this end, PLUS is extended by model transformations based on ATL.

Common Variability Language. A similar approach has been realized by Haugen et al. [Hau+08], who propose two distinct approaches for mixing the domain language with a variability language. First, in an *amalgamated* approach, languages are combined at metamodel level by creating a new, variability-aware metamodel that imports both

languages. Second, when using the *separated-languages* approach, the domain language remains unmodified. Rather, its instances are fragmented and embedded within a generic variability-aware *Common Variability Language* (CVL) [OMG12]. In spite of requiring less overhead for tool set-up, the separated-languages approach offers a coarser object granularity than the amalgamated approach.

Explicit Delta Languages. In [Zsc+10], a bootstrapping approach to transformational variability based on a family of composition languages is described. According to the authors, general purpose model transformations are too unspecific for the needs of MDSPLE. They address this issue by "a family of languages for variability management", VML* [Zsc+10]. From this language, specific dialects – *language instance descriptions* – can be derived that allow to add variability to specific types of models. When referring to Figure 6.3, transformation specifications are written according to the respective language instance descriptions. Fragments can either be imported from existing models or defined in the transformation in an in-place fashion. Each transformation is associated with a specific *variant*, which can by any combination of features defined in an external feature model.

A similar approach has been realized by [SSA14b] in the tool *DeltaEcore*. Here, the units of composition are referred to as *deltas*, which can be used to express both variability in time and variability in space. A delta is specified in a *delta dialect* that is derived from a common base delta language and adds metamodel-specific operations in a similar way as a language instance description in VML*. An example for the usage of DeltaEcore is provided in Section 7.1.4.

Implicit Delta-Based Approaches. An *implicit* delta-based approach, relieving product line developers from having to explicitly specify transformations, has been realized by the tool *SiPL* presented in [Pie+15]. As already explained in Section 5.4.3, a *delta module* connects a presence condition, which refers to the variables defined in the feature model, to a delta, which corresponds to a sequence of operations applied to a core domain model. In contrast to explicit delta language based approaches, however, SiPL derives the delta from an a-posteriori analysis involving two versions of a model representing the states before and after applying the corresponding delta. The analyzed deltas are managed by the tool in a *delta module set*, from which in turn customized products may be automatically derived by applying the selected deltas to a core variant of the product.

6.2 Model Version Control

At first glance, the combination of model-driven software engineering and version control seems unproblematic. After all, models have a persistent textual representation relying on XMI (cf. Section 3.3) or on concrete textual syntax (e.g., using Xtext; see Section 3.6), and may therefore be considered as ordinary text files from the perspective of a VCS.

Applying line-oriented versioning to structured data, however, has proved inadequate when it comes to differencing and merging multiple versions of a model. Rather than being presented insertions and deletions of XMI-based text lines, users would prefer understanding the differences between two model versions in terms of the used modeling language, e.g.,

operations such as “rename UML class”. Even worse, three-way merging XMI files in a line-oriented way may produce well-formedness violations that destroy the compatibility of models with their editing tools, while modelers are forced to resolve merge conflicts in a text-oriented tool such as *diff3*.

These inadequacies gave rise to the discipline *model version control* (MVC), which is an active research area involving a multitude of conceptual and technical challenges [ASW09]. Generally speaking, MVC lifts the *object granularity* of versioning from text files being comprised of text lines up to models being composed of model objects. When using the conceptual notions introduced in Section 4.2.3, the *product space* corresponds to the versioned model(s), *software objects* are equivalent to *model objects*, and composition relationships are broken down to the containment structure defined by the model.

This section first introduces two sub-problems implied by model versioning. The subject of Section 6.2.1 is *model differencing*, which includes the detection of commonalities between two versions of a model as well as the internal persistence and external representation of the derived differences. In Section 6.2.2, problems and solutions of *three-way model merging* are presented. The section is concluded with a presentation of fully-fledged *model version control systems* in Section 6.2.3.

6.2.1 Model Matching and Differencing

In analogy to text-oriented approaches, a comparison of two versions of a model includes two steps: matching and differencing. These are visualized in Figure 6.4 using the running *Graph* example. The commonalities identified by the matches may be understood as a list of *correspondence links* (visualized as blue dashed lines), each connecting an object of the first version with a matching object of the second version of the model. From such a list of commonalities, differences are deduced by identifying those elements that have no correspondence in the opposite version. There exist multiple forms of difference representation—in Figure 6.4, a textual difference report is presented that describes the

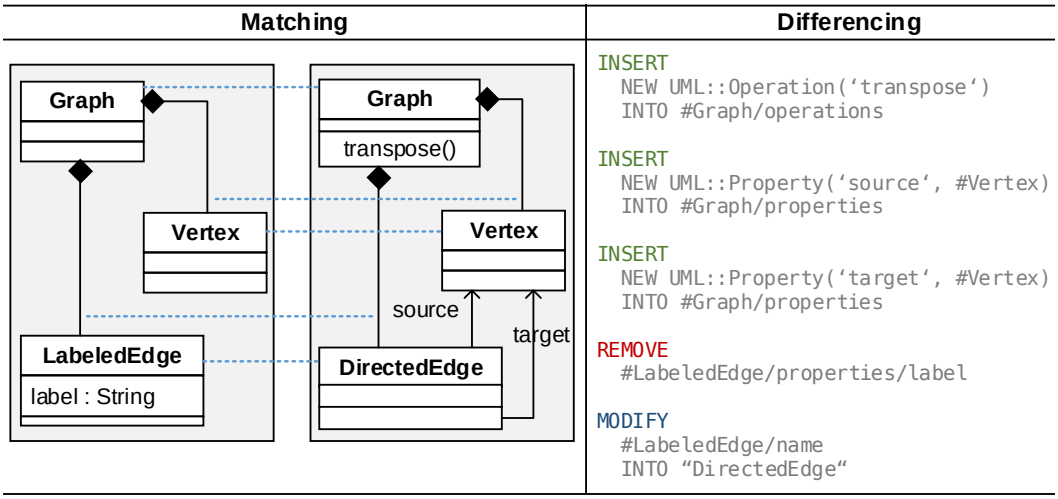


Figure 6.4: General problem statement of model matching and differencing.

changes in terms of insertions, deletions, and modifications of values of structural features of model objects.

Edit Logs. Not at least due to the easier implementation of this approach, many model-based version control systems rely on a *log-based* paradigm. In these cases, the edit operations carried out by the user are recorded, such that a precise edit log is available for pairs of versions immediately following each other in the revision graph. In case other pairs of versions are to be compared, the path of edit scripts connecting them must be combined in an adequate way. Log-based versioning has been implemented, among others, in the MVC systems *EMF Store* [KH10] and *CoObRA* [SZN04] (see Section 6.2.3).

Universally Unique Object Identifiers. Many modeling tools automatically assign *universally unique identifiers* (UUIDs) to all model objects upon their creation. UUIDs are transparent to users and remain unmodified during the lifetime of an object. Assuming that the model versions to compare have emerged from a common ancestor version, UUIDs may serve as a correct and reliable criterion for the identification of model objects. On the contrary, they cannot detect corresponding objects inserted after the model versions diverged. Furthermore, not all modeling tools rely on UUIDs. For example, EMF model instances based on Xtext (cf. Section 3.6) cannot support this approach as the concrete syntax is persisted, which impedes managing UUIDs transparently to the user. In practice, many model comparison tools, including *EMF Compare* [BP08], use UUIDs if available – rather than exclusively relying on them – in order to improve the matching.

Heuristic Matching. If neither edit scripts nor UUIDs are available, the edit operations carried out by the user must be deduced from the available versions in a *comparison-based* fashion. Since a result calculated this way does not necessarily reproduce the actual editing history, the corresponding methods and algorithms are *heuristic*. Given that models are structured graph-like entities, the *sequence comparison algorithms* presented in Section 4.3.3 are not applicable. Rather, a few families of *model comparison algorithms* have been established, each relying on different heuristic criteria.

– *Tree-based* matching has been investigated way before the advent of MDSE on the basis of "hierarchically structured information" [Cha+96]. In general, trees are much easier to compare – e.g., using top-down or bottom-up or a mixture of both strategies – than arbitrary graphs. Albeit, when reducing MOF model instances to their containment links and pruning all links instantiated from non-containment references, they can be considered as trees, too. Tree-based comparison of models has been considered, among others, in [BPV10a].
– *Similarity-based* matching aims at identifying pairs of model elements in order to obtain a set of correspondences whose elements expose the highest possible degree of similarity. Since the potential number of pairs to be considered grows quadratically with model size, methods following this approach differ with respect to optimization. The MOF-compliant tool *SiDiff* [KWN05] first applies a tree-based bottom-up approach, where comparison is restricted to types and attribute values of objects. Thereafter, in a top-down phase, links are taken into account for global optimization. Similarly, *UMLDiff* [XS05], which is restricted

to the comparison of UML class diagrams, relies on the matching of identifiers (e.g., class names) in a first phase and takes structural information into account secondly.

– *Edit-distance based* methods as supported by *ModDiff* [Uhr11] take as optimization criterion for heuristic matching the size of the edit log derived from the matching. This in turn results in a higher sensitivity for renaming operations, which tend to be overseen by similarity-based approaches.

Difference Computation and Visualization. Once the matching between two versions of a model has been calculated, differencing is, at first glance, the less sophisticated part: Model objects part of the modified version that have no matching partner in the original version are classified as insertions; deletions are detected vice versa.

As already pointed out in Section 4.3, the purpose of differences is twofold in version control. On the one hand, they may serve for the internal persistence of model artifacts in the sense of *directed deltas*. This is relevant for a subset of model VCS presented in Section 6.2.3. On the other hand, differences support the user in understanding and comprehending changes performed, e.g., by others. To this end, adequate *visualization* for model differences is required. Among other tools, *EMF Compare* [BP08] provides for a generic tree-based model representation that augments the actual difference report; an example is shown in Figure 6.5.

As sketched in Figure 6.4, however, differences deduced this way may reside at a too low level of abstraction in order to be meaningful to modelers. Several approaches described in the literature, including [RV08; BPV10b; KKT13], deal with lifting the semantical level of differences in order to describe the change as a sequence of *refactorings* – e.g., moving copies of a UML attribute from two classes into a common base class – rather than in terms of fundamental edit operations such as insertion or deletion. Such *high-level model differences*, however, are not a core topic of this thesis.

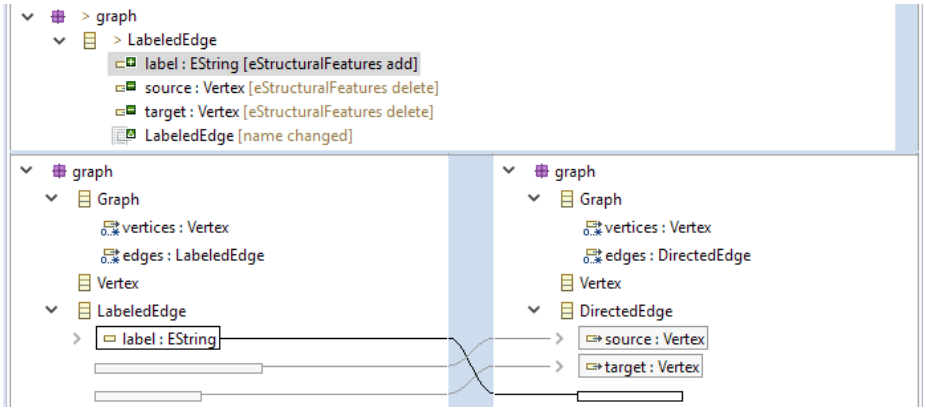


Figure 6.5: A difference report generated by EMF Compare.

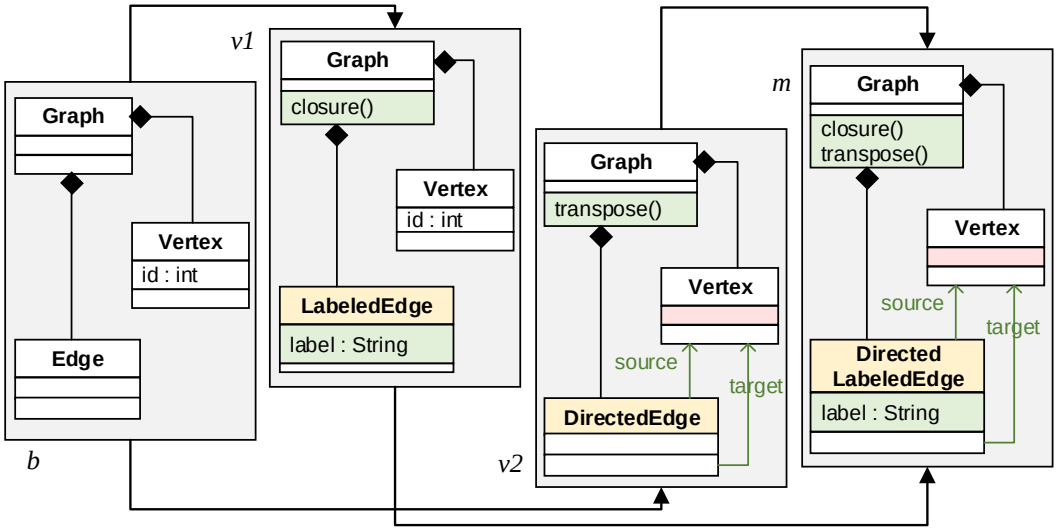


Figure 6.6: An example of three-way model merging.

6.2.2 Three-Way Model Merging

More than comparison and differencing, transferring the three-way merge problem from text-oriented to model versioning demands for new approaches and methodologies. As mentioned before, not only is the merge result required to include both changes made in the alternative versions, but emphasis is put on the merge result being *consistent*, which includes both its syntactical well-formedness and its semantical correctness.

The three-way model merging scenario is exemplified in Figure 6.6. From a base version of a *graph* metamodel, two alternatives v_1 and v_2 are derived. In v_1, an operation closure is introduced to class Graph. Furthermore, Edge is renamed to LabeledEdge and an attribute label is added. In the meantime, v_2 is created by adding a new operation transpose to Graph. The attribute id of class Vertex is removed. Also, class Edge is renamed, now to DirectedEdge. In addition, two references source and target, both connecting instances of Edge to Vertex, are added. The expected merge result m incorporates all element additions and removals. In particular, Graph contains both new operations in a user-defined order. Furthermore, the contradicting renamings of class Edge were combined in the course of a user-directed conflict resolution: DirectedLabeledEdge.

Below, different categories of three-way merging approaches for models are categorized, with a focus on their applicability to MOF-based model instances. With respect to conflicts detected by specific tools, we distinguish between *context-free* conflicts – several values competing for the same structural feature – and *context-sensitive* conflicts, which arise from applying different yet contradicting operations in different versions, e.g., a new reference to versus the deletion of an object. More precise categorizations of model merge conflicts are provided in [AP11; Wes14].

Fallback to Text-Oriented Merging. A first category of solutions to three-way model merging relies on *text-based merging* as performed by ordinary VCS such as Subversion or

Git. To this end, the model instances to be merged are mapped to text artifacts in a suitable way—XMI serialization does not come into question. In [BLF14], an approach that relies on representing atomic pieces of information by one text line each is shown. This way, model-level changes are mapped to text-level changes such as insertions and deletions of text lines. Merge conflicts are reported to the user either in terms of conflicting insertions or deletions of text lines, or by the (textual) model editor that represents the merge result. Altogether, models and code are merged uniformly, but such approaches do not consider more specific requirements such as context-sensitive correctness.

Structure-Oriented and XML-Based Merging. One step closer to the abstraction level of models are *structure-oriented* and *XML-based* three-way merging methods.

A generic tool for structure-oriented merging of documents based on *syntax graphs* has been presented in [Wes91]. The tool relies on UUIDs being attached to all syntax nodes of the input versions. During three-way merging, context-free and partly context-sensitive merge conflicts are detected and resolved. Although targeting structured text documents in general, the underlying approach is transferable to MOF-compliant models.

There exist a multitude of tools in the literature that address three-way merging of different versions of an XML-document. One representative is *3dm* [Lin04], which internally maps the hierarchical and ordered XML structures of each version to a set of *facts*, triples of the form (parent, child, successor). Three-way merging is effectively applied to the set of facts; conflicts are reported in case the parent or the successor of an element are contradicting. Since XMI is XML-based, 3dm can be applied for three-way merging MOF instances, however, context-sensitive conflicts remain undetected.

Change-Based Merging for Comparison-Based Versioning. A state-based three-way merging approach specifically targeting MOF instances has been developed by Alanen and Porres [AP03]. The algorithm assumes UUIDs and considers only a small subset of MOF, namely instances of classes, attributes, and unidirectional references. From a pairwise comparison of the alternative versions v_1 and v_2 with their common ancestor b, two change sequences c_1 and c_2 are derived. These are combined into a single merged change c_m, which is applied to b in order to create the merged version m (cf. Figure 4.9). This algorithm, however, does not correctly handle containment links and conflicts in the order of multi-valued structural features.

The aforementioned tool *EMF Compare* [BP08] has three-way merge functionality integrated into the comparison view. Differences are detected based on a preceding matching of the base version with both alternative versions. The tool allows to propagate changes between the two model versions in order to manually create a merged model version. Conflicting changes are signaled to the user, who may decide either to override the local change or to ignore the incoming change. Many context-sensitive conflicts, however, remain undetected by this tool.

In the context of the *AMOR* project (Adaptable MOdel veRsioning) [Alt+08], several components for model version control were developed, including a three-way merge tool based on the algorithm presented in [Tae+14]. From a matching obtained by EMF Compare, change sets c_1 and c_2 are deduced in the form of *graph modifications*. For the detection of

context-free and context-sensitive conflicts, specific patterns of inconsistent graph modifications have been defined in [Tae+10]; this set may be extended in a language-specific way by graph patterns or by OCL constraints. In the case of conflicts, a preliminary merge result is presented to the user. Conflicts are made visible in the form of model annotations, which help the user resolving them non-interactively in an a-posteriori step.

State-Based Model Merging. A purely state-based three-way merging algorithm specific to EMF model instances is presented in [Wes14]. The algorithm is generic with respect to the question how correspondences between objects of different versions are calculated, and therefore supports UUIDs as well as different matching strategies. The presented merge algorithm, which is applicable also to two-way merging, applies context-free and context-sensitive *merge rules*. Rather than relying on the indirection of change sequences, set-theoretic considerations are made when determining the objects to be included in a merged version. For instance, the context-free merge rule for the calculation of the set of objects O_m to be included in the merged version is implemented by the following set formula:

$$O_m = (O_1 \cup O_2) \setminus ((O_b \setminus O_1) \cup (O_b \setminus O_2)) \tag{6.1}$$

Here, O_b, O_1, and O_2 correspond to the object sets of the base version b and the alternative versions v_1 and v_2, respectively. Analogous rules are applied, e.g., for the values of structural features and for determining the container of an object. Context-sensitive rules are defined by patterns in the preliminary version of the merged model that detect, e.g., dangling references or objects having multiple containers.

The algorithm of [Wes14] has been implemented in the tool *BTMerge* [SUW13b], which performs a consistency-preserving three-way merge for arbitrary EMF-based model instances. The preliminary merge result is presented to the user in a dedicated conflict resolution tool, where conflicts are reported and may be resolved *interactively*. The data structures underlying BTMerge rely on a multi-version model representing the superimposition of the input models. The multi-version representation of ordered collections is

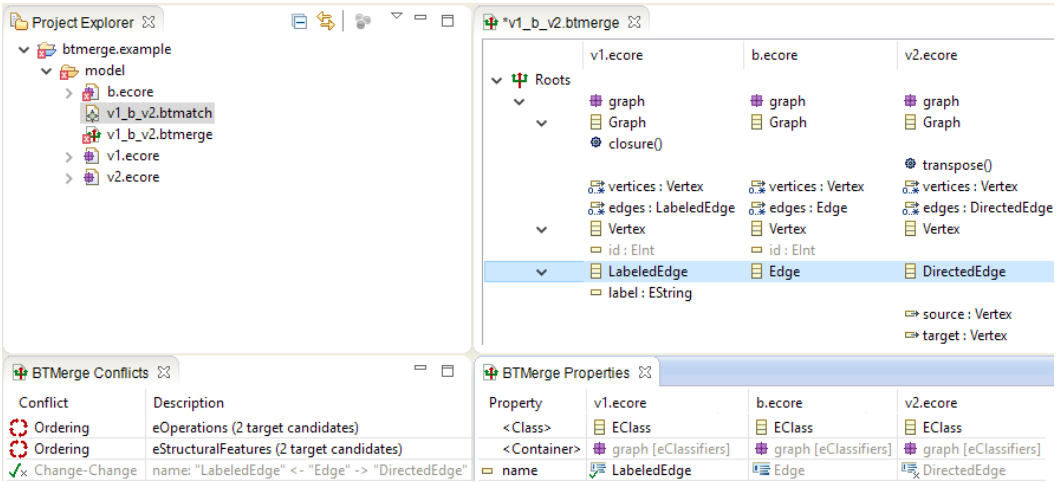


Figure 6.7: Applying BTMerge's conflict resolution tool to the merge scenario.

discussed in Section 8.2.

In Figure 6.7, the application of BTMerge to the three-way merge scenario introduced in Figure 6.6 is illustrated. The tool presents the three versions v_1, b, and $v2$ of the model in a three-column tree editor, thus the superimposed intermediate model is directly presented to the user. In the bottom left window, the detected conflicts are shown: the contradictory renaming of class Edge and the undefined mutual order of, among others, the new operations closure and transpose.

Operation-Based Merging. Unlike text-oriented VCS, many model VCS follow a log-based paradigm, such that performed editing operations are recorded in the IDE. Correspondingly, three-way merging is applied in an operation-based way.

The model repository *EMF Store* [KH10] includes a three-way merging component that integrates two sequences of operations recorded in parallel development branches. Conflicts are detected upon the actually performed modifications, e.g., if the value of a single-valued structural feature has been changed contradictorily, the user may select one modification to be ignored. In addition, a small subset of context-sensitive conflicts are recognized; these include dangling references, but lack, e.g., cyclic containment conflicts.

In [Rut+09], an approach to operation-based three-way merging of models is defined on top of *category theory*. Similarly to BTMerge, a *merge model* serves as an intermediate structure on whose basis conflicts are detected and resolved. The initially defined conflict set considers elementary operations such as insertions, deletions, renamings, or moves. In addition to these, metamodel-specific conflicts may be defined based on category theoretical formalisms.

Semantical Merging. The three-way model merging approaches presented above differ with respect to their support for the detection of context-free and context-sensitive conflicts. Albeit, none of the tools and algorithms may guarantee that the merge result is correct with respect to the *semantics* expressed by the specific modeling language.

In [Fah+14], a three-way merging approach specific to class diagrams (represented in a small subset of the MOF language) is shown; it preserves semantical soundness in the sense that "all instances of the merge of a model and its differences with another model are automatically instances of the second model" [Fah+14, p. 1]. Rather than considering class diagrams as structured syntactical units, the effects of edit operations on their *instances* are taken into consideration. To this end, a *compositional algebra* that describes semantical features such as subtyping, cardinalities, or referential integrity, is defined. To date, no tool implementing this algorithm is known.

Furthermore, the model VCS *SMoVer* [Alt07] is able to detect semantic in addition to syntactic conflicts. To this end, so called *semantic views* are derived from the modifications detected between the base and the alternative versions. Each semantic view covers an individual aspect of the modeling language, e.g., type correctness in class diagrams.

6.2.3 Model Version Control Systems

To conclude the description of this integrating discipline, a subset of the currently available MVC systems are outlined with a focus on the supported model types, their architectural

design, and the functionality made available to users of the respective system.

Connected Data Objects. *Connected Data Objects*[5] (CDO) is an optional EMF component that serves as both a simplistic version control system for all types of EMF-based models and a persistence framework for models on top of which model management applications can be built. Developers or end users may access specific revisions of a model using a read-only view, or start write transactions explicitly by cloning the repository and writing the changes back in a commit-like operation later.

Access to the repository is organized by means of database-like transactions, thus pessimistic synchronization is applied for write transactions. CDO follows a client/server architecture; communication has been implemented based on the *Net4j* library[6], which relies on the *hypertext transfer protocol* (HTTP). For server-side persistence, different database technologies, including SQL as well as NoSQL-compliant systems, are available.

EMF Store. The aforementioned *EMFStore* [KH10] is a fully-fledged *operation-based* version control system for arbitrary EMF-based model instances. The integrated change recording mechanism requires a tight integration of model editors with the VCS. As another option, a default editor based on the *EMF Client Platform*[7] is provided.

Internally, the repository is organized by directed forward deltas. The revision graphs allows for branches and merges in order to support collaborative development. Client/server communication has been implemented with XML-based *remote procedure calls* (RPC); for server-side persistence, the NoSQL database *MongoDB*[8] is used.

Odyssey VCS. *Odyssey-VCS* [OMW08] is a version control system specific to the modeling language UML. In contrast to other VCS, this system is flexible with respect to the object granularity, e.g., UML class diagrams may be versioned with respect to their packages, classes, or details of classes. The version history allows for explicit branching and merging.

The system follows a classical client/server architecture. Rather than relying on directed deltas, different *snapshots* of the modified artifacts are transferred in XMI format using a dedicated *web service*.

CoObRA. *CoObRA* [SZN04] is a persistence framework upon which collaborative modeling environments can be built. It is integrated into the CASE tool *Fujaba* [NNZ00] and follows an operation-based paradigm. Changes performed in Fujaba are recorded and appended to a change log, which also serves as persistence format for models in the workspace. Thus, CoObRA relies on directed forward deltas.

Changes carried out in workspaces of different developers are orchestrated by means of a central server repository, which manages the version histories in a proprietary XML-based format. Concurrent modifications are synchronized in an optimistic way.

5 http://www.eclipse.org/cdo/
6 https://wiki.eclipse.org/Net4j
7 https://eclipse.org/ecp/
8 https://www.mongodb.com

Git Integration for MetaEdit+. In [Kel17], an integration of the modeling environment *MetaEdit+* with the distributed VCS Git has been presented. Concurrent modifications are avoided by instantaneously locking the affected model artifacts upon their first modification. This promises to significantly reduce the complexity to which the modeler is exposed, but the approach becomes problematic when used with large and frequently accessed model resources. To better comprehend changes, a graphical difference view is provided.

From Peer-to-Peer to Distributed Model Versioning. Apart from centralized model versioning, there exist several approaches to transfer model changes in a peer-to-peer fashion between different workspaces.

In *DPraxis* [MBG09], changes performed to a local copy of a distributed model are captured in the form of fundamental edit operations. Peer-to-peer communication is realized by a *publisher/subscriber* mechanism, which allows to selectively filter both the senders and the receivers of a propagated change. The framework also supports rudimentary conflict detection and resolution.

DiCoMEF [KE14] is a collaborative editing framework for models being instances of arbitrary domain-specific modeling languages based on Ecore. In contrast to DPraxis, DiCoMEF assumes one dedicated *master model* to orchestrate synchronizing operations. Rather than relying on a central remote repository, *change requests* and *change propagations* are sent and received by the e-mail protocol.

Comprehensibly, peer-to-peer versioning involves the danger of diverging copies. As shown in Section 4.5, *distributed version control* provides for a meaningful compromise between centralized and peer-to-peer versioning. Albeit, literature lacks real support for *distributed model versioning* as motivated, e.g., by [HP04].

6.3 Software Product Line Version Control

From a conceptual view, each software product line may be decomposed into three different types of artifacts: First, the *problem space*, represented by a variability model such as a feature model. Second, the *solution space*, i.e., the platform that is available as a superimposition (annotative variability) or as a core product augmented with a set of composition rules (compositional variability) or transformations (transformational variability). Third, the *mapping* between problem and solution space, implemented by presence conditions assigned to solution space elements or as application conditions for transformations, respectively. In order to support collaborative SPLE as well as the possibility of a systematic change comprehension, all these artifacts need to be put under version control as illustrated in

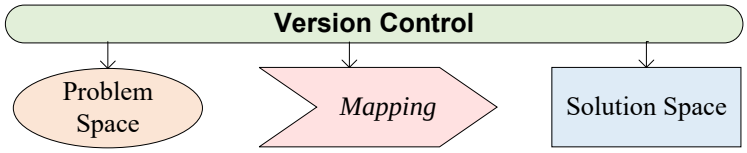


Figure 6.8: General problem statement for software product line version control.

Figure 6.8. For each historical version of the product line, it must be ensured that problem space, solution space, and the mapping in between are consistent with each other.

At first glance, nothing is wrong with applying a state-of-the-art version control system such as Subversion or Git to a software product line (or a model VCS, assuming that the SPL in question is developed in a model-driven way). As shown below, there are, however, good reasons to specifically address the question of *software product line evolution* [LC13], which is here considered as a generalized problem of *software product line version control* (SPLVC). When compared to the previously considered problem of model version control, SPLVC is a broader – due to the heterogeneity of SPL approaches – yet much less densely populated research field. Section 6.3.1 categorizes possible evolution problems to consider, before Section 6.3.2 presents building blocks upon which SPLVC may be realized. Section 6.3.3 gives a short overview of fully-fledged SPLVC systems.

6.3.1 Categorization of Evolution Problems

In [SHA12], different kinds of evolution in SPLE are categorized. When referring to Figure 6.8, the origin of an evolutionary change is either the problem space or the solution space. For instance, in case the problem space is represented by a feature model, viable operations are *duplicate feature*, *insert feature*, or *split feature*. Evolution steps performed in the solution space largely depend on the used implementation language and approach; e.g., in Java, refactorings such as *extract method* or *rename element* come into play.

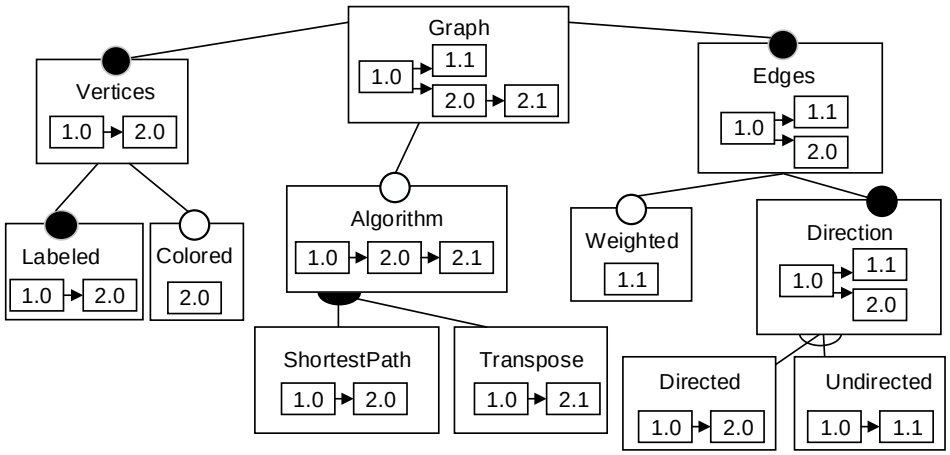
The evolution steps performed in the problem or solution space are seldom confined to the respective space, but more often, a propagation to the mapping or even into the opposite space is desired. Correspondingly, in [SHA12], the *semantical extent* of evolutionary changes is classified as follows:

Intraspatial Changes. These correspond to internal refactoring operations that do neither influence the mapping between problem or solution space, nor propagate to the opposite space. Examples are the introduction of a new feature without any mapping, or corrective changes in the implementation of an existing feature.

First-Degree Interspatial Changes. This category ensues either from the problem space or from the solution space and needs to be propagated to the mapping in order to keep the product line consistent. For instance, in case a feature is deleted, all presence conditions referring to it must be revised.

Second-Degree Interspatial Changes. In this case, an evolutionary change is propagated from the problem space over the mapping to the solution space, or vice versa. For instance, the operation *split feature* requires to replace all instances of the original feature in the presence conditions, and furthermore involves splitting the referencing artifacts in the solution space accordingly.

Taking this distinction into consideration, co-evolution scenarios in SPL may be, on the one hand, better understood; on the other hand, automated propagation can be provided by tools.



$ShortestPath[>1.0] \Rightarrow Weighted$ $ShortestPath[2.0] \Rightarrow Directed[>1.0]$

$ShortestPath[1.0] \Rightarrow Directed[1.0]$ $Transpose[1.0-2.1] \Rightarrow Directed[\geq 1.0]$

Figure 6.9: An example of a hyper feature model.

6.3.2 Partial Solutions Towards Product Line Version Control

Coming back to the more general topic of SPLVC, several solutions on top of which SPL-specific VCS may be built exist in the literature. A first group of approaches is limited to controlling the evolution of the problem space. Secondly, several compositional approaches that consider both the problem and the solution space have been described.

History-Aware Feature Models. The fact that a feature model's evolution needs to be described by a consistent version history has been addressed by different formalisms. For example, *hyper feature models* [SSA14a] organize an individual revision graph for each feature. In order to allow for the definition of valid combinations of versions of features, a *version-aware constraint language* is provided. During feature configuration, the twofold semantics of variability in time and in space is taken into account. Furthermore, an algorithm for automatic version selection is provided in [SSA14a]. This way, the categories *maintenance/evolution* and *software configuration management* of variability in time (see above) can be adequately addressed.

An example of a hyper feature model with reference to the running Graph example is depicted in Figure 6.9. Version-aware constraints ensure that algorithms operate on compatible versions of their data structures.

Fragmenting and Merging Feature Models. In [Dhu+08], a collaborative evolution approach to MDSPLE based on *fragments* is presented. Motivated by the large scale and life-span of product lines, it is proposed to divide up the feature model into several sub-trees. In analogy, the corresponding solution space fragments, here represented as domain model fragments, are divided up; references to elements contained in external fragments are established by means of a *placeholder* mechanism. As different fragments are versioned in

isolation, many collaboration problems are avoided.

Using this approach, product derivation first involves a composition of a stable and up-to-date version of the feature model. For this purpose, a semi-automatic merging strategy is employed, where several types of merge conflicts, e.g., name mismatches or multiple definitions of features, must be resolved. Next, after feature selection, the corresponding parts of the solution space are composed, while references to placeholder elements are re-targeted to their corresponding domain model elements located in other fragments.

Similar techniques for fragmenting and merging feature models are presented in [Ach+11], where multiple SPLs share (different sub-trees of) the same feature model.

Delta-Oriented Evolution Approaches. Particularly when applying transformational variability, dedicated evolution support is required in order to ensure consistent version selection as well as a correct and meaningful product composition strategy.

In [SSA14b], an integrated approach towards variability in space and in time, based on hyper feature models and delta-oriented composition languages is described. The connection between the multi-version problem space and the solution space is established by means of explicit deltas. The approach distinguishes between two types thereof. *Configuration deltas* are conditional to the existence of specific features, i.e., they describe their implementation in the solution space. On the contrary, *evolution deltas* describe the product-level change recorded between two subsequent revisions of a feature. In order to define the operations applicable in these deltas, the delta language family *DeltaEcore*[9] is provided (cf. Section 6.1.2).

Another delta-oriented approach to SPLVC is presented in [Hab+12]. It is focused on architectural models of SPLs as created during the *domain design* phase of the traditional SPLE process. Baselines of architectural descriptions may be modeled using a dedicated *architectural description language* (ADL). For explicit deltas modifying ADL model instances, another *delta language*, which can be used to define evolutionary as well as variation-related changes, is introduced. Both the evolution and the variability of the product line are captured in a uniform representation combining revision graphs and feature model concepts. Special emphasis is put on *refactoring* operations that aim at maintaining the consistent configurability of the SPL. To this end, *application conditions* are attached to deltas rather than to the variability model, which allows a more fine-grained control over product composition, but also increases the complexity of the overall approach.

6.3.3 Software Product Line Version Control Systems

To conclude this section, several tools that offer holistic SPLVC support, yet rely on completely different implementation techniques, are contrasted. Existing SPLVC systems are based on contemporary version control solutions or have been developed from scratch.

Feature-Driven Versioning. In [ME08], a *feature-driven versioning* framework building upon *Subversion* is described. It transparently maps elements of feature models, domain models, and traceability links between those, to versioned objects. New variants are created

9 http://deltaecore.org/

locally by merging branches, assuming that each branch is responsible for a specific feature. Here, in so called *feature/product containers*, features may be implemented in isolation; however, this also coarsens the granularity of feature implementation to source code files. Although product derivation is transparently delegated to merging of branches organized by Subversion, it remains unclear how product-specific changes, performed in a merged branch, are propagated back to the repository in order to affect a larger set of variants.

DE/AE Propagation. In [MD15], an extension to the VCS Git enabling SPLE management is presented. It allows to propagate product-specific changes, performed in a merged branch, back to the trunk, such that domain and application engineering are bidirectionally integrated. Rather than being integrated into the underlying version control system Git, the extension is built upon the hosting platform *GitHub*, resulting in a comparably loosely-coupled tool integration.

VCS-Supported Clone-and-Own. [Pfo+16] present a *clone-and-own* support tool particularly designed for transitioning from loose variant management to SPLE. At the heart of the tool is a feature-to-code tagging editor that allows to define specific source code regions as varying. Tags comprise arbitrary boolean expressions on features. Once migrated to a product line, it is possible to merge specific source code fragments into the local workspace under the same context, such that the feature tags are extended and managed automatically. This heuristic strategy, however, does not obviate further manual reviews of feature tags.

Selecta. The SPLVC system *Selecta* [EDL10] is built around a database-centric *computer aided domain-specific environment*, which covers not only version control but also more general aspects of software configuration management. This way, in addition to the evolution of the variability model and the product space, evolving engineering environments and the economical scope of the product line are controlled. Products themselves are derived in a compositional way based on *configuration specifications*, which may also be partial, such that the resolution of specific variation points can be delayed. Moreover, version constraints can be phrased in a dedicated *constraint language*. Rather than relying on feature models, variability is expressed in terms of *family groups*, which constitute a nested representation of variation points and variants.

Component-Based SPL Versioning. Thao [Tha12b] presents a configuration management solution for SPLs, which intends to address evolution problems in an explicit tool-supported way rather than relying on low-level techniques such as preprocessors. A product line versioning layer is built on top of a generic low-level versioning layer and on a component/project versioning layer (the *Molhado* system [Ngu+05]). The framework applies a component-based style of versioning, where versioned XML documents are used for the description of components. This way, components may be shared between domain and application engineering, which enables the propagation from platform artifacts to products and vice versa. Albeit, this propagation happens in a semi-automatic way. Particularly when integrating product-specific changes into the platform, manual variability management becomes necessary.

6.4 Integrated Historical and Logical Versioning

The combination of historical version management as provided by VC and variant management as provided by SPLE may be seen from a second perspective. Rather than considering the product line as an entity to be versioned with respect to historical evolution, the platform may be regarded as a piece of software to be versioned along the historical and the logical dimension equally, thus applying *integrated historical and logical versioning* (IHLV) to a software basis. This addresses SCM in the broader sense; initial ideas were developed far in advance of the advent of SPLE.

To begin with, a discussion about the commonalities and differences of VCS and SPLE is provided in Section 6.4.1. In the remainder, approaches that explicitly support both version dimensions are portrayed. Rather than taking the specific requirements for software product lines into consideration, the scope is widened to the general SCM problem of *configuration options*. The solutions listed differ with respect to how the relationship between historical and logical versioning is seen, giving rise to a distinction between *asymmetric* (see Section 6.4.2), *orthogonal* (see Section 6.4.3), and *hybrid* (see Section 6.4.4) approaches.

6.4.1 VC and SPLE: Are they so Different?

From a comparison of the disciplines version control and SPLE, a list of commonalities, similarities, and differences are deduced in the following[10]. Together with the theoretical foundations to be discussed in Chapter 8, this list has been taken into account for the design of the conceptual framework presented in Part IV. Throughout the discussion, we assume "traditional" representatives, e.g., a contemporary text-oriented VCS such as Subversion and one of the SPL implementation tools presented in Section 5.4. Table 6.1 summarizes the explanations given in the following.

Commonalities. Both VC and SPLE provide a notion of the entirety of products. In VC, this is a *repository*, whereas in SPLE, this corresponds to the *platform*. For single product versions, the terms *workspace* and *product (configuration)* are used.

Moreover, in both disciplines, there exist two distinct representations for the connection between version and product space. On the one hand, it is possible to store all variants as a *superimposition*, which corresponds to *symmetric deltas* in VCS and to *annotative variability* in SPLE. On the other hand, only a minimal core may be defined, which is then manipulated by specific operations. This is realized by *directed deltas* in VC and by *transformational variability* in SPLE. For compositional variability, no VCS-related counterpart exists.

In either case, it is necessary to assign *visibilities* to program fragments or to transformations. These correspond to *version identifiers* – sets or ranges of revisions – and to *presence conditions* – propositional logical expressions on the available features.

10 Parts of this section have been pre-published in [SBW15, Section 3].

Table 6.1: Differences, similarities, and commonalities among VC and SPLE. Based on [SBW15, Table 2].

	Generalization	Version Control	SPLE
Commonalities	all product versions	repository	platform
	single product version	workspace	product configuration
	superimposition	symmetric deltas	annotative variability
	transformations	directed deltas	transformational var.
	visibilities	version identifiers	presence conditions
Similarities	version model	revision graph	feature model
	version	revision	feature configuration
	filter	check-out	product derivation
Differences	variability kind	variability in time	variability in space
	variation points	spontaneous, hidden	planned, explicit
	product variability	unconstrained	constrained
	version specification	extensional	intensional
	version membership	immutable	mutable
	editing model	filtered	unfiltered
	visibility management	automatic (commit)	manual (edit)

Similarities. In both disciplines, there is an abstraction for the entirety of available versions. In VC, *revision graphs* describe the commit history. In SPLE, *feature models* organize mandatory and/or optional features of a product line within a tree. The specification of a single *version* is done by selection of a *revision*, or by a *feature configuration*, which in turn describes a product variant.

Both VC and SPLE provide a *filter* operation. In VC, it populates the workspace after a revision has been selected for *check-out*. In SPLE, filtering is applied as *product derivation* during application engineering.

Differences. Both disciplines deal with different kinds of *variability*. Version control manages *variability in time*, i.e., the fact that a software project is subject to evolution. SPLE, in contrast, deals with *variability in space*, using variability models to describe commonalities and differences among related variants explicitly. In SPLE, it is intended that several configurations of a software project co-exist. This kind of variability is typically *planned* in advance by suitable *variation points* in the platform; in contrast, VC assume that variation points are *hidden* and that they are created *spontaneously*. Most SPLE tools require the platform to be free of context-free or context-sensitive conflicts, e.g., a syntactically correct program that is accepted by the respective compiler, or a valid instance of the metamodel in the case of MDSPLE (*constrained* variability). In VC, there are no restrictions concerning product variability—neither a superimposition nor directed deltas need to be syntactically well-formed; constraints are merely imposed to single-version products (*unconstrained* variability).

VC and SPLE also differ in terms of version specification. Typically, VCS make a fixed

set of versions available for selection (*extensional versioning*). In SPLE, versions may be described by a combination of features, allowing to create variants that have not been committed earlier (*intensional versioning*). VCS guarantee the *immutability* of version membership of an element. Once committed, it is not possible to remove a product element from a revision. In contrast, it is allowed to modify the visibility of an element in SPLE.

In VC, *filtered editing* is applied. After check-out, the developer sees and may modify elements belonging to the selected revision only. As soon as a commit is issued, changes are detected in the local workspace and written back to the repository, while visibilities are updated *automatically*. In contrast, SPLE typically requires the user to edit a multi-version view (*unfiltered editing*) and to manage visibilities (i.e., presence conditions) *manually*.

VCS and SPLE tools both offer operations that are not realized by the opposite. The VCS operation *commit* detects differences in the workspace in order to write changes back to the repository automatically. No equivalent operation, which would allow to propagate product-specific modifications back to the platform, exists in SPLE tools. Conversely, it is possible to directly modify the *mapping* between the variability model and the platform, i.e., the visibilities. There exists no VCS that intentionally allows to manually edit version membership (which would, indeed, destroy the property of immutable revisions).

Conclusions. VC and SPLE share a – maybe unexpectedly – large amount of similarities, particularly with respect to underlying data structures. Most differences are behavioral and due to the underlying editing models. It is worth investigating the benefit of transferring some differences to the respective opposite domain. This requires a stable foundation for *integrated historical and logical versioning*. Subsequently, we investigate and assess three candidate architectures that potentially provide this foundation.

6.4.2 Asymmetric Integrated Versioning

Under *asymmetric*, we here subsume approaches that explicitly combine historical and logical versioning, yet at different conceptual levels, i.e., the relationship between the variability model and the product space has another quality as the relationship between the *evolution model* (e.g., revision graph) and the product space. From the perspective of the evolution model, the platform and the variability model, in turn, are homologous. A generalized architecture for asymmetric approaches is sketched in Figure 6.10. Asymmetric IHLV can be achieved by putting an SPL managed with a state-of-the-art SPLE tool under VC.

An example of a system intentionally following this approach is *Adele* [EC94]. A multi-variant platform is managed by a variability-aware object-oriented schema definition

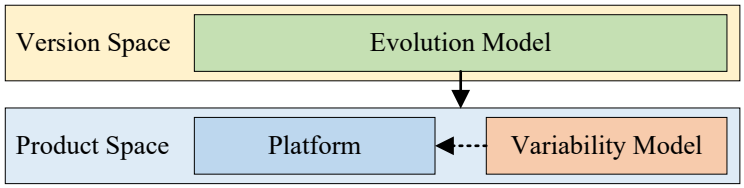


Figure 6.10: Asymmetric architecture for integrated versioning.

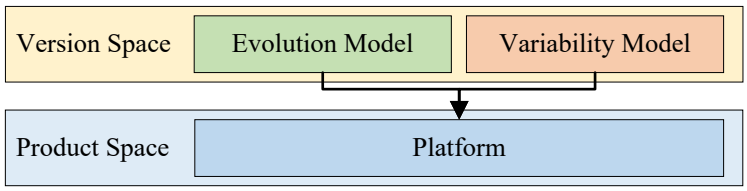


Figure 6.11: Orthogonal architecture for integrated versioning.

language. Here, variants can be explicitly defined, following the paradigm of *annotative variability* when speaking in SPLE terms, or *symmetric deltas* in the version control nomenclature. A historical versioning layer has been added on top, which in turn relies on *directed deltas*. Specific parts of schema definitions may be defined as *modifiable*, such that multiple historical versions of it are allowed to exist. In contrast to logical variants, historical revisions are managed transparently in a repository.

When compared to naive approaches to SPL/VC integration, the added value seen from the user's perspective is low. An additional problem is the conceptual handling of the variability model, which primarily serves for version selection, but is also subject to evolution itself.

6.4.3 Orthogonal Integrated Versioning

When compared to asymmetric approaches, *orthogonal integrated versioning* lifts the variability model up to the version space, making it homologous with the evolution model (e.g., revision graph). As a consequence, elements of the product space are versioned with respect to both the historical and logical version dimension (cf. Figure 6.11).

The orthogonal architecture has been realized, among others, in the tool *VOODOO* [Rei95]. A so called *version cube* is provided as a conceptual abstraction for version identification in the historical dimension (a sequential revision graph), the logical dimension (variation points – here referred to as *version group nodes* – and *variants*), and the product dimension (*components* that may be selected individually). Physically, the objects part of the version cube are stored in an *object pool*. The selection of a specific object involves a choice in the revision graph and a subsequent selection in the intertwined variant/product space. For managing the variant space, specific pre-defined operations are offered.

Although providing homologous IHLV, orthogonal approaches suffer from a conceptual disadvantage when compared to asymmetric (see above) and hybrid approaches (see below): the evolution of the variability model is not adequately addressed. A concrete example is provided in Section 7.1.7.

6.4.4 Hybrid Integrated Versioning

Both approaches to IHLV discussed above imply problems which are due to the respective role of the variability model. On the one hand, asymmetric approaches do not consider that it resides at the same conceptual level as the evolution model, such that revision selection and variant configuration are covered by mutually independent mechanisms. Consequently, the added value when compared to naive VC/SPLE integration is low. On the other hand, by

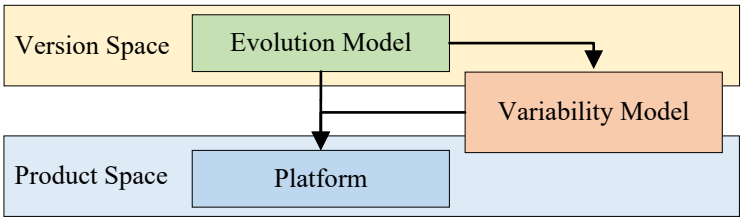


Figure 6.12: Hybrid architecture for integrated versioning.

considering it as part of the the version space, orthogonal approaches neglect to control the historical evolution of the variability model.

Attempting to "get the best of both worlds" motivates *hybrid* approaches to integrated versioning (cf. Figure 6.12), where the variability model plays a dual role, being part of both the version space and the product space. Thus, the evolution model applies historical versioning to both the variability model and the platform, whereas the platform is versioned by both the evolution model and the variability model uniformly.

To date, there exists no tool that strictly and fully implements the architecture proposed in Figure 6.12. Supplementarily, two formal approaches based on which a hybrid solution can be built are presented subsequently. Both of them provide a generic base mechanism that does not assume specific forms of representation for the evolution model, the variability model, and the product space, respectively.

Unified Versioning Based on Feature Logic. In [ZS97], Zeller et al. introduce the *ICE* system, which is based on a version set model that relies internally on *feature logic*. *Feature terms* essentially correspond to nested $[key : value]$ assignments, where a value can be either atomic or another feature term. When transferring this concept to software configuration management, different properties of software systems – belonging to the historical or variant dimensions, as well as other SCM aspects such as building or testing – can be expressed by feature logical terms [ZS97]. Versions are unambiguously defined by *unification* of different feature terms, which resolves the variability present in the versioned software. By unification, on the one hand, partial configurations, each expressed by a feature logical expression, can be combined. On the other hand, illegal combinations of version configurations can be detected.

Feature logic naturally supports *intensional versioning* (cf. Section 4.6). In addition, *extensional versioning* is needed in order to map selections in revision graphs to feature logical expressions. To this end, [ZS97] propose a *change-oriented* mapping; a change feature Δ_i corresponds to an edge in the revision graph and is labeled according to its target revision i.

Uniform Version Model. When compared to the aforementioned ICE approach, the *Uniform Version Model* (UVM) [WMC01] resides at an even higher level of abstraction by relying on *set theory* and *propositional logic*. UVM defines a couple of basic concepts based upon which an integration of historical and logical versioning – realized by a combination of intensional and extensional version control – can be achieved. The connection between

the product space and the version space is established by *visibilities*—propositional logical expressions on options that control in which versions a product space element is present, generalizing the concepts of *presence conditions* and *version identifiers*.

UVM includes a *filtered editing model* that is based on the notions of *choice* and *ambition* that have their origins in *multi-version editors* [Kru84; SBK88] and *change-oriented versioning* (CoV) [Mun+93]. As sketched in Section 1.4.3, the developer edits the product space transparently in a *view*, where visibilities are hidden. The changes are connected to a subset of the version space, defined by the *ambition*, and applied representatively in a version described by the *choice*.

6.5 Summary and Outlook

In this chapter, we have summed up the state of the art concerning the mutual combinations of MDSE, SPLE, and version control.

Just like conventional SPLE, model-driven software product line engineering may be applied based upon three distinct paradigms. In the case of *annotative variability*, the platform consists of a multi-variant domain model; product derivation is realized by removing those model elements referring to features not included in the selected variant. *Compositional variability*, in contrast, is based on the idea of adding specific fragments to a minimal *core model*. Several techniques exist for expressing composition semantics. For realizing *transformational variability*, specific variability-aware approaches are preferred over general purpose model transformation languages.

Model version control and software product line version control aim at improving the support for *evolution* and *collaboration* of MDSE and SPLE, respectively. Both integrating disciplines are motivated by the inadequately coarse object granularity of state-of-the-art VCS, which typically interpret versioned items as plain text files, whose contents are defined in terms of text lines. MVC and SPLVC systems promise to ease comparison and merging, but also to improve the consistency of the product space. Albeit, such systems are underrepresented in literature, especially with regard to fully *automated* and *distributed* solutions.

Many VCS support a restricted form of variant management by offering the concept of *branches*, where different offsprings of the main development line may be organized in an *extensional* way. *Integrated historical and logical versioning* may be performed on the basis of different architectures; we have come to the conclusion that the *hybrid* architecture combines the benefits of *asymmetric* and *orthogonal* integration.

The concepts and mechanisms underlying UVM are revisited in a more formal way in Section 8.4. Furthermore, in [WC09], preliminary considerations on transferring UVM to SPLE have been made; the contributions presented in Part IV build upon this. In particular, Chapters 9 and 10 explain how hybrid IHLV has been realized by representing the feature model as a part of both the version space and the product space. This way, a well-established abstraction is provided for (intensional) logical versioning, while the management of the (extensional) historical dimension is fully automated.

After more than [40] years
of research and practice
in software configuration management,
constructing consistent configurations
of versioned software products
still remains a challenge.

REIDAR CONRADI AND
BERNHARD WESTFECHTEL (1998)

Chapter 7

Design Choices and Decisions

Abstract

This chapter ties on the previous chapter's literature review and transitions to the conceptual contributions of this thesis. Using concrete examples where adequate, eight obstacles that hamper the application of a combination of MDSE, SPLE, and VC are identified and explained. Moreover, based upon the internal properties of the surveyed tools, we identify twelve design choices for a tool to integrate MDSE, SPLE, and version control. The choices are resolved by corresponding design decisions, which have been considered both in the conceptual framework and in the tool SuperMod. The choices are justified with the requirements identified in advance and concretized with the conceptual obstacles in mind. Seven additional design principles complete the description.

Contents

7.1 Obstacles to the Application of MDSE+SPLE+VC

The literature review provided in the previous chapter has considered different approaches to MDSPLE, MVC, and SPLVC. In this chapter, the information is further processed and constructively assessed. The current Section 7.1 lists and exemplifies eight typical obstacles state-of-the-art tools or combinations thereof are faced with. The challenges connected to these obstacles have been neglected or only partly addressed by approaches described in the literature.

On top of the identified obstacles, Section 7.2 elaborates *design choices* towards an integrated conceptual framework, before concrete *design decisions* are made in Section 7.3.

7.1.1 Plan-Driven SPL Approaches Hamper Agility

As hinted in the introduction, agile software development has become more and more important over the recent years. It still depends on the specific requirements of projects whether agile or plan-driven development processes are to prefer. Without contributing to this discussion, one can argue that development support tools should at least be compatible with agile software development in a sense that they do not interfere with agile principles.

In Section 5.3, it was stated that the majority of established SPLE processes assume a *plan-driven* and *proactive* paradigm, which is also reflected by the landscape of available SPLE tools. Several properties of proactive SPLE on the one hand and agile development on the other hand are mutually exclusive.

To get more precise, let us recite an excerpt of the twelve principles of the *agile manifesto*:

> [...] Our highest priority is to satisfy the customer
> through early and continuous delivery of valuable software.
>
> [...] Welcome changing requirements, even late in development.
> Agile processes harness change for the customer's competitive advantage.
>
> [...] Deliver working software frequently, from a couple of weeks
> to a couple of months, with a preference to the shorter timescale. [...]
> ([Bec+01])

These agile principles clash with the properties of proactive SPLE for the following reasons:

- Plan-driven processes impede the early delivery of software through the strict separation of domain engineering (DE) and application engineering (AE). Before the first product can be deployed to its customer(s), one complete DE and one AE run are necessary. The DE run includes the design and implementation of *all* features identified during domain analysis.
- The high amount of planning involved by proactive SPLE complicates the handling of new or changing requirements, particularly when connected to the maintenance phase after several products have been derived. In general, (proactive) SPLE assumes that the individual requirements for all products are defined in advance and that they remain stable [PBL05].
- Application engineering is a "one-way road": Once a product has been derived, the connection to the platform gets lost. Therefore, maintenance activities must be synchronized

between DE and AE. The additional maintenance time hampers the quick and frequent re-generation of products.

Taken together, *agile SPLE* is impeded by the absence of a suitable process but also by a lack of tools following the *reactive* SPL adoption path, which "aligns well with agile methods of software construction" [Ap+13b, p. 42]. Such tools should give up the strict separation between DE and AE in favor of early adoption, quick reactions to customer feedback, and frequent delivery.

7.1.2 Variability in Time and in Space may Overlap

Traditionally, version control and SPLE have been considered as two independent disciplines. The explanations given in Sections 6.3 and 6.4 have shown that the combination of both disciplines, resulting in an integration of historical and logical versioning, has been researched to a sparse extent. Here, we motivate by an example that demonstrates that variability in time and in space may overlap, leading to context switches and to repeated version/variant management overhead when following the separate-tools approach.

This obstacle is exemplified by conducting a cut-out of the *Graph* example using an off-the-shelf tool chain: A state-of-the-art VCS supports the development of the platform in multiple iterations. First, a feature model is defined; next, an MDSPLE tool relying on annotative variability based on explicit mapping (cf. Section 6.1.1) is used to annotate domain model elements with variability information. [1]

The platform is defined in the form of a *multi-variant domain model* (MVDM), whose elements are committed to the VCS "feature by feature". In Table 7.1, the version history is listed. Figure 7.1 shows the final version of the MVDM in class diagram notation. [2]

After having defined the variability model and the platform, they need to be connected. A *mapping model* for the example product line, realized with the help of the tool *FAMILE*

Table 7.1: History of the multi-variant domain model for the *Graph* product line. Based on [SBW15, Table 1].

Rev.	Inserted Elements	Commit Message
1	(Feature model)	"Initial commit."
2	Class Graph	"Realization of root feature Graph."
3	Class Vertex, association has vertices	"Realization of feature Vertices."
4	Class Edge, association has edges	"Realization of feature Edges."
5	Property Edge::label	"Realization of feature labeled."
6	Property Edge::weight	"Realization of feature weighted."
7	Association connects	"Realization of feature undirected."
8	Association starts at and ends at	"Realization of feature directed."
9	Class Color, ass. has color	"Realization of feature colored."
10	(Mapping model)	"Established mapping."

1 This subsection is based on a motivating example pre-published in [SBW15, Section 2].

2 Keep in mind that this is a synthetic example; the granularity of commits is artificially fine. Furthermore, the development of the mapping model might also happen in parallel with the domain model.

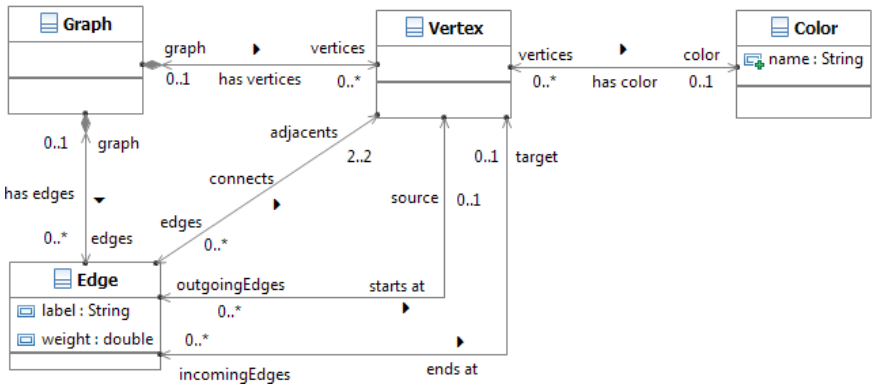


Figure 7.1: Multi-variant domain model, which realizes all features of the *Graph* product line, as a superimposed UML class diagram. From [SBW15, Figure 4].

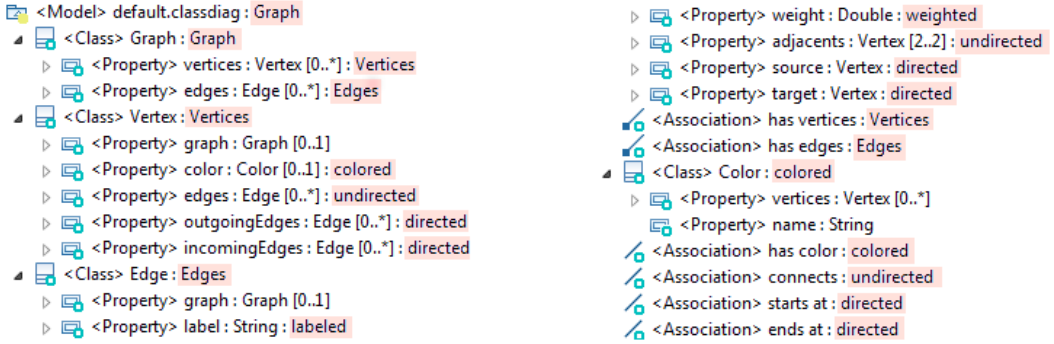


Figure 7.2: Mapping between the multi-variant domain model and the feature model. Based on [SBW15, Figure 5].

[BS12b], is shown in Figure 7.2. A total of 22 feature annotations are necessary. Last, the mapping is committed to the repository as revision 10.

Although having successfully applied an off-the-shelf combination of VC and MDSPLE, several obstacles can be identified. On the one hand, variability in time and variability in space are managed by means of two different mechanisms, which leads to undesired context switches between tools belonging to MDSPLE and to VC, respectively. On the other hand, the user has to repeatedly specify identical version information (i.e., feature expressions) for the same change. All manually provided mapping information (see Figure 7.2) could have been inferred, however, based on an analysis of the commit messages. Differently speaking, the overlap between variability in time and in space could have been exploited.

7.1.3 Annotative MDSPLE Means Constrained Variability

In a code-oriented SPL, annotative variability can be achieved using preprocessor approaches, where code fragments belonging to the multi-variant source code are mixed with conditional compilation instructions that control how the variability is resolved (cf. Sections 5.4.2). When removing the conditional instructions, such multi-variant source code is not necessarily a syntactically correct program ready to be processed by the compiler. Hence, preprocessors

```
1 public class
2 #ifdef LABELED
3    LabeledEdge
4 #else
5    Edge
6 #endif
7    { ... }
```

Listing 7.1: Unconstrained variability enabled by a hypothetical Java preprocesor.

allow for *unconstrained variability*. For example, in the class declaration in Listing 7.1, two mutually exclusive alternative names for the same class are defined in a superimposed way.

In model-driven SPLE, the limitations implied by annotative variability are comparably severe (see Section 6.1.2). In order to be editable, models need to be syntactically correct, i.e., conform to their metamodel. The UML metamodel, for instance, mandatorily defines an upper bound of 1 for the attribute Class::name; therefore, the example introduced in Listing 7.1 cannot be transferred to a multi-variant UML class model.

Taken together, when transitioning from code-oriented to model-driven SPLE, we are faced with the obstacle of *constrained variability*.

7.1.4 Compositional/Transformational MDSPLE Requires a Tool Mix

Tying on the preceding subsection, other potential solutions to remove the obstacle of constrained variability in MDSPLE are *compositional* (cf. Sections 5.4.3) and *transformational variability* (5.4.4), where variation points are described by means of, e.g., composition languages or (model) transformation specifications. These in turn complicate MDSPLE by introducing new concepts and tools which, on the one hand, have to be apprehended and understood by SPL engineers, and on the other hand, require to manage artifacts of different heterogeneous types (e.g., diagram-like models and text-like transformations/deltas, respectively).

To illustrate this obstacle, the language *DeltaEcore* [SSA14b] (cf. Section 6.1.2) is used here. Returning to the above example, one might model the base variant (unlabeled graph) as an ordinary UML class with name Edge. In order to describe a conditional renaming to DirectedEdge in case feature Directed is selected, three steps are required:

1. Define a domain-specific *delta dialect* including the operation `renameClass`.
2. Use the delta dialect for the definition of the *delta* that applies the defined operation to the class Edge to the base version of the model. This is shown in Listing 7.2.
3. Connect the delta to the feature Directed by a corresponding *delta module mapping* in the hyper feature model.

The example demonstrates that the definition of the delta does not only require additional model management effort (step 3) but also to deviate from the preferred modeling language (UML class diagrams) in order to make models variability-aware (step 2). Furthermore, conceptual additions to the modeling language may be required (step 1, which can be reused

```
delta "DirectedEdge"
  dialect <http://example/UML/rename>
  requires <../model/base/Graph.ecore>
{
    renameClass(<Edge>, "DirectedEdge");
}
```

Listing 7.2: Definition of a delta for renaming a UML class using *DeltaEcore*.

for multiple deltas instantiating similar operations). Assuming that developers want to stick to their familiar formalisms, approaches based on compositional or transformational variability may put a considerable obstacle in their way by new languages and tools that must be learned and applied.

Notice that similar obstacles are also implied by approaches relying on annotative variability. For instance, the formalisms of mapping models or templates (cf. Section 6.1.1) require context switches, yet to a smaller extent.

7.1.5 Multi-Variant Editing Increases Cognitive Complexity

The presence of variability demands additional *cognitive capacities* from the user having to make multi-variant design decisions. Provided that even in single-system development, architectural modeling is difficult due to the sheer size of diagrams, variability annotations scattered composition rules potentially increase confusion.

An example of a quite complex multi-variant design is given in Figure 7.3. The UML package diagram describes the architecture of MOD2-SCM, a model-driven software product line of software configuration management systems which in total consists of 147 features. The underlying architecture was created with the results of a preceding feature-oriented domain analysis in mind.

Let us assume that after a certain period of time, the scope of the product line shall be extended by a new feature whose realization is supposed to interact with few existing features. First, the feature is added to the feature model. Second, in order to incorporate this change into the package diagram, it is first necessary to understand the existing design as well as those parts of the architecture that refer to features that interact with the new feature. Third, having realized the change by corresponding modifications of the package diagram, it needs to be connected to the newly introduced feature by assigning corresponding presence conditions and by modifying the presence conditions of interacting domain model elements.

While performing such a change, a large part of the existing architecture is irrelevant, since it refers to features that shall not be addressed by the considered change request. Nevertheless, these elements and their assigned presence conditions may considerably distract the user. A potential solution to this obstacle would be temporarily fading out the irrelevant model elements. This, however, implies new challenges, e.g., the irrelevant elements must be identified based upon a description of the scope of the change.

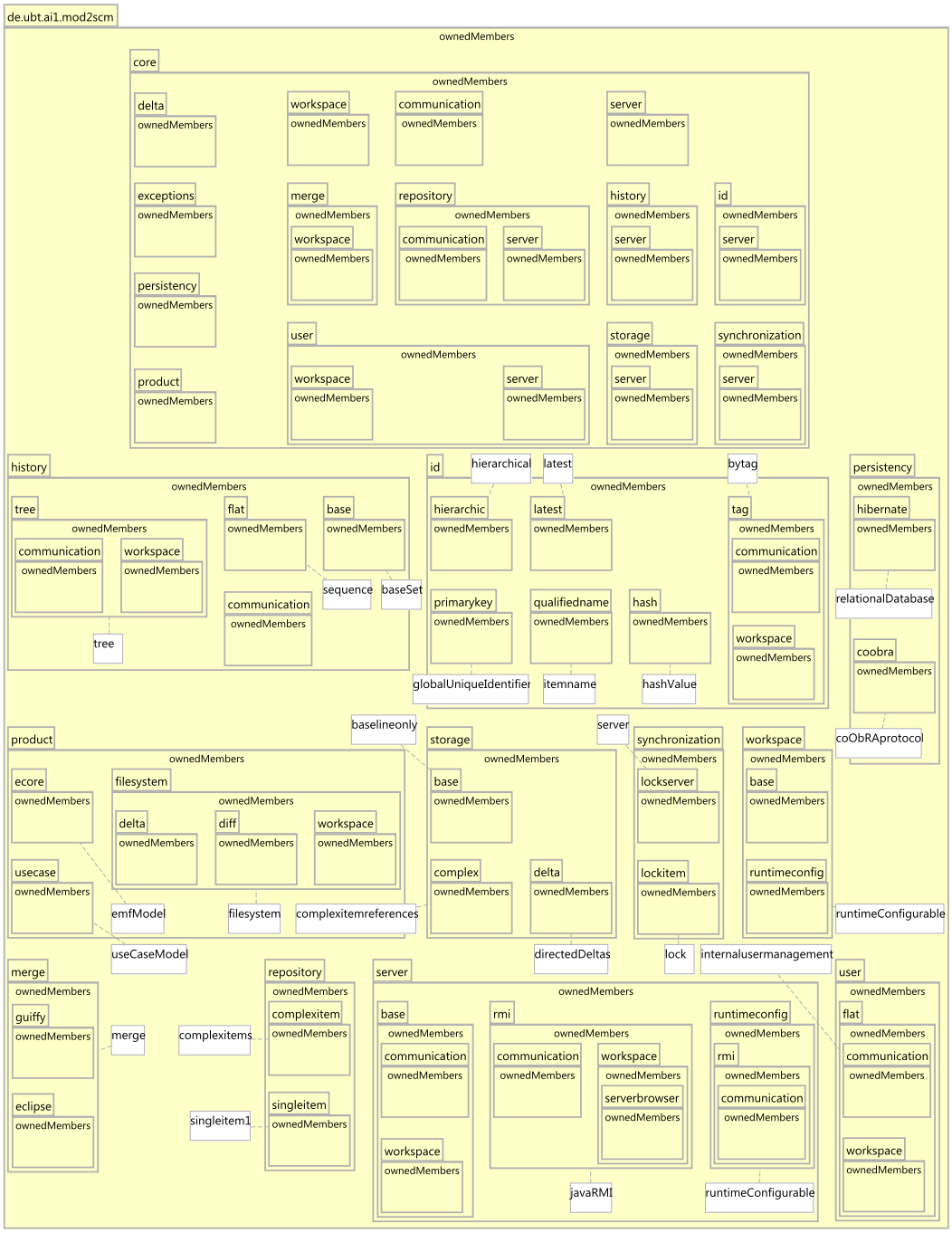


Figure 7.3: Multi-variant architecture of MOD2-SCM as UML package diagram. Presence conditions are represented as UML comments with a white background color. From [BDW12, Figure 10].

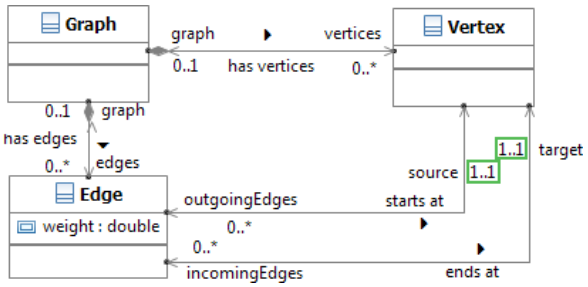


Figure 7.4: Corrective change retrospectively applied to a derived product. Affected elements are marked with a green box. Based on [SBW15, Figure 7].

7.1.6 Corrective Changes Cause Duplicate Maintenance

Let us come back to the adaptation of the *Graph* product line presented in Section 7.1.2, assuming that several products conforming to different feature configurations have already been derived. It turns out that the user of a derived product, a weighted, directed, and uncolored graph, suffers from a problem related to dangling edges that have no source or target vertex. Obviously, this problem is due to the too liberally chosen multiplicities [0..1] of the references source and target.

Apparently, the bug is not confined to the specific product variant where it has been reported in, but it affects related variants that include the feature *Directed*. Therefore, a corrective change must be applied to the total product line. A local bugfix performed in the affected variant is shown in Figure 7.4. When following a traditional distinction between domain and application engineering, there are two possible modes of procedure for making the bugfix effective both locally and globally:

– Apply the bugfix in the multi-variant domain model, i.e., as a part of the activity *domain engineering*. Then, the product must be derived anew, and all custom modifications made during previous *application engineering* activities get lost.

– Apply the bugfix twice: First in the affected product variant and then in the multi-variant domain. Then, check whether other product variants are affected and propagate the bugfix accordingly.

Both modes of procedure require *duplicate maintenance*, such that either the same bugfix must be applied at least twice, or product-specific adaptations must be repeated. This problem is due to the fact that after a product has been derived, its connection to the platform of the product line gets lost.

When referring to this concrete example, a desirable solution to remove this obstacle would be a possibility to fix the bug directly in the product, before propagating it back to the platform in a tool-supported and preferably automated way.

7.1.7 Orthogonal Versioning Fosters Destructive Updates

In Section 6.4.3, the concept of *orthogonal* versioning has been introduced as a potential solution to integrated historical and logical versioning. As hinted in Figure 2.1, specific

versions of objects may be considered as one point in a three-dimensional cube spanned by the object identifiers, revisions, and variants defined. Therefore, we may consider the identification of an object version as a mathematical function with three parameters:

$$obj : OID \times REV \times VAR \mapsto OBJ \tag{7.1}$$

where OID refers to the set of object identifiers, REV to the set of available revisions, and VAR to the (intensionally or extensionally defined) set of product variants. Since an object version does not exist for every possible combination of object identifier, revision, and variant, the function is partial.

As a main problem of orthogonal versioning, it was stated that the historical evolution of the variability model is ignored. When versioning an SPL, however, the evolution of the variability model should be allowed. By introducing or deleting, e.g., features in a feature model, the elements of the variant space become time-dependent. To this end, the variant space VAR must be defined as a partial function with at least two parameters, the set of variant identifiers $VARID$ and the revision set REV, the latter of which also controls the evolution of the product space.

$$var : VARID \times REV \mapsto VAR \tag{7.2}$$

Not representing the variant space as a time-dependent artifact can foster *destructive updates*, which may refer to the variant space or to the mapping between product and variant space. When considering the mapping in terms of *presence conditions* attached to fine-grained items of objects, orthogonal versioning allows for only one revision of a presence condition per item.

Such destructive updates may also lead to an inconsistent mapping between problem and solution space. For example, a fine-grained element e carries a presence condition f_1 in revision r_1. In revision r_2, a new feature f_2 is introduced to the variability model, and the visibility of e is updated to $f_1 \wedge f_2$. When restoring the old state r_1, the visibility refers to a non-existing feature.

Notice that such problems do not only appear when using an explicitly orthogonal integrated historical/logical versioning solution, but also when decoupling revision and variant control in off-the-shelf approaches, e.g., when putting only the platform but not the variability model (and/or the mapping in between) under revision control.

A remedy to destructive updates was presented in Section 6.4.4. In *hybrid IHLV*, the variability model is treated as a first-class artifact with respect to the historical dimension, i.e., its evolution is controlled in the same way as the platform's.

7.1.8 Collaborative MDSPLE is Limited by a Lack of Tool Support

The list of obstacles and challenges is concluded with a rather technical shortcoming that can be identified based on the literature reviews given in Chapter 6. For short, there exists no tool that satisfies the specific requirements of *collaborative* (model-driven) software product line engineering. These include:

– *Distributed* development of the product line – including the platform and the variability

model – by orchestrating different developers that should be allowed to perform modifications in isolation, before they are combined and integrated into the platform.

– *Three-way merging* specific to product lines is mandatory when expecting that optimistic versioning should be supported. To date, no three-way merging algorithm or tool exists that explicitly considers multi-variant models.
– Different *collaboration partitioning* strategies should be feasible for coordinating developers. The area of responsibility of different developers might be defined by product components, by features, by combinations thereof, or in another arbitrarily shaped form.
– *Low data traffic* should be caused by synchronizing changes in order to reduce both run-time and memory consumption.

Such a collaborative (MD)SPL version control system has not been developed to date. This obstacle is mostly considered as a technical one, but it also adds several new conceptual challenges connected to requirement **R18** (*collaborative SPLE*) to the list.

7.2 Design Choices

After having surveyed the obstacles that existing approaches towards combined MDSE/SPLE/VC are faced with, we now transition to the exploration of the design choices of the conceptual framework to be developed in Part IV. Some of the design choices listed below may also be considered as refinements of requirements towards an integrated conceptual framework and tool established in Section 2.3.

C1. Repository Architecture. Section 6.4 of the previous chapter has been dedicated to different candidate architectures upon which integrated historical and logical versioning might be realized. The *asymmetric* architecture suffers from the drawback that variability in time and in space are managed in a non-uniform way, leading to an undesired heterogeneity (see Section 7.1.2). Conversely, *orthogonal* versioning does not consider the fact that the variability model itself is subject to evolution; this fosters destructive updates (Section 7.1.7). A third alternative is represented by the *hybrid* architecture, which seemingly complicates the management of particularly the middle layer—here, the feature model. Though, the hybrid approach combines the advantages of the asymmetric and of the orthogonal architecture by handling the historical and logical dimension, as well as the logical and the product dimension uniformly.

C2. Symmetric vs. Directed Deltas. Every version control system – regardless of whether being line-oriented or model-based, and of whether addressing historical or logical versioning – needs multi-version storage. In Section 4.3.1, different forms of *delta storage* have been investigated. *Snapshots*, as an unoptimized technique, rely on full storage of every version of an artifact. Since this is not feasible for intensional versioning, where the number of versions may rapidly explode, this option is not considered as a design choice here.

The two remaining options, *symmetric* and *directed* deltas, correspond to the principles of *annotative* and *transformational variability* in SPLE, respectively, as we have learned in Section 6.4.1. Both come into question for the design of the conceptual framework.

C3. Log-Based vs. Comparison-Based Differentiation. According to Section 4.3.2, *comparison-based* approaches are preferred over *log-based* approaches for the calculation of differences between versions in line-oriented VCS. This is due to the high precision and low internal complexity of sequence comparison algorithms.

Many model-driven product line management tools or version control systems, in contrast, rely on an *operation-based* (a specialization of *log-based*) paradigm, where changes carried out by developers are recorded, resulting in a precise edit log. This requires tight tool integration, which noticeably limits the choice of usable editing tools.

Generally, the choice between log-based and comparison-based versioning is also a choice between difficult technical integration and reduced precision of difference calculation.

C4. Degree of Filtering. Filtered editing has already been motivated in the introduction of this thesis, but it has turned out that few SPLE approaches follow this principle. We distinguish between *fully filtered* (one product variant is modified), *partly filtered* (a partial feature configuration leaves some variation points unbound), and *unfiltered* editing (the whole product line is edited). Which amount of filtered editing is supposed to be provided by the developed conceptual framework and tool?

In Section 2.3.2, specific requirements for the variant dimension of the to be developed conceptual framework have been listed. Obviously, **R7** (management of variability annotations) and **R8** (views on product variants) cannot be completely satisfied at a time. On the one hand, *fully filtered editing* would hide variability annotations, disallowing their direct manipulation. On the other hand, *unfiltered editing* permits arbitrary editing of annotations, but does not offer the complexity reduction gained by filtering.

C5. Hidden vs. Explicit Variation Points. The different members of a software product line are distinguished by valuation of their features, whose presence or absence is typically modeled in the form of *variation points*. These can be expressed in different ways depending on the employed programming/modeling language—for instance, through inheritance.

Giving the user the possibility to define and maintain variation points *explicitly* on the one hand facilitates conscious architectural decisions, but on the other hand also increases the cognitive complexity; see Sections 7.1.4 and 7.1.5. An alternative would consist in *hiding* variation points and having the user edit particular variants, where variation points are created *spontaneously* on demand.

C6. Internal Product Representation: Intrinsic or Extrinsic? The comparison provided in Section 6.4.1 revealed that the concepts *repository* and *platform* are mutually corresponding in VC and SPLE. We use the term *repository* also when referring to an integrated solution for historical and logical versioning. Particularly considering the requirements for model version and variation control, it remains to be decided how its contents are internally represented.

Unfiltered approaches to annotative MDSPLE rely on an *intrinsic* representation: A multi-variant domain model (MVDM) is managed as an ordinary instance of the domain metamodel. As argued in Section 7.1.3, this involves the drawback of constrained variability. Technically, this obstacle is due to the assumption that multi-variant domain models have

an *intrinsic representation*, i.e., they are persisted as an ordinary instance of the domain metamodel [WC09].

Extrinsic product representation relies on "a meta-level mechanism universally applied to any kind of model" [WC09]; this allows to circumvent variability constraints, but also destroys the compatibility of the MVDM with standard model editing tools.

C7. Transactional vs. View-Based Filtered Editing. Requirement **R8** states that *views* on single-version products shall be supported by the framework, which naturally involves filtered editing. Concerning the nature of the filter, in the literature, two different forms have been described, which we here refer to as *transactional* or *view-based* filtered editing.

In the transactional approach, there are well-defined iterations during which the filter and therefore the modifiable product version remain equal. Transactions may be opened and closed, e.g., using the VCS metaphors check-out and commit. In contrast, the view-based approach considers the filter as temporary, which is, on the one hand, more flexible, but, on the other hand, reduces the awareness of subsequent evolution steps and is therefore hard to reconcile with historical versioning.

C8. Pessimistic vs. Optimistic Synchronization. Every version control system must choose between a *pessimistic* or *optimistic* synchronization strategy (cf. Section 4.4), provided that collaborative editing is supported. Pessimistic strategies rely on temporary locks of resources, whereas optimistic versioning denotes the application of three-way merging.

C9. Centralized vs. Distributed Version Control. The pros and cons of *distributed* over *centralized* versioning strategies have been discussed in Section 4.5. As stated in Chapter 6, the distributed approach has been thoroughly researched neither for product line version control nor for MVC.

C10. Intensional on Top of Extensional Versioning, or Vice Versa? According to **R2** and **R6**, the conceptual framework is supposed to support both extensional and intensional version specification, using revision graphs and feature models as user-visible abstractions.

In [WMC01], two fundamental approaches towards the combination of both types of version specification have been discussed. On the one hand, *intensional versioning* may be implemented *on top of extensional versioning*, such that a selection in the feature model would be broken down to the selection of one element in an enumerated set of variants. This is easy to realize but artificially restricts the number of available product configurations. Conversely, realizing *extensional on top of intensional versioning* potentially complicates historical version control in favor of an on-demand construction of versions described by a predicate (such as a feature configuration).

C11. Texts as Models, or Models as Text? Realistic model-driven projects, regardless of whether or not being pursued in a multi-variant context, do not consist exclusively of model resources (which can be instances of different metamodels) but also of a multitude of text files—generated source code, configuration files, or documentation.

As explained in Section 6.2, some approaches to model versioning rely on a textual mapping of the information encoded in model instances, thus, they follow a *models as text* approach. Albeit, the inverse principle, *text as models* is also conceivable: a text may be represented as model instance either by taking its internal structure into consideration (e.g., representing the abstract syntax tree as model) or by a language-agnostic, line-oriented representation.

C12. Product Well-Formedness Analysis Approach. In Section 5.5.2, different approaches to the analysis of the well-formedness of all products derivable from the product line have been explained. Requirement **R12** motivates that one of the available methods should be applied also in the developed conceptual framework.

Product-based consistency checking may guarantee the well-formedness of only one particular product, but is easy to realize and computationally feasible. In contrast, the *family-based* approach checks for global consistency, but is inherently complex both from a computational point of view and for the end user, who must understand the results of consistency analysis conducted in a multi-variant context. With *sample-based* and *feature-based* analysis, two viable compromises have been presented.

7.3 Design Decisions

By selecting one alternative for each design choice, a list of justified *design decisions* is now derived. This list is taken into consideration while developing both the conceptual framework (see Part IV) and the tool SuperMod (see Chapter 14) in the face of the requirements listed in Section 2.3.

Some of the decisions made here cause more or less severe limitations with respect to the usability of the here contributed approach and tool. A retrospective discussion is, to this end, provided in Section 16.2.

D1. Hybrid Repository Architecture. From the three conceivable repository architectures, the *hybrid* model has been chosen. It combines the advantages of the asymmetric approach – the evolution of the variability model is supported – and of the orthogonal approach—logical and historical versioning is supplied uniformly. The increased architectural complexity is not passed to the user since a fixed version selection order is defined: first in the revision graph and then in the (selected version of the) feature model.

D2. Symmetric Deltas. This internal design decision has been resolved by selecting *symmetric* deltas. They may handle cross-references between versioned elements, which frequently occur in model-based products, in a more convenient way. This decision also determines that, behind the curtains, *annotative* variability is applied to the SPL.

D3. Comparison-Based Differentiation. Relying on logs would restrict the set of tools available for modification of the product line contents considerably; therefore, despite being technically more challenging, a *comparison-based* model differentiation approach is employed. For models, the comparison strategy relies on *universally unique identifiers*

(UUIDs) wherever applicable. Sequence-oriented contents such as text files are differentiated with the support of sequence comparison algorithms.

D4. Fully Filtered Editing. Due to the complexity passed to the user, unfiltered editing is not feasible, particularly with historical versioning in mind. Partially filtered editing would leave the choice of product complexity up to the user, who must still deal with a subset of conditionally visible content in the workspace. When considering the specific requirements of multi-variant modeling, however, a standardized generic representation of presence conditions, which would be kept in the workspace, does not exist [SBW16b]. To keep the approach working with existing single-version editing tools, the only feasible choice is *fully filtered editing*. Conversely, it is this decision that effectively enables *product-based product line development*.

D5. Hidden Variation Points. As a consequence of **D4**, the workspace does, on the one hand, not contain any version metadata (i.e., presence conditions). On the other hand, all variation points must be resolved while defining the filter. Altogether, this completely hides variation points in favor of spontaneous single-version architectural decisions.

D6.1. Extrinsic Product Representation in the Repository. Constrained variability, caused by metamodel restrictions, limits the means of expression for the definition of variation points and forces developers to coarsen the object granularity in the product space. In the conceptual framework, metamodel restrictions should be enforced for single-version models available in the workspace. To this end, *unconstrained variability* is enabled by an *extrinsic representation*, which is managed transparently in the repository.

As a consequence of this decision, multi-version metamodels for all versioned artifacts, including the feature model, EMF models, and text files, remain to be developed.

D6.2. Intrinsic Product Representation in the Workspace. To keep the selected product version compatible with arbitrary single-version editing tools, the ordinary *intrinsic representation* is used for models presented in the workspace.

To convert between the intrinsic and extrinsic representation, suitable transformations remain to be defined individually for each specific product dimension and content type. These transformation also raise the problem of well-formedness violations (**D12**).

D7. Transactional Filtered Editing. With *check-out* and *commit*, version control provides two meaningful metaphors for the enclosing of a transaction, which, when transferred to filtered editing of product lines, may be understood as a session in which several modifications are applied to a single-version product under the same scope. The decision for a transactional approach offers side benefits such as the possibility to cancel transactions (REVERT) or to retrospectively revise the scope of a change (AMEND).

D8. Optimistic Synchronization. The framework and tool support *optimistic* synchronization in the case of concurrent modifications. For this purpose, three-way merge support

for model-driven software product lines must be provided. This strategy should, however, not destroy existing design decisions such as fully filtered editing (cf. **D4**).

Therefore, the here contributed three-way merge approach relies on a twofold three-way merging strategy: First, *context-free* merging is applied as soon as concurrent modifications have been detected upon commit. Later on, *context-sensitive* merging is applied only *partially* in the variant selected at check-out. This way, the developer can stay in the single-variant tool without having to switch to a three-way multi-version context, which would introduce additional cognitive complexity to the three-way merge problem, which is difficult enough in single-variant contexts.

D9. Distributed Version Control. *SuperMod* shall fill the gap caused by a lack of distributed version control systems for models and for software product lines, respectively. Correspondingly, the underlying conceptual framework should be aware of the fact that multiple clones of the same repository have to be coordinated (cf. Section 4.5).

To this end, redefined synchronization operations PULL and PUSH, known from the VCS Git [Cha09], are added to the existing generalized operations CHECKOUT and COMMIT. This distinction also introduces a two-level organization within the revision graph.

D10. Extensional on Top of Intensional Versioning. Being the more flexible version definition principle, intensional versioning is chosen as base mechanism for version definition. Extensional versioning can be realized on top by mapping every enumerated version (i.e., each historical revision) to a configuration option and by ensuring through dedicated version constraints that exactly one configuration option is selected [WMC01].

D11. Texts as Models. With a similar reasoning, representing the lines of a text file as an ordered collection of objects contained in a model offers a greater amount of flexibility and extensibility than attempting to break down the information encoded by models into lines of a text file. Therefore, in the extrinsic product representation used in the repository internally, texts are represented as models. By default, a language-agnostic line-oriented representation is assumed.

D12. Product-Based Well-Formedness Analysis. The conceptual framework essentially offers *product-based product line development*, having the user exclusively edit single-version products. Therefore, it would be inadequate to switch to a multi-version context for syntactical well-formedness analysis. The here employed analysis strategy ties on the filtered editing model's property that products are edited *representatively* for a broader set of variants. The same property holds for analysis—product repair actions are not confined to the variant available in the workspace, but affects related variants indirectly. In this way, the effects of sample-based validation are achieved by filtered product-based analysis.

7.4 Further Design Principles

The following items of discussion have their origins not in the weighing of possible design alternatives provided in Section 7.2, but they are derived from observations of non-functional

properties or process-related issues. Nonetheless, they constitute conscious design decisions and therefore they are explicitly listed and justified in the following.

D13. Iterative Editing Model. Agile development assumes that software is developed using multiple types of *iterations* lying on different levels of granularity.

The iterations defined in agile processes are – seen from the perspective of developers – rather coarse-grained. For example, the daily increments and monthly sprints of *Scrum* [BS02] are organized by product backlog and sprint backlog, respectively.

Version control systems also foster an iterative style of development. Albeit, the iterations are comparably smaller and frequently associated with only a few lines of code or few model elements. The advantages of short-running and fine-grained commits include better change comprehension and easier orchestration of multiple developers.

The framework and tool should support both coarse-grained and fine-grained iterations.

D14. Incremental Editing Model. The provided editing model should be capable of describing the history of a versioned (MD)SPL as a sequence of increments, which can be mandatory or optional. The relationship between iterations and increments is flexible, i.e., an increment may be created by multiple fine-grained iterations; likewise, many increments may also form a coarse-grained iteration.

We here take a rather general notion of *increments*, which may refer to single features, feature combinations, or to all variants. Moreover, they may be *corrective* – bug-fixes or design refactorings affecting a limited scope of the SPL –, *evolutionary* – providing new functionality related to new or existing features –, or *variational*—introducing a new variation point and possibly subordinate variants.

D15. The Uniform Version Model as Theoretical Foundation. Taking into consideration the argument that variability in time and in space may overlap, which is also reflected by requirement **R14**, the framework should provide a *uniform* mechanism to support both evolution and logical variability.

In Section 6.4, several candidate formalisms were discussed, of which UVM is here chosen as an underlying formalism for the contributed conceptual framework and tool. Through the concepts of *choice* and *ambition*, as well as through a generalized COMMIT operation, UVM provides a universal and instrumentable theoretical foundation, relying on building blocks as simple as set theory and propositional logic. *Filtered editing* is natively supported by UVM.

D16. Propagation of Application-Level Changes. In Section 7.1.6, we have been faced with an example where *duplicate maintenance* was required due to a bug identified in a derived product. A *propagation* mechanism has been motivated to transfer changes performed in the context of application engineering back to domain engineering artifacts.

Not only does the here contributed conceptual framework support this type of propagation, but it even advances it to the central development principle. Changes are generally performed in a single-variant view and transparently propagated back into the repository.

D17. Implicitly Versioned Presence Conditions. Considering the relationships between the evolution, the variability, and the product dimension as reflected by the hybrid architecture, elements may carry different logical visibilities in different historical versions.

Here, presence conditions referring to the variability model are implicitly versioned by propositional logical *conjunction* with a historical component. Since visibilities are only extended, but never reduced, the *immutability* of presence conditions is achieved in order to allow for long-term availability of previous states of them.

D18. Hierarchical Product Space Organization. By using a fine-grained, hierarchically organized product space model, the presented framework intentionally merges the notions of *fragment*, *element*, and *element version*, which is in contrast to many related approaches, for instance, UVM [WMC01].

In UVM, it is assumed that the product space is made up of elements (also: items), which carry an identifier and for which an arbitrary number of versions can be defined. The correspondence between element identifiers, element versions, and corresponding visibilities is organized by the concept of *fragments* (see Section 8.4.1). The visibilities of different element versions are *disjoint*, such that it is ensured that the workspace always contains at most one version of an element.

When using the concept of fragments, the question of their *object granularity* inadvertently arises. Coarse-grained elements raise the need of explicitly versioning product versions for each possible combination of options involving a product-level difference; an example for this is shown in Figure 7.5, where $2^2 = 4$ versions must be maintained for two options. This way, pseudo conflicts at product level may arise when mutually exclusive product fragments represent independent changes. Conversely, fine-grained elements demand for top-level organization, i.e., hierarchical arrangement.

Although fragments may prevent many product-level conflicts a-priori by disallowing the combination of mutually exclusive element versions, they are omitted in the conceptual framework in favor of an infinitely fine-grained hierarchical product space model. Thus, the

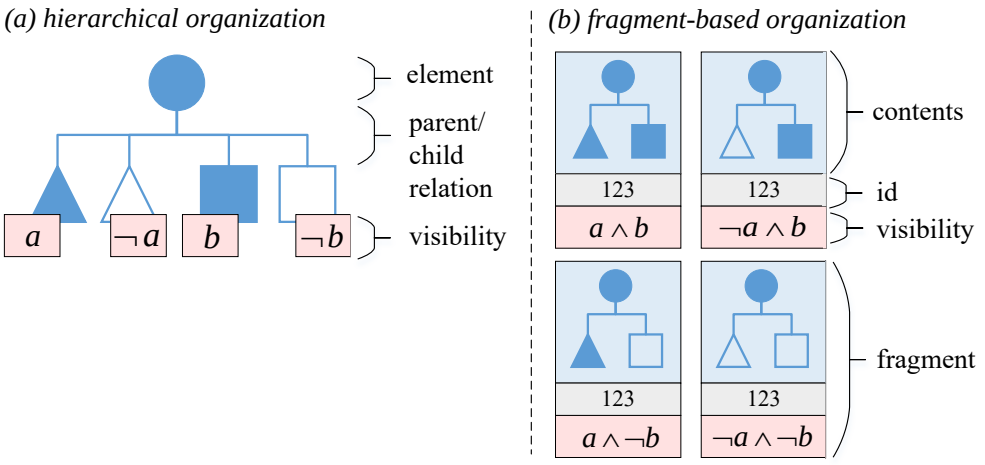


Figure 7.5: Hierarchical vs. fragment-level product space organization.

product space essentially consists of a tree of versioned items, each carrying an individual visibility. Mutually exclusive elements are detected in the form of *product well-formedness constraint violations* (see design decision **D12**).

D19. Support for Heterogeneous Interconnected Artifacts. In previous work, namely the MDSPLE tool *FAMILE* [BS12b] and the three-way merge tool *BTMerge* [SUW13b], many technical problems have emerged due to a too idealistic assumption about model-driven projects. These tools essentially treated models as self-contained EMF-compliant artifacts. Though, experiences have shown the importance of considering, on the one hand, models as interconnected entities (i.e., *resource sets*) that must be managed by a superordinate mechanism, and on the other hand, model-driven projects as a set of *heterogeneous* artifacts including non-model resources such as generated source code, configuration files, et cetera. In FAMILE, heterogeneous support was added by a retroactive extension [BS15a]. To date, BTMerge still suffers the mentioned restrictions.

SuperMod should build upon these experiences and provide support for *heterogeneous* (i.e., EMF-based and non-model) as well as *interconnected* (i.e., cross-references between models) artifacts right from the beginning.

7.5 Conclusion

The key research result provided by the thesis at hand is a conceptual framework that integrates MDSE, SPLE, and VC. The design of such a framework is not trivial as no previous attempt for such an integrated solution has been described in the literature. The design decisions made in this chapter rely on a careful interpretation of a literature review regarding the integrating disciplines MDSPLE, MVC, and SPLVC, as well as a comparison of existing approaches to integrated historical and logical versioning. Furthermore, a collection of obstacles affecting state-of-the-art solutions have been taken into account.

The central design decision concerns the architecture of the repository, where the feature model provides both a version and a product dimension. Several decisions are subordinate to the VC domain: symmetric deltas, comparison-based differentiation, optimistic synchronization, and distributed version control. Design choices in the SPL domain have been resolved with the principles of filtered editing in mind: product-based product development and analysis, hidden variation points, and transactional editing. The integration of historical and logical versioning is achieved by considering extensional as a special case of intensional versioning. In the product space, an extrinsic product model, which allows for unconstrained variability in the repository, is employed. Herein, texts are treated as model instances.

All design decisions made here are formalized and refined throughout the description of the conceptual framework in Chapters 9 until 13 of Part IV. In advance, the subsequent chapter introduces the prerequisite mathematical formalisms.

If people do not believe
that mathematics is simple,
it is only because
they do not realize
how complicated life is.

JOHN VON NEUMANN

Chapter 8

Formal Foundations

Abstract

The conceptual framework presented in this thesis builds upon two simple mathematical formalisms, namely set theory and propositional logic. In previous work, these formalisms have been combined and extended in a way that they suit the specific requirements of historical and logical version management. Also, multi-version representations for generic data structures – sets, sequences, and graphs – are introduced. Furthermore, propositional logic is extended to three values in order to deal with incomplete information as well as with satisfiability problems. The chapter is concluded by an explanation of the Uniform Version Model (UVM), the conceptual predecessor of the henceforth developed framework.

Contents

8.1 Sets, Sequences, and Digraphs

Although being an integral part of common knowledge of computer science, *sets*, *sequences*, and *directed graphs* are formally defined, since subsequently provided definitions, in particular Section 8.2, build on top of the notational and semantical details provided here.

8.1.1 Sets

A *set* S is an *unordered* collection of elements e_i, where multiple occurrences of the same element are forbidden (*uniqueness*).

$$S = \{e_1, \ldots, e_n\}, \quad \forall i, j \in \{1, \ldots, n\} : (i \neq j) \Rightarrow (e_i \neq e_j) \tag{8.1}$$

The *cardinality* $|S|$ of a set S denotes the number of elements it contains. The *empty set* $\emptyset = \{\}$ has a cardinality of 0. Furthermore, the operations *union*, *intersection*, and *difference* are defined.

$$S_1 \cup S_2 = \{e \mid (e \in S_1) \vee (e \in S_2)\} \tag{8.2}$$

$$S_1 \cap S_2 = \{e \mid (e \in S_1) \wedge (e \in S_2)\} \tag{8.3}$$

$$S_1 \setminus S_2 = \{e \mid (e \in S_1) \wedge \neg(e \in S_2)\} \tag{8.4}$$

Furthermore, we define the *subset* relationship between two sets as follows:

$$S_1 \subseteq S_2 \Leftrightarrow (\forall e \in S_1 : e \in S_2) \tag{8.5}$$

Next, the *Cartesian product* $S_1 \times S_2$ of two sets is a set of tuples associating each element of the first set with each element of the second set.

$$S_1 \times S_2 = \{(e_1, e_2) | e_1 \in S_1, e_2 \in S_2\} \tag{8.6}$$

The *power set* $\mathcal{P}(S)$ of a set contains all possible subsets U of S, including S itself as well as the empty set $\emptyset$.

$$\mathcal{P}(S) = \{U | U \subseteq S\} \tag{8.7}$$

Last, a *partition* π of a set S is a set of disjoint sets $S_1, \ldots, S_m$ whose union corresponds to S. We define a boolean predicate $partition$ that decides whether an arbitrary set of sets matches this definition:

$$\begin{aligned} &partition(\pi : \{S_1 \subseteq S, \ldots, S_m \subseteq S\}, S) \Leftrightarrow \\ &(S_1 \cup \ldots \cup S_m = S) \wedge \big(\forall i, j \in \{1, \ldots, m\} : (i \neq j) \Rightarrow (S_i \cap S_j = \emptyset)\big) \end{aligned} \tag{8.8}$$

8.1.2 Sequences

A *sequence* $\vec{S}$ is an *ordered* collection of elements e_i, where multiple occurrences of the same element are allowed (i.e., *uniqueness* is not in general required).

$$\vec{S} = [e_1, \ldots, e_n], \quad \forall e_i \in \vec{S} : e_i \in base(\vec{S}) \tag{8.9}$$

The elements e_i occurring in $\vec{S}$ are taken from a *base set*, $base(\vec{S})$, which in turn contains all elements of the sequence exactly once. Therefore the *length* $|\vec{S}|$ of a sequence is greater or equal than the cardinality of its base set: $|\vec{S}| \geq |base(\vec{S})|$.

In addition to elements of the base set, sequences may contain a generic *null element* ϵ at arbitrary positions (see Section 8.2.2).

To convert sets into sequences, we introduce an operation seq that arranges the elements contained in the base set in a random order:

$$seq(S : \{e_1, \ldots, e_n\}) = \vec{S} : [e_1, \ldots, e_n], \quad S = base(\vec{S}) \tag{8.10}$$

Furthermore, $\vec{S}(i)$ denotes the element of $\vec{S}$ at position i, where $i \in \{1, \ldots, |\vec{S}|\}$. The inverse operation $index(\vec{S}, e_j)$ returns a set of indexes denoting the locations where e_j occurs in $\vec{S}$, or the empty set in case $e_j \notin base(\vec{S})$.

A special kind of sequences is *ordered sets*, which have the additional property of *uniqueness* and which do not contain ϵ. In this special case, $|\vec{S}| = |base(\vec{S})|$.

8.1.3 Directed Graphs

A *directed graph – digraph* for short – g is a tuple consisting of a vertex set V and an edge set E.

$$g = (V, E), \quad V = \{v_1, \ldots, v_n\}, \quad E = \{e_1, \ldots, e_m\} \subseteq V \times V \tag{8.11}$$

An instance of an edge is here written as a tuple $e_{ij} : (v_i, v_j)$. By the components of its tuple, each edge connects one *source* element of the vertex set to one *target* element thereof. Correspondingly, the accessor functions $src : E \to V$ and $trg : E \to V$ are defined.

The *in-degree* of a vertex v denotes the number of edges in E having v as target; the *out-degree* corresponds to the number of outgoing edges.

A *path* $\vec{P}$ in a graph g is a sequence of edges taken from the edge set E, where the source vertex of a contained edge must correspond to the target vertex of the previous edge (if any).

$$\vec{P} = [e_1, \ldots, e_k], \quad base(\vec{P}) \subseteq E, \forall i \in \{2, \ldots, k\} : trg\big(\vec{P}(i-1)\big) = src\big(\vec{P}(i)\big) \tag{8.12}$$

The boolean predicate $path(\vec{P}, E)$ denotes whether an arbitrary edge sequence $\vec{P}$ with base set E matches this definition. The source vertex of the first edge and the target vertex of the last edge in the sequence are referred to as *path source* and *path target*, respectively: $psrc : P \to V$ and $ptrg : P \to V$, where P denotes the set of all valid paths over E.

A specific target vertex $v_i \in V$ is defined to be *reachable* from a source vertex $v_j \in V$ if there exists any path from v_i to v_j in E. This is denoted by the operator $\xrightarrow{+}$.

$$v_1 \xrightarrow{+} v_2 \;\Leftrightarrow\; \exists \vec{P} : path(\vec{P}, E) \wedge \big(v_1 = psrc(\vec{P})\big) \wedge \big(v_2 = ptrg(\vec{P})\big) \tag{8.13}$$

A *cycle* is a path whose source and target vertex are identical. To this end, a predicate $cycle$ is introduced:

$$cycle(\vec{P}, E) \Leftrightarrow path(\vec{P}, E) \wedge \big(psrc(\vec{P}) = ptrg(\vec{P})\big) \tag{8.14}$$

A graph is *cyclic* in case there exists a cyclic path over its edge set.

$$cyclic\big(g : (V, E)\big) \Leftrightarrow \exists \vec{P} : cycle(\vec{P}, E) \tag{8.15}$$

For several graph-theoretical problems, the *transitive closure* g^+ of a graph is of interest. This is another graph g^+ that contains the same vertex set as g, but its edge set E^+ represents tuples of vertices mutually reachable in E.

$$g^+ = (V, E^+), \quad E^+ = \{(v_1, v_2) \in V \times V \mid v_1 \xrightarrow{+} v_2\} \tag{8.16}$$

The inverse operation, *transitive reduction* g^-, removes as many redundant edges as possible without destroying existing reachability relationships. In contrast to closure, transitive reduction is ambiguous. Rather than fully specifying the operation, we therefore define a constraint that forbids transitive edges in E^-.

$$g^- = (V, E^-), \quad \forall (v_1, v_2) \in E^- : \big(\nexists v_3 \in V : (v_1 \xrightarrow{+} v_3 \wedge v_3 \xrightarrow{+} v_2)\big) \tag{8.17}$$

Thus, if there already exists a path (containing any vertex v_3) from the source vertex v_1 to the target vertex v_2, the direct edge (v_1, v_2) would be redundant.

Furthermore, the transitive reduction of g must still represent the same reachability information, such that:

$$(g^-)^+ = g^+ \tag{8.18}$$

The *union*, *intersection*, and *difference* of two graphs $g_1 : (V_1, E_2)$ and $g_2 : (V_2, E_2)$ is defined by the corresponding connection of vertex and edge sets, respectively:

$$g_1 \cup g_2 = (V_1 \cup V_2,\ E_1 \cup E_2) \tag{8.19}$$

$$g_1 \cap g_2 = (V_1 \cap V_2,\ E_1 \cap E_2) \tag{8.20}$$

$$g_1 \setminus g_2 = \big(V_1 \setminus V_2,\ E_1 \setminus E_2 \setminus ((V_1 \setminus V_2) \times (V_1 \setminus V_2))\big) \tag{8.21}$$

Next, a *topological sort* of a graph is an *ordered partition* – i.e., a sequence of sets of vertices – $\vec{\pi}$, representing a *partial order*, such that for every edge (v_i, v_j), the index of the set containing v_i is lower or equal than the index of the set containing v_j. Correspondingly, the predicate $topsort$ is defined for partitions of g whose element sets match this criterion: [1]

$$\begin{aligned} topsort\big(\vec{\pi} : [V_1 \subseteq V, \ldots, V_n \subseteq V], g : (V, E)\big) \Leftrightarrow partition\big(base(\vec{\pi}), V\big) \wedge \\ \forall \big((v_i, v_j) \in E, v_i \in V_i, v_j \in V_j\big) : index(\vec{\pi}, V_i) \leq index(\vec{\pi}, V_j) \end{aligned} \tag{8.22}$$

Several procedures deal with topological sorting of a graph. One example of an algorithm that is applicable to potentially cyclics graph is *Kosaraju's Algorithm* [SS03].

1 This definition of $topsort$ does not necessarily require an acyclic graph.

8.1.4 Converting Between Digraphs and Sequences

As shown below, we use graphs for a generalization of sequences, such that different mutual orders of elements are stored in a superimposed way. Subsequently, we deal with both directions for the conversion between graphs and sequences, respectively.

First, we define a function $chain$ that converts an arbitrary sequence $\vec{S}$ into a linear directed graph g, taking the base set of the sequence as vertex set and representing immediate predecessor/successor relationships defined in the sequence order as edges with corresponding source and target vertex.

$$chain(\vec{S}) = g : \big(base(\vec{S}), E_{\vec{S}}\big),\ E_{\vec{S}} = \{\big(\vec{S}(1), \vec{S}(2)\big), \ldots, \big(\vec{S}(|\vec{S}|-1), \vec{S}(|\vec{S}|)\big)\} \quad \textbf{(8.23)}$$

Second, the opposite direction, graphs to sequences, is comparably more complicated, since the graph may contain ambiguities in the form of mutually unrelated vertices or cycles. To this end, we specify in Algorithm 8.1 a universalized form of the *generalized topological sort* (GTS) algorithm presented in [SUW15], which was developed in the context of three-way model merging. The procedure linearizes the graph by first applying a topological sort and by thereafter traversing the individual components with a depth-first search, starting

---

function LINEARIZE($g : (V, E)$)
  $\vec{S} = []$
  $\vec{\pi} : [V_1 : \{v_{1_1}, \ldots, v_{1_{m_1}}\}, \ldots, V_n : \{v_{n_1}, \ldots, v_{n_{m_n}}\}] \leftarrow$ topological sort of g
  for all $V_i : [v_{i_1}, \ldots, v_{i_{m_i}}] \in \vec{\pi}$ **do**
    while $|V_i| > 0$ **do**
      if $|V_i| = 1$ **then**
        $v_i \leftarrow$ sole element of V_i
        Append v_i to S
        Remove v_i from V and all adjacent edges from E
      else
        $\vec{\delta}_i : [V_{i_1}, \ldots, V_{i_p}] \leftarrow$ partition V_i by in-degree in ascending order
        $V_{min} \leftarrow \vec{\delta}_i(1)$
        $v_{dfs} \leftarrow \epsilon$
        if $|V_{min}| = 1$ **then**
          $v_{dfs} \leftarrow$ sole element of V_{min}
        else
          $v_{dfs} \leftarrow$ select next element from V_{min}
        while $v_{dfs} \neq \epsilon$ **do**
          Append v_{dfs} to S
          Remove v_{dfs} from V and all adjacent edges from E
          Remove v_{dfs} from V_i
          if v_{dfs} has exactly one successor $v_{succ} \in V_i$ **then**
            $v_{dfs} \leftarrow v_{succ}$
          else
            $v_{dfs} \leftarrow \epsilon$
  return $\vec{S}$

Algorithm 8.1: Linearization of a digraph into a sequence.

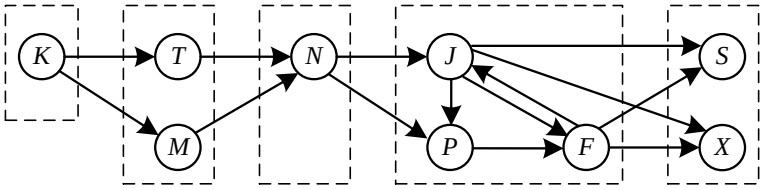


Figure 8.1: Example input for graph linearization algorithm. Modified from [SUW15, Section 4.4.1]

from vertices having the lowest in-degree. In the case of ambiguities, a non-deterministic[2] resolution strategy, e.g., user input, is utilized.

Example. Figure 8.1 depicts an example digraph to be linearized. The result of the first decisive step – topological sorting – is represented by dashed rectangles, which are assumed to be ordered from left to right. We proceed component-wise.

1. In the first component, K is selected as the sole element.
2. The second component contains two vertices with equivalent in-degree (0, after having removed the edges ensuing from K in the previous component). The non-deterministic strategy arbitrarily selects M as next vertex. The depth-first search fails (since M has no direct successor in the same component), but the next iteration of in-degree partitioning returns T as sole vertex.
3. Next, N is selected deterministically. The intermediate result is $\vec{S} = [K, M, T, N]$; furthermore, edges (N, J) and (N, P) are removed.
4. Sorting the next component by in-degree results in $\vec{\delta}_4 = [\{J, P\}, \{F\}]$. From $\{J, P\}$, the vertex J is chosen non-deterministically as starting point for the depth-first search. The search identifies P and F as next vertices, respectively.
5. All edges targeting S and X have been removed, such that their in-degree is equal. We select S non-deterministically, such that X is inserted last. g now represents an empty graph; the overall result is $\vec{S} = [K, M, T, N, J, P, F, S, X]$.

8.2 Multi-Version Sets, Sequences, and Digraphs

Above, single-version representations for sets, sequences and digraphs have been introduced. The here considered framework internally relies on a *superimposition* of multiple versions of the same data structures. To this end, we here introduce corresponding *intrinsic multi-version representations*. The explanations given below build on [Wes14, Definitions 5 and 6], where *extensional versioning* is assumed, such that the version space is defined as an enumeration of version identifiers[3]:

$$VER = \{ver_1, \ldots, ver_m\} \tag{8.24}$$

2 In the algorithmic descriptions provided in throughout the thesis, non-determinism that potentially involves user interaction is indicated by underlined statements.

3 For consistency with other definitions provided here, some symbols and identifiers have been renamed.

For the sake of simplicity, we here stick to the extensional representation; Section 8.2.4 discusses how the subsequently introduced multi-version structures can be made compatible with intensional versioning (cf. Section 4.6).

8.2.1 Multi-Version Sets

A *multi-version set* S^* is a set to whose elements several version identifiers can be assigned. It is based on a single-version set S. The connection between elements and version identifiers is established by a visibility mapping function vis. Therefore, S^* is a triple:

$$S^* = \big(S : \{e_1, \ldots, e_n\}, VER : \{ver_1, \ldots, ver_m\}, vis : S \to \mathcal{P}(VER)\big) \tag{8.25}$$

In order to decide for an element's inclusion in a specific version $ver \in VER$ of S^*, the operator $\in$ is redefined with a version-aware semantics:

$$e \in_{ver} S^* \;\Leftrightarrow\; (e \in S) \wedge \big(ver \in vis(e)\big), \quad ver \in VER \tag{8.26}$$

Correspondingly, a *filter* operator is defined. It converts a multi-version set into a single-version representation based on a given version identifier:

$$S^*|_{ver} = \{e \in S | ver \in vis(e)\}, \quad ver \in VER \tag{8.27}$$

The *union*, *intersection*, and *difference* of two multi-version sets is realized by the corresponding operations on their base sets, their version sets, and the version identifiers assigned to specific elements, such that:

$$S_1^* \cup S_2^* = \big(S_1 \cup S_2,\ VER_1 \cup VER_2,\ vis_1(e) \cup vis_2(e)\big) \tag{8.28}$$

$$S_1^* \cap S_2^* = \big(S_1 \cap S_2,\ VER_1 \cap VER_2,\ vis_1(e) \cap vis_2(e)\big) \tag{8.29}$$

$$S_1^* \setminus S_2^* = \big(S_1 \setminus S_2,\ VER_1 \setminus VER_2,\ vis_1(e) \setminus vis_2(e)\big) \tag{8.30}$$

In the three equations above, $e \in S_1 \cup S_2$.

8.2.2 Multi-Version Sequences

The transition to multi-version sequences $\vec{S}^*$ is realized in the analogous way.

$$\vec{S}^* = \big(\vec{S} : [e_1, \ldots, e_n],\ VER : \{ver_1, \ldots, ver_m\}, vis : \{1, \ldots, n\} \to \mathcal{P}(VER)\big) \tag{8.31}$$

Since equal elements can occur in multiple positions, the mapping function assigns version identifiers to indexes rather than to elements, respectively, such that $vis(i)$ denotes the visibility of the i^{th} element of $\vec{S}$, thus $\vec{S}(i)$.

The *sequence filter* operator maintains the absolute position of elements defined in the multi-version set. Unselected elements are replaced by null entries ϵ.

$$\vec{S}^*|_{ver}(i) = \begin{cases} \vec{S}(i) & \text{if } ver \in vis(i) \\ \epsilon & \text{otherwise} \end{cases} \tag{8.32}$$

Multi-version sequences as defined here may, however, not adequately express *multi-version ordered sets*, since expressing the fact that the same element occurs at different relative locations in different versions usually requires that multiple occurrences be allowed. One solution to this problem is the conversion of the ordered set into a *multi-version digraph* as described in [Wes14; SUW15]. The mutual order of elements is defined by edges representing immediate predecessor/successor relationships. Single versions of the ordered set may be obtained by selecting and traversing a single version of the graph.

8.2.3 Multi-Version Digraphs

According to [Wes14], a *multi-version digraph* is a four-tuple:

$$\begin{aligned} g^* = \big(V : \{v_1, \ldots, v_n\}, E \subseteq V \times V, \\ VER : \{ver_1, \ldots, ver_m\}, vis : (V \cup E) \to \mathcal{P}(VER)\big) \end{aligned} \tag{8.33}$$

Version identifiers are assigned to both vertices and edges. In order to ensure the *referential integrity* of single versions of the graph – i.e., to avoid dangling edges in single-version selections of the graph – the following constraint must be additionally ensured by the visibility mapping function vis:

$$\forall e_{ij} : (v_i, v_j) \in E : vis(e_{ij}) \subseteq vis(v_i) \cap vis(v_j) \tag{8.34}$$

The *union*, *intersection*, and *difference* of two multi-version digraphs is formed in analogy to multi-version sets; cf. (**8.28**) until (**8.30**):

$$g_1^* \cup g_2^* = \big(V_1 \cup V_2,\ E_1 \cup E_2, VER_1 \cup VER_2,\ vis_1(ve) \cup vis_2(ve)\big) \tag{8.35}$$

$$g_1^* \cap g_2^* = \big(V_1 \cap V_2,\ E_1 \cap E_2,\ VER_1 \cap VER_2, vis_1(ve) \cap vis_2(ve)\big) \tag{8.36}$$

$$g_1^* \setminus g_2^* = \big(V_1 \setminus V_2,\ E_1 \setminus E_2,\ VER_1 \setminus VER_2, vis_1(ve) \setminus vis_2(ve)\big) \tag{8.37}$$

In the three equations above, $ve \in V_1 \cup V_2 \cup E_1 \cup E_2$.

From a multi-version digraph, a single-version digraph may be *selected* in a straightforward way by selecting matching elements of their vertex end edge sets.

$$g^*|_{ver} = (\{v \in V | ver \in vis(v)\}, \{e \in E | ver \in vis(e)\}) \tag{8.38}$$

8.2.4 Intensional Multi-Version Structures

For the reasons of clearness and comprehensibility, we have assumed extensional versioning so far, having defined the version space as an enumerating set. Through the following changes, the discussed multi-version structures can be made applicable to *intensional versioning*:

- A *version* – still denoted as ver – is now an arbitrary boolean proposition.
- Redefine VER by using a boolean predicate $cons$ that defines whether a given version ver is consistent (cf. (**4.2**)): $VER = \{ver | cons(ver)\}$. As a consequence, the concept *version*

identifier disappears in favor of *consistent version*, which denotes any allowed user-based version specification, but is not defined more precisely.

- The combination of two version spaces is realized by combination of their predicates:
  $VER_1 \cup VER_2 = \{ver | cons_1(ver) \vee cons_2(ver)\}$;
  $VER_1 \cap VER_2 = \{ver | cons_1(ver) \wedge cons_2(ver)\}$; and
  $VER_1 \setminus VER_2 = \{ver | cons_1(ver) \wedge \neg cons_2(ver)\}$.
- Redefine the mapping function vis in a way that it does not produce a set of version identifiers, but rather returns a specific *visibility* v_i, which is a boolean-valued function that decides whether the element e_i belongs to the provided version: $v_i : VER \rightarrow \{true, false\}$.
- Rather than expecting a single version identifier ver as input, the operators $\in$ (inclusion) and $|$ (filter) now expect intensional version specifications, which must be accepted by the version consistency predicate, such that $cons(ver)$ holds.
- Replace all occurrences of the version inclusion check $ver \in vis(e_i)$ by $v_i(ver)$. Thus, the specified intensional version ver is applied to the visibility of e_i, returning a boolean decision whether the version is "included in" the visibility.

With the UVM, a concrete adaptation of intensional versioning to multi-version sets is presented in Section 8.4. Furthermore, *multi-version graphs* are revisited in Section 10.3 as a solution for applying intensional versioning to multi-version ordered (unique and/or non-unique) collections.

8.3 Three-Valued Propositional Logic

The here considered framework deals with *incomplete* information, which stems from resolving configuration decisions only partly. In particular, in the *feature ambition*, several feature options may be left unbound. Propositional logic is usually defined upon *boolean logic*, where two values, $true$ and $false$, are defined. In order to deal with incomplete information, a third value, representing the state $unknown$, must be introduced.

8.3.1 Kleene Logic

Several *three-valued* logics exist in the literature, of which *Kleene logic* [Fit91] is further considered here. In Kleene logic, a variable may have one out of three values:

$$\mathbb{K} = \{true, false, undefined\} \tag{8.39}$$

Table 8.1 presents a value table for the basic combinations of the base operators *not* ($\neg$), *and* ($\wedge$), *or* ($\vee$). From those, further logical combinations may be derived by applying propositional axioms, e.g.:

$$(a \Rightarrow b) \;\Leftrightarrow^{\mathbb{K}}\; (\neg a \vee b),\; a, b \in \mathbb{K} \tag{8.40}$$

To formally cope with three-valued Kleene logic, naming conventions and additional notation-related conversion operations are introduced here.

$\boldsymbol{a}$	$\boldsymbol{b}$	$\boldsymbol{\neg a}$	$\boldsymbol{a \wedge b}$	$\boldsymbol{a \vee b}$
$true$	$true$	$false$	$true$	$true$
$true$	$undefined$	$false$	$undefined$	$true$
$true$	$false$	$false$	$false$	$true$
$undefined$	$undefined$	$undefined$	$undefined$	$undefined$
$undefined$	$false$	$undefined$	$false$	$undefined$
$false$	$false$	$true$	$false$	$false$

Table 8.1: Value table for three-valued Kleene logic and conjunction/disjunction operators. Symmetric cases are omitted.

First of all, a *logical expression* Ξ may contain propositional logical combinations of Kleene literals defined in $\mathbb{K}$ and variables enumerated in a *variable space*. With Ξ_O, we explicitly state that an expression uses variables from a specific variable space O.

Expressions may be evaluated given a *variable binding* b, which maps each variable to a value defined in $\mathbb{K}$:

$$b : O \to \mathbb{K} \tag{8.41}$$

A binding b over a variable space O (explicitly, b_O) may be represented as a set of tuples:

$$b_O = \{(o_1, k_1), \ldots, (o_m, k_m)\}, \quad o_i \in O, k_i \in \mathbb{K}, i \in \{1, \ldots, m\} \tag{8.42}$$

Furthermore, when using this notation, tuples for *undefined* variables may be omitted. When evaluating expressions, the value $undefined$ is implicitly assumed for unbound variables.

$\Xi(b)$, or explicitly, $\Xi_O(b_O)$ denotes the logical *evaluation* of an expression with respect to a variable binding under three-valued Kleene logic. All variables referenced in Ξ are virtually replaced by their value bound in b, before simplifying the expression using the rules in Table 8.1 until the expression is reduced to a Kleene literal, which represents the evaluation result.

Tuple-represented variable bindings b may be converted into logical expressions $\hat{b}$ by conjunction of positively or negatively bound options.

$$\hat{b} = b_1 \wedge \ldots \wedge b_m, \quad b_i = \begin{cases} o_i & \text{if } (o_i, true) \in b \\ \neg o_i & \text{if } (o_i, false) \in b \\ true & \text{if } (o_i, undefined) \in b \\ true & \text{if } (o_i, \ldots) \notin b \end{cases}, \quad i \in \{1, \ldots, m\} \tag{8.43}$$

The resulting conjunction refers to bound variables by references to their identity or negation, respectively. Furthermore, references to undefined variables are eliminated by appending the neutral element $true$ to the conjunction.

8.3.2 Approximation of Constrained Satisfiability

Obviously, in case a propositional logical expression is evaluated under an option binding that contains no $undefined$ (or unbound) variables, the evaluation strategy degenerates to two-valued boolean logic. On the contrary, three-valued logic may also be used as an

approximation for *satisfiability* questions formulated in two-valued logic.

By definition, an expression Ξ over the variable set O is *satisfiable* if and only if there exists an option binding b whose application to Ξ returns $true$:

$$\big(\Xi \text{ is satisfiable over } O\big) \Leftrightarrow \big(\exists b_O : \Xi(b_O) = true\big) \tag{8.44}$$

In general, satisfiability checks are NP-complete and require a high computation effort [Lee90]. In many cases considered in this thesis, the *constrained satisfiability* of an expression Ξ with respect to a binding b is addressed, such that the conjunction $\Xi \wedge \hat{b}$ must be satisfiable. We introduce a sufficient but not necessary precondition for constrained satisfiability, which requires only linear computation time:

Theorem 8.1— Provided that Ξ is a satisfiable boolean expression and b an option binding, both over O. Then, $\Xi(b) = true$ is a sufficient precondition for the satisfiability of $\Xi \wedge \hat{b}$.

$$\big(\Xi(b) = true\big) \Rightarrow \big(\Xi \wedge \hat{b} \text{ is satisfiable over } O\big) \tag{8.45}$$

Proof. Without loss of generality, let us assume that Ξ is represented in conjunctive normal form: $\Xi = \xi_1 \wedge \ldots \wedge \xi_i$. Let us suppose that $\Xi(b) = true$, but that $\Xi \wedge \hat{b}$ is unsatisfiable over O. Ξ is satisfiable by definition; as a conjunction of options, each occurring at most once, $\hat{b}$ is satisfiable, too. The conjunction of two satisfiable expressions can only be unsatisfiable in the case of contradicting valuations. To this end, there must be a term ξ_i the algebraic signs of whose members all contradict with the signs in $\hat{b}$. Then, however, the corresponding options in b must also result in a false evaluation of the same term, such that $\xi_i = false$. This would imply $\Xi(b) = false$, which contradicts with the premise. ■

Theorem 8.2— Provided that Ξ is a satisfiable boolean expression and b is an option binding, both over O. Then, $\Xi(b) = false$ is a sufficient precondition for the unsatisfiability of $\Xi \wedge \hat{b}$.

$$\big(\Xi(b) = false\big) \Rightarrow \big(\Xi \wedge \hat{b} \text{ is unsatisfiable over } O\big) \tag{8.46}$$

Proof. Ξ be represented in conjunctive normal form: $\Xi = \xi_1 \wedge \ldots \wedge \xi_i$. Since Ξ is satisfiable but $\Xi(b) = false$, there must be at least one conjunctive term ξ_i the signs of whose members all contradict with b. As a consequence, $\hat{b} \Rightarrow \neg\xi_i$. Given that Ξ contains ξ_i and $\hat{b}$ contains $\neg\xi_i$, the conjunction $\Xi \wedge \hat{b}$ is a contradiction and therefore unsatisfiable. ■

Being allowed to use these sufficient but not necessary preconditions for constrained satisfiability has the following practical consequences:

- If $\Xi(b) = false$, then we may conclude that $\Xi \wedge \hat{b}$ is unsatisfiable. Correspondingly, $\Xi(b) = true$ safely indicates constrained satisfiability. In these cases, significant computation effort is saved.
- If $\Xi(b) = undefined$, then we cannot make any safe conclusion. We have to solve the NP-complete satisfiability problem for $\Xi \wedge \hat{b}$.
- $\Xi(\{\}) \neq false$ is a sufficient precondition for the satisfiability of Ξ. [4]

4 This way, however, not even trivial contradictions, e.g., $o_1 \wedge \neg o_1$, may be detected.

– The more variables are bound to a defined value in b, the better the approximation of satisfiability will be. On the contrary, a high degree of uncertainty included in b leads to frequent NP-complete satisfiability checks.

8.4 The Uniform Version Model

The Uniform Version Model (UVM) was repeatedly mentioned above; an informal explanation was provided in Section 6.4.4 with a focus on hybrid integration of logical and historical versioning. Now, the theoretical formalisms of UVM are supplied in their originally published form [WMC01], where the description is based on set theory and three-valued propositional logic. After recapitulating the definition of the general UVM concepts, we present the *editing model* that instantiates the concepts in order to provide an "instrumentable version engine" [WMC01]. After a short example demonstrating both the concepts and the editing model, an outlook motivates modifications to UVM that have been added to the new conceptual framework provided in Part IV.

When compared to the original literature, we use different symbols and deviating notations throughout the section, with the goal to keep the description in line with definitions provided both above and in the subsequent chapters.

8.4.1 General Concept Definitions

Although extensional adaptations have been proposed, among others in [WMC01], UVM primarily assumes *intensional versioning*.

Versions are created from a well-defined set of *options*, historical or logical configuration properties that are collected in an *option set*[5] constituting a global variable space.

$$O = \{o_1, \ldots, o_n\} \tag{8.47}$$

The *product space* is defined by means of *fragments*, triples of the form $f_i = (id_i, v_i, e_i)$, denoting (item identifier, visibility, contents). Implicitly, a *fragment set* is defined.

$$F = \{(id_1, v_1, e_1), \ldots, (id_k, v_k, e_k)\} \tag{8.48}$$

The item identifiers and visibilities of different triples may be identical. Conversely, the fragments' unique contents are taken from a global *product element set* P:

$$P = \{e_1, \ldots, e_k\}, \quad e_1 \neq \ldots \neq e_k \tag{8.49}$$

This way, versioned items are generally allowed to carry different contents in different versions. The question in which version which contents are available is answered by *visibilities* v_i, implemented by propositional logical expressions over the variables of O. The visibilities of multiple versions of a fragment – carrying the same item identifier – must

5 The option set was not explicitly defined in [WMC01]. Rather, a fixed numbers of options, identified by individual names, were assumed.

be *disjoint*, i.e., their conjunction must evaluate to *false*:

$$\forall f_i, f_j \in F : (id_i = id_j \wedge i \neq j) \Rightarrow \neg(v_i \wedge v_j) \tag{8.50}$$

UVM distinguishes between a *read filter* – a *choice* – and a *write filter* – an *ambition* –, respectively. Both are represented as option bindings in a conjunctive form. While a choice refers to a *unique* version, an ambition may represent a *set* of versions to which a performed change applies. Correspondingly, unbound options are forbidden in a choice, while in an ambition, omitted bindings are excluded by the neutral element $true$.

$$\hat{c} = c_1 \wedge \ldots \wedge c_n, \quad c_i \in \{o_i, \neg o_i\},\ i \in \{1, \ldots, n\} \tag{8.51}$$

$$\hat{a} = a_1 \wedge \ldots \wedge a_n, \quad a_i \in \{o_i, \neg o_i, true\},\ i \in \{1, \ldots, n\} \tag{8.52}$$

The choice must *represent* the ambition. Therefore, choice and ambition must agree in all option bindings. Mathematically, this has been implemented by the *implication* operator:

$$\hat{c} \Rightarrow \hat{a} \tag{8.53}$$

Furthermore, a *rule base* is a conjunction of *version rules*—logical expressions implementing the intensional predicate $cons$ that must be fulfilled by option bindings in order to represent a consistent version specification[6].

$$\mathcal{R} = \rho_1 \wedge \ldots \wedge \rho_m, \quad \rho_i \text{ is an expression over } O,\ i \in \{1, \ldots, m\} \tag{8.54}$$

The rule base is used to validate both choices and ambitions. In the case of a choice, *strong consistency* is required, i.e., the conjunction of constraints must evaluate to $true$ given the option binding as premise:

$$\hat{c} \Rightarrow \mathcal{R} \tag{8.55}$$

In the case of ambitions, however, only *weak consistency* is required. The ambition must include at least one valid version to which the associated change applies. Therefore, ambition and rule base must have a non-empty overlap, such that there exists any strongly consistent option binding $\hat{b}$ in it.

$$\exists \hat{b} : (\hat{b} \Rightarrow \hat{a}) \wedge (\hat{b} \Rightarrow \mathcal{R}) \tag{8.56}$$

8.4.2 Editing Model

On top of the definitions given above, UVM provides a *static filtered editing model*[7] that abstracts from the functional principles of *multi-version editors* [Kru84; SBK88], thus offering *transparent variability management*. The iterative editing model assumes that the user performs a change, scoped by the ambition, in a representative view, defined by the choice. The version to edit is made available in a *workspace* in form of the contents of the selected fragments. Visibilities are transparent to the user and updated automatically upon

6 In [Mun93] and [WMC01], several types of version rules were distinguished, e.g., *preferences* and *defaults*. These are revisited with modified semantics in the context of the new framework in Section 9.2.2.

7 The term *static* as used in this context is precisely defined in Chapter 11.

finishing an iteration. To formally define the transfer of contents between the fragment set and the workspace, two operations – *filter* and *write* – must be formally defined in advance.

First, *filter* takes a specific choice as argument and returns the contents belonging to the described version, which are exported into the workspace.

$$F|_{\hat{c}} = \{e_i \in P | \exists f_i \in F : f_i = (id_i, v_i, e_i) \wedge \hat{c} \Rightarrow v_i\} \quad \textbf{(8.57)}$$

Second, *write* creates a new version of the fragment set F_{new} by taking the old version F_{old}, a user-defined ambition $\hat{a}$, as well as the set of locally inserted (E_{ins}), deleted (E_{del}), and modified content elements (E_{mod}) into account:

- For inserted elements $e_{ins} \in E_{ins}$, create a new fragment $f_{new} := (id_{new}, \hat{a}, e_{ins})$. id_{new} is a new, automatically generated item identifier. Moreover, the visibility of new elements corresponds to the specified ambition: $v_{ins} := \hat{a}$.
- Process deleted elements $e_{del} \in E_{del}$ as follows: Retrieve all fragments f_{del} from which e_{del} is referenced. Then, modify the visibilities v_{del} of all fragments carrying the same identifier id_{del} as the deleted element as follows: $v_{del} := v_{old} \wedge \neg\hat{a}$. This way, the set of versions described by the ambition is removed from the visibility.
- Modified elements $e_{mod} \in E_{mod}$, whose original versions are still contained in P, are broken down into insertions and deletions. Thus, add a new fragment $f_{new} := (id_{mod}, \hat{a}, e_{mod})$ and change the visibility of all old versions of the fragment as follows: $v_{new} := v_{old} \wedge \neg\hat{a}$.

Theorem 8.3— UVM's *write* operation preserves the property of disjointness of fragments' visibilities as required by (**8.50**).

Proof. Given are two fragments $f_x, f_y \in F$ referring to the same item identifier id. Their original visibilities v_x and v_y are disjoint: $\neg(v_x \wedge v_y)$. Without loss of generality, we assume that v_x was selected by the choice ($\hat{c} \Rightarrow v_x$) and v_y was filtered out ($\hat{c} \not\Rightarrow v_y$). We must consider two cases, deletion of e_x and modification of e_x into e'_x.

Deletion. f_x is updated as follows: $(id, v_x \wedge \neg\hat{a}, e_x)$. Under the premise $\neg(v_x \wedge v_y)$, the visibilities are still disjoint: $\neg((v_x \wedge \neg\hat{a}) \wedge v_y)$, thus, $\neg(false \wedge \neg\hat{a}) = true$.

Modification. f_x is updated to $(id, v_x \wedge \neg\hat{a}, e_x)$, f_y is updated to $(id, v_y \wedge \neg\hat{a}, e_y)$, and a new fragment $f'_x = (id, \hat{a}, e'_x)$ is inserted into F. The visibilities of the three fragments f_x, f'_x, and f_y are pairwise disjoint: First, $\neg(\hat{a} \wedge (v_x \neg\hat{a})) = \neg(false \wedge v_x) = true$ for any v_x. Second, $\neg((v_x \wedge \neg\hat{a}) \wedge (v_y \wedge \neg\hat{a}))$ corresponds to the deletion case after eliminating the duplicate conjunctive term $\neg\hat{a}$. Third, $\neg(\hat{a} \wedge (v_y \wedge \neg\hat{a})) = true$ for any v_y. ■

Both operations are combined in UVM's *static filtered editing model*. Each iteration of it is organized as follows:

1. Let the user specify an ambition $\hat{a}$ that delineates the scope of the change to be performed. The ambition must be weakly consistent (cf. (**8.56**)).
2. Let the user specify a choice $\hat{c}$ that designates the version in which the change shall be applied. The choice must be strongly consistent (cf. (**8.55**)) and, furthermore, must be representative for the ambition (cf. (**8.53**)).

3. Filter the contents according to the choice (cf. (**8.57**)) and make them available for modification in a workspace.
4. As soon as the user has finished the iteration, detect insertions, deletions, and modifications by comparing the original and the modified workspace contents. Apply the *write* operation as defined above, and replace F_{old} by F_{new}. Furthermore, clear the workspace contents in order to prepare for subsequent iterations.

8.4.3 Example

The usage of UVM is now illustrated by abstracting from the running *Graph* example as represented in Section 2.6 on page 29. The example also demonstrates how a hybrid integrated solution to historical and logical versioning – as claimed in Section 6.4.4 – can be conceptually realized.

First of all, we define the *product space* by its element set P, which abstracts from the concrete product space representation of elements:

$$P = \{e_{Graph}, e_{Vertex}, e_{Edge}, e_{label}, e_{weight}\}$$

For defining the version space, we assume that some features of the example feature model, as well as three revisions, are mapped to options:

$$O = \{f_{Weighted}, f_{Labeled}, f_{Colored}, r_1, r_2, r_3\}$$

The rule base includes two version rules ensuring that the selection of a revision option requires the selection of the corresponding predecessor option.

$$\mathcal{R} = \{r_2 \Rightarrow r_1,\ r_3 \Rightarrow r_2\}$$

The connection between elements $e \in P$ and options of the version space is established in the *fragment set*:

$$\begin{aligned} F = \{&(Graph, r_1, e_{Graph}), (Vertex, r_1, e_{Vertex}), (Edge, r_1, e_{Edge}),\\ &(label, f_{Labeled} \wedge r_2, e_{label}), (weight, f_{Weighted} \wedge r_3, e_{weight})\} \end{aligned}$$

For a better readability, we here use user-defined (e.g., $Edge$) rather than artificially generated item identifiers.

To demonstrate UVM's editing model, in a new revision 4, a new element e_{Color} is introduced that is connected to option $f_{Colored}$:

0. As a first bookkeeping step, the revision graph must be organized. To this end,the new revision option r_4 is added to O and the rule $r_4 \Rightarrow r_3$ is added to $\mathcal{R}$.
1. The *ambition* $\hat{a}$ defines the scope of the change to be performed:
   $\hat{a} = f_{Colored} \wedge r_4$.
   The ambition is weakly consistent according to (**8.56**).
2. The *choice* $\hat{c}$ is a unique representative version included by $\hat{a}$:
   $\hat{c} = \neg f_{Weighted} \wedge \neg f_{Labeled} \wedge f_{Colored} \wedge r_1 \wedge r_2 \wedge r_3 \wedge r_4$.

The choice is strongly consistent according to (**8.55**) and represents the ambition as required by (**8.53**).

3. The product space is transparently filtered by elements whose visibilities satisfy $\hat{c}$: $F|_{\hat{c}} = \{e_{Graph}, e_{Vertex}, e_{Edge}\}$.
   In the workspace, the element e_{Color} is newly introduced.
4. After having finalized the edit session, the changes are written back to the repository's product space P, which is extended by e_{Color} transparently. Furthermore, the fragment set is extended by the tuple $(Color, f_{Colored} \wedge r_4, e_{Color})$, which is created based on the detection of the inserted element e_{Color}. As a consequence, the change is visible only for revisions 4 or later of variants that include $f_{Colored}$.

8.4.4 Outlook

By intention, UVM was developed in a generic way from the beginning, making only minimal assumptions about the structure of the version space and of the product space. The example has demonstrated that UVM may provide a basis for hybrid historical and logical versioning. In addition, the editing model shares similarity with the check-out/modify/commit workflow provided by version control systems. The new conceptual framework contributed in this thesis is an extension and specialization of UVM that specifically considers the design decisions stated in Section 7.3. The most prominent differences are listed below.

– The new framework explicitly combines historical and logical versioning, the former of which is managed completely transparently in order to provide fully-fledged version control functionality. (In particular, the bookkeeping step 0 of the example is automated.)

– Both the product space and the version space are concretized by providing mappings of the option set and the element set to SCM-related, model-driven, or SPL-related formalisms such as revision graphs, domain models and feature models.

– The concept of *fragments* is removed in the new framework. In favor of this, *fine-grained hierarchical* versioning is applied, where each detail of the product space is considered as a versioned item, such that there is essentially no difference between *item* and *item version*. Correspondingly, (**8.50**) and Theorem 8.3 become obsolete.

– Therefore, mutually exclusive workspace contents are not checked on the base layer level in the new framework. As a replacement, context-free well-formedness conditions are used to detect equivalent, but also context-sensitive, conflict types.

– For the same reason, the update operation does not include a dedicated handling for *modified* elements; rather, the operation is broken down into insertions and deletions of child elements in the tree hierarchy formed by the product space.

– The editing model is *dynamic* inasmuch as the option space may co-evolve with the product space, and the ambition may be specified at any point in time within an iteration. Rather than clearing the workspace contents after commit, it is assumed that the user continues in the existing workspace view as usual in VC.

Precise discussions of the differences between the new conceptual framework and UVM are given in the related work sections of the individual chapters of the subsequent Part IV.

Part IV

An Integrated Conceptual Framework

[Extensional] version control systems should not be used for feature variability [. . .].

SVEN APEL, DON BATORY, CHRISTIAN KÄSTNER AND GUNTER SAAKE (2013)

Chapter 9

Hybrid Version Model

Abstract

In the following, the theoretical core contribution of this thesis is presented: a conceptual framework for the integration of MDSE, SPLE, and version control. This chapter considers the conceptual framework from the perspective of version management, relying on two distinct formalisms. The structural part of the framework is defined by Ecore-compliant metamodels. For the functional and behavioral perspective, the notions of UVM are extended and specialized, still relying on (multi-version) set theory and propositional logic. First, architecture and functionality of the whole conceptual framework are sketched, before the focus is put on the version space. This is defined by an abstract base layer, which is concretized by mappings to revision graphs and feature models, which are offered as version definition metaphors to the user. We present a preliminary editing model, which combines historical and logical versioning in the face of the previously discussed requirements. The presentation is rounded out with two pieces of optimization, namely the change space and so called visibility forests. Finally, related conceptual approaches are discussed.

Contents

9.1 Architectural and Functional Overview

The framework utilizes well-known abstractions both from version control and from software product line engineering. VC contributes the update/modify/commit paradigm as editing model and the revision graph as a version definition and selection abstraction. From SPLE, feature models and feature configurations are borrowed for the definition of the version space as well as for specific versions therein. For delineating the scope of a change, the concept *feature ambition* is newly introduced here. Both disciplines are combined in a *hybrid version model* whose formal basis is the Uniform Version Model (see Section 8.4).

The chapter is organized as follows: First, user-visible abstractions of the framework as well as the architecture of the transparent repository are presented in Sections 9.1.1 until 9.1.3. In Section 9.2, definitions for the *version space base layer*, which consist of Ecore-compliant metamodels and of mathematical definitions based on the UVM, are provided. The base layer in turn is instantiated by revision graphs and by feature models, as explained in Sections 9.3 and 9.4, respectively. These version dimensions are finally combined into a *preliminary editing model* in Section 9.5, which also contains a summarizing example. Two sections are dedicated to optimization of the framework. First, the *change space* is described as an optimizing, user-invisible base layer instantiation in Section 9.6. Second, in Section 9.7 *visibility forests* are introduced as a global data structure for version

membership information. Related work is presented in Section 9.8, before the chapter is concluded with a summary. [1]

9.1.1 Abstractions for Version Space Definition and Version Selection

The conceptual framework automates version management by exposing version definition and version selection abstractions, represented by familiar concepts of VC and SPLE, to the user. These concepts must be well-understood in order to guarantee a correct and meaningful behavior of the framework. Figure 9.1 visualizes the abstractions introduced subsequently. Both for choices and for ambitions, the framework assumes a fixed version selection order: first in the revision graph, then in the feature model. This is due to the hybrid architecture, which has been justified by design decision **D1** in Section 7.3.

Revision Graph. For historical versioning, the framework utilizes a *sequence* of revisions, such that branches are disallowed (see Section 4.2.2). The elements of this sequence are here referred to, however, as nodes of a graph in order to conceptually prepare for multi-user versioning (see Chapter 12). Each revision corresponds to one historical state of the product line; the selection of multiple or no revision is forbidden. In this way, *extensional versioning* is realized. For each node in the revision graph, details such as the revision number, the commit message, author, as well as the revision creation date are recorded. Rather than being available for arbitrary modification, the revision graph may be extended only indirectly by committing one new revision per edit session.

Revision Choices. A *choice* in a revision graph, seen from the user's perspective, consists in the selection of one revision, i.e., a dedicated node in the revision graph. When confining to historical versioning, precisely the state that has been committed as the selected revision

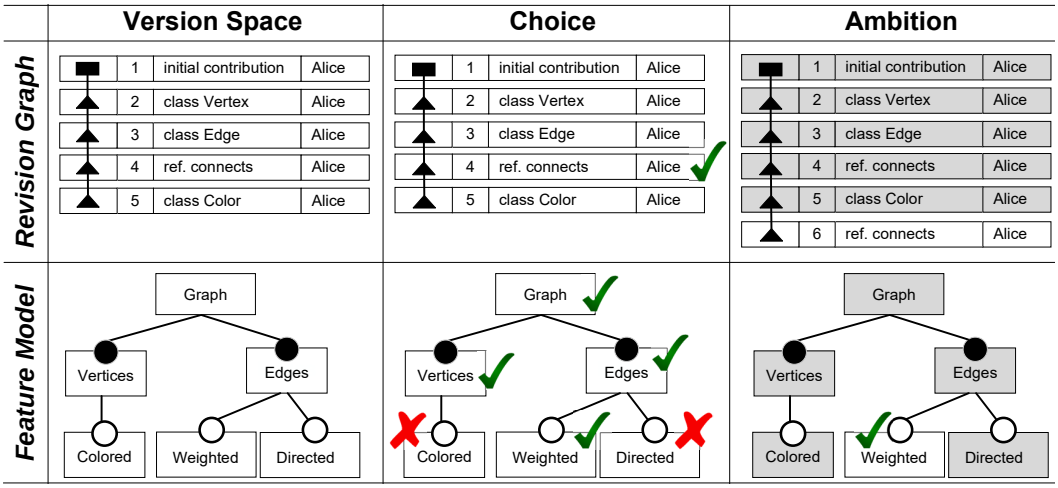


Figure 9.1: Version definition and selection abstractions provided by the framework.

1 A large portion of the text of this chapter was pre-published in [Schwä+15; Schwä+16].

is supposed to be reproduced in the workspace, whereas all changes performed in later revisions (i.e., successors of the selected node) remain invisible.

Revision Ambitions. For the VCS user, the concept *revision ambition* is almost transparent. When committing, a new revision is created automatically as a successor of the most recent revision, the *head*. Furthermore, details such as commit date, author, and revision number can be inferred automatically (e.g., by auto-incrementing a revision counter). The only detail to be added manually is a *commit message* that describes the user's intent(s) behind the committed change.

Feature Model. Logical variability is managed by a *feature model*, which provides a high-level representation of individual properties of a system. Unless stated otherwise, we assume feature models in their form as presented in Section 5.2. In contrast to the revision graph, the feature model may be arbitrarily modified by the user in the course of an edit session, by adding new features, changing details of features, or removing features. To this end, the feature model is made available as an additional artifact in the workspace.

Feature Choices. As the feature model defines features as configuration options, a unique selection in the feature model corresponds to what has been introduced as *feature configuration* in Section 5.2. Thus, the user is requested to assign a unique selection state (selected or deselected) to each feature. Moreover, the selected configuration must conform to version selection rules defined in the feature model, including parent-child relationships, groups, and requires/excludes relationships—the precise formal semantics of these constraints are explained in Section 9.4.

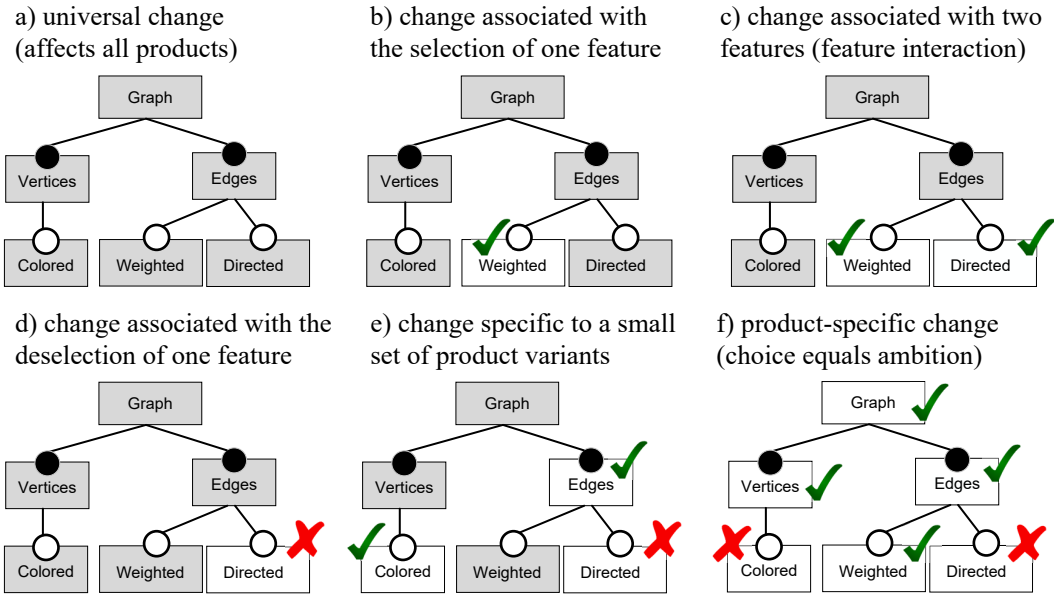


Figure 9.2: Example feature ambitions with different specificity.

Feature Ambitions. After selection of a specific product variant by a feature choice, the user applies changes to the working copy of the artifact made available in the workspace. Nevertheless, these changes are seldom confined to the selected product variant, but typically address a broader scope of product variants. During commit, therefore, the affected versions must be identified. This is where the novel concept of *feature ambition* steps in.

A feature ambition is a partial selection in the feature model, such that in addition to *selected* and *deselected*, a third selection state, *neutral*, is introduced. To neutral features, the performed change is immaterial, i.e., it applies for product variants where the respective feature is selected or deselected.

To illustrate the concept of feature ambition, several examples are provided in Figure 9.2. Feature ambitions reside in a continuum between *universal* (i.e., all variants are affected), and *product-specific* (i.e., only the variant described by the choice is affected). In general, the more features are bound by non-neutral selection, the more *specific* is the change and the *smaller* is the set of affected versions.

9.1.2 Repository Architecture

Constituting the entirety of available product versions, the *repository* is provided as a persistent storage for the elements of the version space and of the product space. Like in state-of-the-art version control systems, the contents of the repository are transparent to the user, who accesses the repository indirectly by check-out and commit. Here, a specific architecture is underlying the repository. Its elements are explained on a coarse-grained level subsequently.

As depicted in Figure 9.3, the conceptual framework consists of elements distributed over four spaces: A *revision graph*, which controls the historical evolution of both the *domain model* and the *feature model*, which are summarized as *product space*. Elements of the domain model are additionally versioned with respect to the feature model in order to provide for logical variability. Optionally, both the revision space and the feature space are organized by a transparent *change space*, an optimization that reduces run-time and memory consumption and thereby increases comprehensibility. The revision graph, the change space, and the feature model are subsumed under the term *version space*.

The feature model plays a dual role by being part of both the version space and the product space. For the revision space, it is versioned the same way as the product space; for

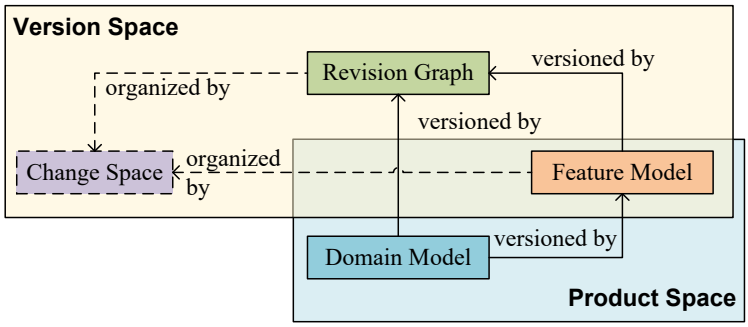


Figure 9.3: Relationships between different members of version and product space.

the product space, it incorporates an additional variability model.

The relationship "versioned by" needs to be further explained. It is conceptually realized by transparent *visibilities* assigned to elements of the space that is source of the relationship. Visibilities in turn refer to elements of the relationship target. In particular, elements of the feature model contain visibilities that refer to specific revisions, whereas elements of the domain model carry hybrid visibilities that refer both to revisions and to features.

Internally, elements of the version space are mapped onto abstractions borrowed from the UVM (cf. Section 8.4), such as *options* and *version rules*. This mapping happens behind the scene, while only the higher-level abstractions of revision graph and feature model/configuration are presented to the user for version selection and version space editing.

9.1.3 The Editing Model as Seen by the User

The interaction between the workspace and the repository is organized by generalized forms of the version control metaphors *check-out* and *commit*, which utilize choices and ambitions in the revision graph and in the feature model, respectively. Each edit session instantiated from the iterative editing model consists of three – partly automated – steps, which are further refined below. Figure 9.4 complements the explanations.

Check-Out. The user performs a *version selection* (a *choice*) in the repository. In the revision graph, the selection comprises a single *revision*. In the (selected revision of the) feature model, a *feature configuration* has to be specified. A single-version working copy of the repository contents, filtered by the selected version, is exported into the workspace.

Modify. In the workspace, the user applies a set of changes to the single-version domain model and/or to the feature model.

Commit. The changes are written back to the repository. For this purpose, the user is prompted for an additional selection of a *feature ambition* to delineate the logical

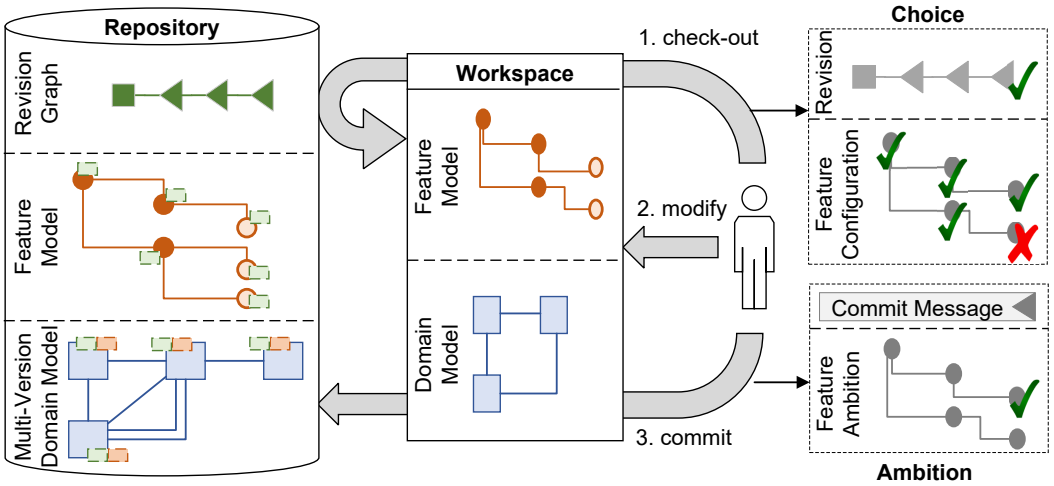


Figure 9.4: The editing model provided by the framework. Based on [SBW15, Figure 8].

scope of the changes applied to the domain model. Visibilities of versioned elements are updated automatically, and a newly created revision – to be described by a commit message written by the author of the change – is submitted to the repository.

The editing model reflects the design decisions of an *iterative* and *incremental editing model* (**D13** and **D14**). Furthermore, the suggested *propagation* mechanism (**D16**) is provided; see Section 7.3.

9.2 Version Space Base Layer

In the following, the *base layer* of the *version space*, which is subsequently concretized by revision graph, feature model, and change space, is presented. For the description, we rely on a combination of two formalisms. The structural part is introduced in the form of metamodels, which describe instantiable elements and their possible relationships part of the repository[2]. The semantics of the framework is defined on the basis of propositional logic and set theory, where concepts of UVM (cf. Section 8.4) are reused and extended as suggested by design decision **D15**.

9.2.1 Core Metamodel

The general architecture of a repository is described by the *core metamodel* depicted in Figure 9.5. By providing a mapping between version space and product space, the approach follows *annotative variability* and therefore makes use of *symmetric deltas* (see **D2**).

A *repository* combines a version space and a product space, which are in turn divided up into several *product dimensions* and *version dimensions*. A product dimension contains

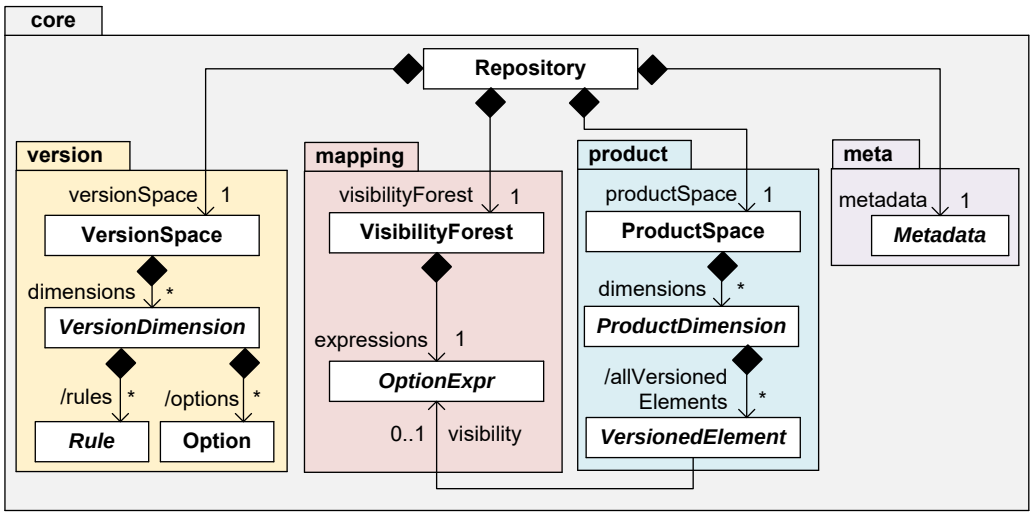


Figure 9.5: The core metamodel of the conceptual framework. Based on [SBW16a, Figure 2].

2 In the model-driven implementation of the conceptual framework, these metamodels have been taken as input for the generation of initial source code for SuperMod; see Chapter 14.

a tree of *versioned elements*, to each of which a *visibility* is assigned. Those visibilities are organized in a global mapping structure, the *visibility forest*, which is explained in Section 9.7. A version dimension contains *options* and *(version) rules*. Both visibilities and specific sub-kinds of version rules are represented by *option expressions*, propositional logical expressions on options (cf. Section 9.2.3). Contents of the packages product and meta are further refined in Chapters 10 and 13.

By leaving product and version dimensions abstract, the framework is highly extensible with respect to the concrete product and version space used in a specific versioning scenario. This way, different repository architectures than described above, e.g., purely historical or logical versioning, or variability formalisms different from feature models may be supported. Below, however, we stick to the hybrid architecture sketched in Figure 9.3.

9.2.2 Version Space Core

Let us further refine the package core::version, which represents a base layer of the version space to be instantiated by the revision graph, the feature model, and the change space below. A detailed structural view is provided in Figure 9.6. When compared to UVM, we here introduce a *specialized rule base* that divides version rules up into three categories, *invariants*, *preferences*, and *defaults*, which have been originally described in [Mun93] but whose semantics have been modified here (see related work).

Options. An *option* represents a (logical or historical) property that is either present or absent in an individual product. Each version dimension defines a global *option set*, whose elements are distinguished and uniquely identified by their index.

$$O = \{o_1, \ldots, o_n\} \tag{9.1}$$

It depends on the higher-level version space implementation how the derived reference /options, which makes this set available, is realized.

Specialized Rule Base. In the here considered conceptual framework, the rule base does not only serve to *validate* choices and ambitions, but also *complete* them in order to assist

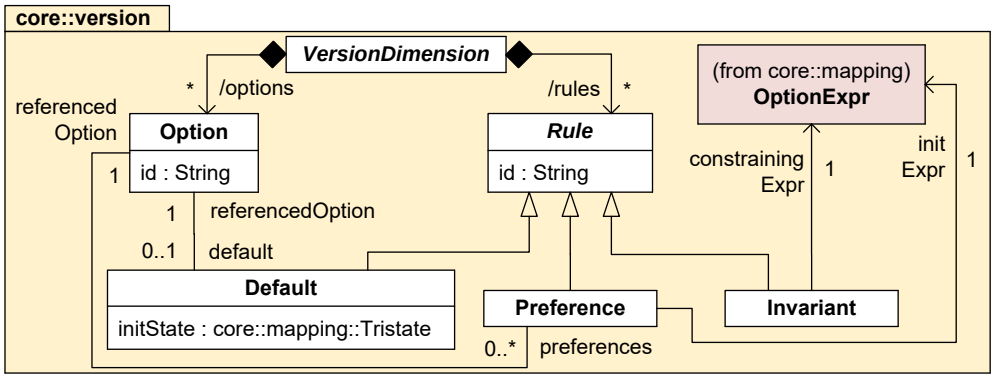


Figure 9.6: Detailed metamodel for the version space core in the base layer.

the user in consistent version selection. For this sake, *preferences* and *defaults* are employed as an orthogonal constraint propagation mechanism designed to enforce (not necessarily all) invariants. [3]

In its specialized definition, the *rule base* $\mathcal{R}$ is a triple consisting of an *invariant* set $\mathcal{J}$, a preference set $\mathcal{P}$, and a default set $\mathcal{D}$:

$$\mathcal{R} = (\mathcal{J}, \mathcal{P}, \mathcal{D}) \tag{9.2}$$

Invariants. Having a purely validating semantics, *invariants* adopt the notion of version constraints. Thus, $\mathcal{J}$ is defined by a conjunction of invariants ρ_j, each being represented as a logical expression over the option set O of the version dimension (see constrainingExpr in the metamodel).

$$\mathcal{J} = \rho_1 \wedge \ldots \wedge \rho_m,\ \rho_j \text{ is an expression over } O,\ j \in \{1, \ldots, m\} \tag{9.3}$$

Preferences. In complement, elements of the *preference set* $\mathcal{P}$ define initialization expressions π_{i_j} (initExpr) for unbound options o_{i_j} (referencedOption) of the option set O.

$$\mathcal{P} = \{(o_{i_1}, \pi_{i_1}), \ldots, (o_{i_k}, \pi_{i_k})\},\ o_{i_j} \in O,\ \pi_{i_j} \text{ is an expression over } O,\ i_j \in \{1, \ldots, k\} \tag{9.4}$$

Defaults. Last, *defaults* contained in $\mathcal{D}$ are applied only for options that are not connected to a preference, or all of whose corresponding initialization expressions returned $undefined$. They define a default state s_j (initState) for unbound options o_j (referencedOption) of O. For each option, at most one default may be defined.

$$\mathcal{D} = \{(o_{i_1}, s_{i_1}), \ldots, (o_{i_l}, s_{i_l})\},\ o_{i_j} \in O,\ s_{i_j} \in \{true, false\},\ i_j \in \{1, \ldots, l\} \tag{9.5}$$

Below, the term *version rule* is used as a generalization of invariants, preferences and defaults. It also depends on the concrete version space implementation how the derived reference /rules is realized.

9.2.3 Option Expressions

Option expressions are propositional logical expressions over the option set. They appear, among others, within the visibilities of versioned elements, and in logical expressions assigned to invariants and to preferences. Here, the structure, the formal syntax, and the evaluation semantics of option expressions are clarified.

As shown in the metamodel in Figure 9.7, there exist three categories of option expressions. *Option references* target an option $o \in O$. *Compound expressions* hierarchically combine

3 Recall that invariants, preferences, and defaults are not defined manually by the user, but automatically derived by the framework from higher-level descriptions such as feature models (see Section 9.4) and revision graphs (see Section 9.3). The corresponding mappings to the low-level rule base also have to ensure that preferences and defaults are in line with corresponding invariants.

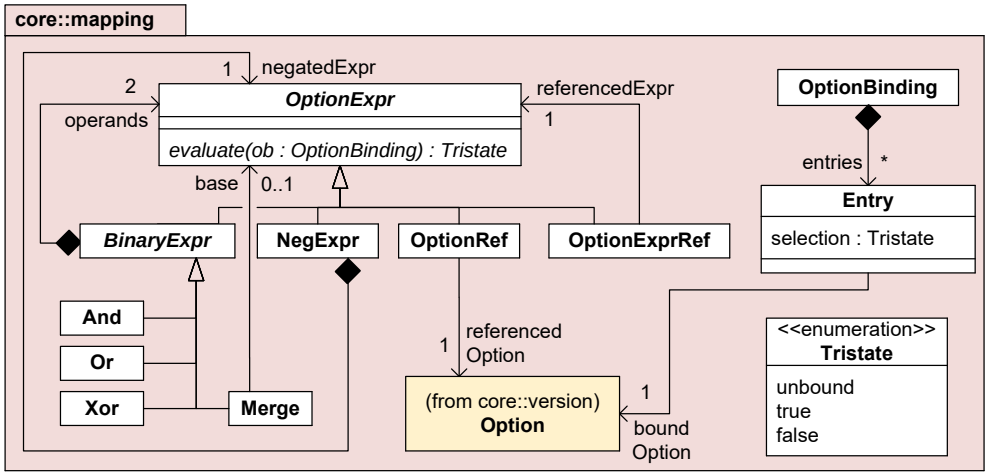


Figure 9.7: Metamodel for option expressions and option bindings.

option expressions (e.g., the negation $\neg$ is represented by NegExpr, the conjunction $\wedge$ by AndExpr)[4]. *Option expression references* re-use existing expressions in order to avoid their repeated duplication (see Section 9.7).

Choices and ambitions are internally represented as *option bindings*, sets of entries binding an option to a selection state. The enumeration Tristate defines the three values allowed in *three-valued Kleene logic* (cf. Section 8.3).

Option expressions can be evaluated with respect to a given option binding. This is realized by corresponding implementations of the operation evaluate in the subclasses of OptionExpr. During evaluation, options are virtually replaced by the bound Kleene value. The expression is then reduced to a value literal, which is returned as result.

9.2.4 Option Bindings and their Completion

Choices and ambitions, denoted by the symbols c and a, are represented as instances of OptionBinding, which reflect *variable bindings* as introduced in Section 8.3.

As mentioned above, preferences and defaults assist the user in version specification by assigning selection values to options not considered explicitly by a given variable binding. The procedure of applying the semantics of preferences and defaults is called *completion*. It consists of two steps, *preference application* and *default application*.

Preference Application. As defined by Algorithm 9.1, preference application is realized by traversing the option set, looking for unbound options, and applying a suitable preference. In case several preferences are defined for an option, but their evaluation yields different boolean results, the situation is resolved non-deterministically. [5]

4 The role of Merge expressions is explained in Section 12.4.4.

5 The non-determinism is caused by the preference set being unordered. Such situations, however, should be avoided by corresponding higher-level mappings. This is the case for the revision graph and feature model mappings provided subsequently.

```
function ApplyPreferences(b)
    for all o ∈ O do
        if ((o, true) ∉ b) ∧ ((o, false) ∉ b) then
            for all (o, π) ∈ P do
                s ← π(b)
                if s ≠ undefined then
                    b ← b ∪ (o, s)
                    break
    return b
```

Algorithm 9.1: Preference application.

```
function ApplyDefaults(b)
    for all (o, s) ∈ D do
        if ((o, true) ∉ b) ∧ ((o, false) ∉ b) then
            b ← b ∪ (o, s)
    return b
```

Algorithm 9.2: Default application.

Default Application. Since at most one default may be defined for each option, no conflicting situations may occur here. Algorithm 9.2 precisely defines default application.

Completion. This comprises the application of preferences and defaults. As preferences have a higher priority, they must be taken into account prior to defaults. Preferences may, however, influence each other. Furthermore, after having applied a default, a previously inapplicable preference may become applicable, too. Therefore, as formalized by Algorithm 9.3, preferences are enforced twice, before and after default application, and moreover, in a loop that terminates as soon as no preference could be applied.

In the subsequent chapters, the shorthand notations ${}^{\mathcal{P}}b$ and ${}^{\mathcal{D}}b$ indicate that the algorithms ApplyPreferences and ApplyDefaults have been applied to a given option binding b.

```
function Complete(b)
    curr ← |b|
    last ← curr − 1
    while curr > last do
        last ← curr
        ApplyPreferences(b)
        curr ← |b|
    last ← curr
    ApplyDefaults(b)
    curr ← |b|
    while curr > last do
        last ← curr
        ApplyPreferences(b)
        curr ← |b|
```

Algorithm 9.3: Option binding completion.

Furthermore, ${}^{\mathcal{PD}}b$ denotes that the COMPLETE operator has been utilized.

Notice that the application of preferences or defaults does not allow for any guarantees with respect to the "uniqueness" of an option binding. It may still include unbound options and therefore represent a set of versions rather than a single version. Furthermore, on base layer level, preferences (not defaults) may be contradicting, such that non-determinism is introduced to APPLYPREFERENCES. It must be ensured by the higher-level mappings that such situations are avoided.

9.3 Mapping Revision Graphs to the Version Space Base Layer

In SCM and particularly in version control, *evolution* and *collaboration* of software development are addressed. This chapter focuses on the evolutionary aspects of VC, assuming a single-user environment. The coordination of several collaborating users is a subject of Chapter 12.

The history of a VC repository is typically represented by a *revision graph*, which is here reduced to a linear sequence. Revision control deviates from variability management in two aspects. First, revisions are organized *extensionally*, i.e., only revisions that have been committed earlier may be checked out. Second, revisions are *immutable*: Once committed, they are expected to be permanently available, and should not be affected by *destructive updates* (see Section 7.1.7).

9.3.1 Structural Design

Figure 9.8 depicts the structural perspective onto revision graphs, which play the role of a *version dimension*. A revision graph contains revisions as vertices, for each of which particular version details are persisted; here, revision number, date, a commit message, and the name of the authoring user are defined as attributes. The linear revision order is expressed by the (acyclic and irreflexive) predecessor/successor relationship. Both classes

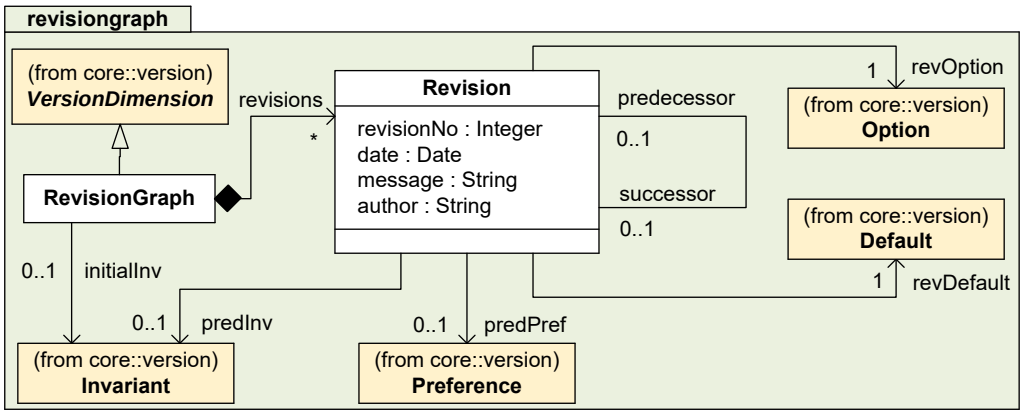


Figure 9.8: Metamodel for revision graphs including references to core concepts. Based on [Schwä+16, Figure 7].

Table 9.1: Mapping revision graphs to low-level rule base elements. Based on [Schwä+16, Table 2].

Pattern	Transformation		Metamodel
revision i	option	r_i	revOption
non-initial revision i	default	$(r_i, false)$	revDefault
initial revision 0	invariant	r_0	initialInv
successor revision $i+1$ of revision i	invariant	$r_{i+1} \Rightarrow r_i$	predInv
	preference	(r_i, r_{i+1})	predPref

RevisionGraph and Revision contain references to concepts of the base versioning layer; their semantics are explained below.

9.3.2 Formal Mapping

Revisions are intended to supersede each other in a series of commits. Nevertheless, it is crucial that old states of the versioned data may be restored. For this reason, committed changes are connected to an *option* that clearly identifies the revision and makes it available for selection. Thus, historical versioning is realized by mapping each revision i to a *revision option* r_i. This way, *extensional versioning* is realized *on top of intensional versioning*; cf. design decision **D10**.

In contrast to revisions themselves, revision options are interpreted with a transitive semantics in mind: when selecting a revision, the workspace should not only incorporate the changes associated with the corresponding revision option but also those changes which have been committed earlier along with predecessor revisions. As summarized in Table 9.1, the initial revision option r_0 must always be selected, such that the corresponding baseline in the product space is always active. To implement the transitive semantics, invariants of the form $r_{i+1} \Rightarrow r_i$ are introduced for consecutive revisions r_i and r_{i+1} transparently.

The options, invariants, preferences, and defaults derived this way from a revision graph are referred to as O_r, $\mathcal{J}_r$, $\mathcal{P}_r$, and $\mathcal{D}_r$, respectively.

9.3.3 Version Selection

Revision Choices. A version in the revision graph is selected as a single revision j by the user. The invariants described above require that, in addition to r_j, all options of predecessor revisions be selected. To ease version selection, a preference has been introduced to the mapping shown in Table 9.1. Repeated applications of (r_i, r_{i+1}), beginning with the selected option r_j, propagate backwards until the initial revision r_0. All remaining – i.e., more recent – revisions' options are explicitly deselected by the default $(r_i, false)$. Effectively, after applying preferences and defaults, a *revision choice* ${}^{\mathcal{PD}}c_r$ is derived as:

$$
\begin{aligned}
{}^{\mathcal{PD}}c_r &= \{(r_0, b_0), \ldots, (r_n, b_n)\},\ r_i \in O_r,\ i \in \{0, \ldots, n\}, \\
b_i &= \begin{cases} true & \text{if } r_i \text{ belongs to the selected revision } j \text{ or to a predecessor.} \\ false & \text{otherwise.} \end{cases}
\end{aligned}
\tag{9.6}
$$

Revision Ambitions. In contrast to choices, ambitions in the revision space only consist of one bound option, namely a newly introduced option whose revision k is a successor of the *head*, i.e., the most recent revision h (regardless of which revision j was selected for check-out). As a consequence, within a *revision ambition* a_r, exactly one revision, r_k, occurs in a positive state. Bindings for further revision options are implicitly set to *undefined*.

$$a_r = \{(r_k, true)\}, \quad \text{revision } k \text{ is a successor of head revision } h. \tag{9.7}$$

Due to the mechanisms explained subsequently, this results in comparably short visibilities when confining to the revision graph; the corresponding expressions $\hat{a}_r$ consist of one option reference to r_k only.

9.3.4 Example

Figure 9.9 introduces an example of a revision graph. At the beginning of the edit session, it is mapped to the following options, invariants, preferences, and defaults:

$$O_r = \{r_0, r_1, r_2, r_3\}$$

$$\mathcal{J}_r = (r_0) \wedge (r_1 \Rightarrow r_0) \wedge (r_2 \Rightarrow r_1) \wedge (r_3 \Rightarrow r_2)$$

$$\mathcal{P}_r = \{(r_0, r_1), (r_1, r_2), (r_2, r_3)\}$$

$$\mathcal{D}_r = \{(r_1, false), (r_2, false), (r_3, false)\}$$

The user now decides to use revision 2 as starting point for modifications in the workspace. Based upon this selection, the original revision choice would correspond to:

$$c_r = \{(r_2, true)\}$$

Thereafter, preferences, involving the selection of predecessor revision options of r_2, are transparently applied:

$${}^{\mathcal{P}}c_r = \{(r_2, true), (r_1, true), (r_0, true)\}$$

At this point in time, a binding for option r_3 is missing. This is inferred from the defaults, which produce the final revision choice:

$${}^{\mathcal{PD}}c_r = \{(r_2, true), (r_1, true), (r_0, true), (r_3, false)\}$$

As soon as the changes are committed to the repository, a new revision is introduced transparently; its number 4 is generated automatically. The revision supersedes the former head revision 3. When referring to the structural design, an instance of Revision is created.

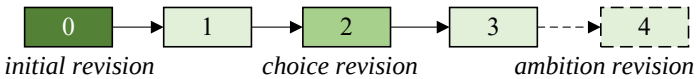


Figure 9.9: Example revision graph.

A commit message is assigned based upon user inputs. Triggered by the mapping rules, an option r_4 is introduced to O_r, the invariant $(r_4 \Rightarrow r_3)$ is appended to $\mathcal{J}_r$; $\mathcal{P}_r$ is extended by (r_3, r_4), and $\mathcal{D}_r$ by $(r_4, false)$, respectively.

9.4 Mapping Feature Models to the Version Space Base Layer

Being based on UVM, the base layer of the framework is natively applicable to *intensional versioning*. For reasons explained above, we rely on *feature models* as a higher-level representation of the *variant dimension* (see Section 2.3.2). To this end, a feature metamodel is defined in this section. In the literature, e.g., [Bat05; Ap+13b], several mappings of feature models to propositional logical formula have been defined (see related work). Building upon the distinction between invariants, preferences, and defaults presented above, we introduce a mapping that is in line with the common understanding of feature model semantics. In Section 10.7, (multi-revision) feature models are revisited in their role of an additional product space dimension that is subject to historical evolution.

9.4.1 Structural Design

For the subsequent explanations, we assume the feature modeling constructs informally introduced in Section 5.2. Figure 9.10 provides a view on the feature metamodel. By extending the core concept VersionDimension, feature models provide a second version dimension orthogonal to revision graphs.

Each feature model contains a dedicated root feature. Features carry a unique name, and they may or may not be mandatory. Furthermore, to maintain the feature model's history

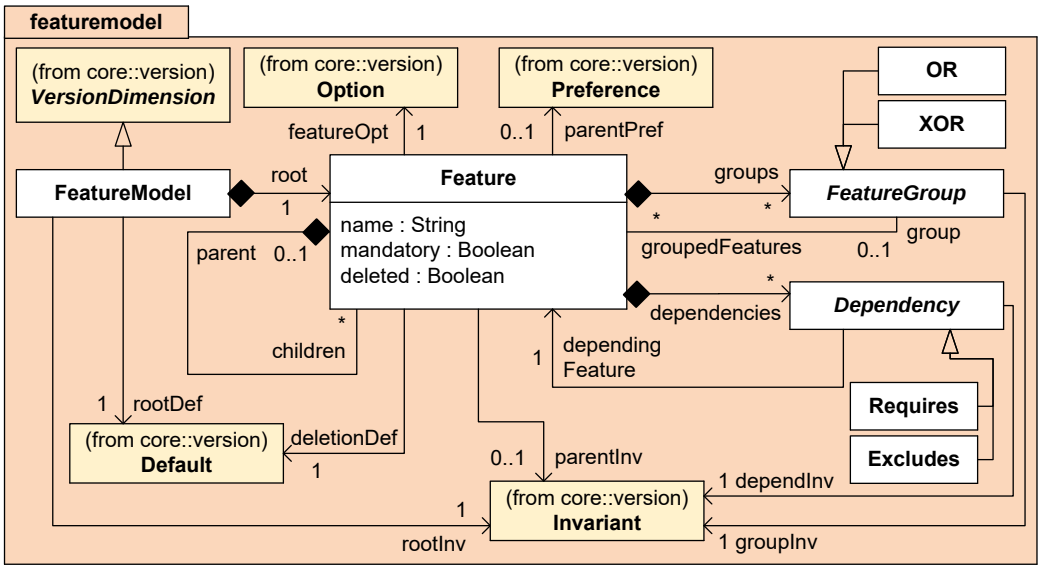


Figure 9.10: Ecore class diagram for the metamodel of feature models. Based on [Schwä+16, Figure 8].

available, instances of Feature are never physically deleted; rather, a deleted flag is activated that removes it from the user's display. [6]

The hierarchical composition of features is expressed by instantiating the reference children, such that each feature except for the root has a dedicated parent feature. Children of the same parent may be additionally organized in FeatureGroups. To this end, two group types, OR and XOR, are available.

In addition, cross-tree dependencies, namely Requires and Excludes, may be instantiated. They are contained by their source and cross-reference their target (dependingFeature).

Like the revision graph metamodel, the feature metamodel contains several references to core version space concepts, implementing the dynamic semantics of feature models in terms of propositional logic. It is assumed that these elements are created and maintained transparently according to the mapping rules introduced below.

9.4.2 Formal Mapping

The formal mapping between feature model instances and specialized rule base concepts is summarized in Table 9.2. Being the core configuration concept in SPLE, it is natural to map features to logical options. Deleted features are no more available for selection, but their options may still occur within visibilities; therefore, a default ensures that deleted features are automatically deselected. Furthermore, the selection of the root feature is always mandatory in a feature configuration. To this end, an invariant and a default are maintained, which ensure and enforce root selection, respectively.

The way how parent/child relationships are mapped to the rule base depends on whether the child is optional or mandatory. Optional features may or may not be selected in case the parent is selected, whereas for mandatory children, the selection state must coincide

Table 9.2: Mapping feature models to low-level rule base elements. Based on [Schwä+16, Table 1].

Pattern	Transformation		Metamodel
feature A	option	f_A	featureOpt
deleted feature D	default	$(f_D, false)$	deletionDef
root feature R	invariant	f_R	rootInv
	default	$(f_R, true)$	rootDef
optional child C of parent P	invariant	$f_C \Rightarrow f_P$	parentInv
mandatory child C of P	invariant	$f_C \Leftrightarrow f_P$	parentInv
	preference	(f_C, f_P)	parentPref
OR group with members $M_1, \ldots, M_n$ below feature F	invariant	$f_{M_1} \vee \ldots \vee f_{M_n} \Leftrightarrow f_F$	groupInv
XOR group with members $M_1, \ldots, M_n$ below feature F	invariant	$(f_{M_1} \vee \ldots \vee f_{M_n} \Leftrightarrow f_F) \wedge \bigwedge_{i<j\leq n} \neg(f_{M_i} \wedge f_{M_j})$	groupInv
dependency A *excludes* B	invariant	$\neg(f_A \wedge f_B)$	dependInv
dependency A *requires* B	invariant	$f_A \Rightarrow f_B$	dependInv

6 Details of the feature deletion operator are elaborated in Section 11.3.2.

with the parent. Configuration is supported by the preference (f_C, f_P), which involves a selection of all mandatory children as soon as a feature is selected by the user.

The formal mapping also implements the individual semantics of feature groups. In the case of an OR group, at least one of the grouped features must be selected if and only if the grouping feature is selected. For XOR groups, it is additionally required that at most one grouped feature is selected, such that each pair of grouped features is mutually exclusive.

Last, *requires* and *excludes* dependencies are mapped to invariants straightforwardly by combining the features with suitable propositional logical operators.

Below, we use O_f, $\mathcal{J}_f$, $\mathcal{P}_f$, and $\mathcal{D}_f$ to explicitly refer to the options, invariants, preferences, and defaults inferred from a feature model instance.

9.4.3 Version Selection

Feature Choices. As explained in Section 9.1.1, the specification of a *choice* in a feature model comprises an unambiguous selection in the form of a *feature configuration*. Without loss of generality, the framework assumes that the feature model is configured in a top-down way, beginning with the feature model root. Furthermore, after each selection made by the user, preferences and defaults are applied to all child features. A configuration obtained this way is effectively converted to a low-level choice as follows:

$$
\begin{aligned}
{}^{\mathcal{PD}}c_f &= \{(f_1, b_1), \ldots, (f_n, b_n)\},\ f_i \in O_f,\ i \in \{1, \ldots, n\}, \\
b_i &= \begin{cases} true & \text{if } f_i \text{ belongs to a selected feature } i \\ false & \text{if } f_i \text{ belongs to a deselected feature } i \\ false & \text{if } f_i \text{ belongs to a deleted feature } i \end{cases}
\end{aligned}
\tag{9.8}
$$

Feature Ambitions. It has been anticipated that *feature ambitions* are partial selections in a feature model, which leave features, namely those to which the change is immaterial, unbound (*neutral* selection state). Feature ambitions are converted to option bindings:

$$
\begin{aligned}
a_f &= \{(f_1, b_1), \ldots, (f_n, b_n)\},\ r_i \in O_r,\ i \in \{1, \ldots, n\}, \\
b_i &= \begin{cases} true & \text{if } f_i \text{ belongs to a selected feature } i \\ false & \text{if } f_i \text{ belongs to a deselected feature } i \\ undefined & \text{if } f_i \text{ belongs to a neutral feature } i \\ undefined & \text{if } f_i \text{ belongs to a deleted feature } i \end{cases}
\end{aligned}
\tag{9.9}
$$

9.4.4 Example

The left part of Figure 9.11 depicts a cut-out of the feature model of the running Graph example in the previously introduced concrete graphical syntax. The right hand side of the figure shows the corresponding internal representation (abstract syntax) as an object diagram conforming to the metamodel for feature models shown in Figure 9.10. Notice that it contains an additional object for a deleted feature X, which is hidden from the user's display in the graphically represented feature model.

For compactness, references to rule base elements are faded out in the object diagram. Given the mapping rules defined in Table 9.2, the following options, invariants, preferences,

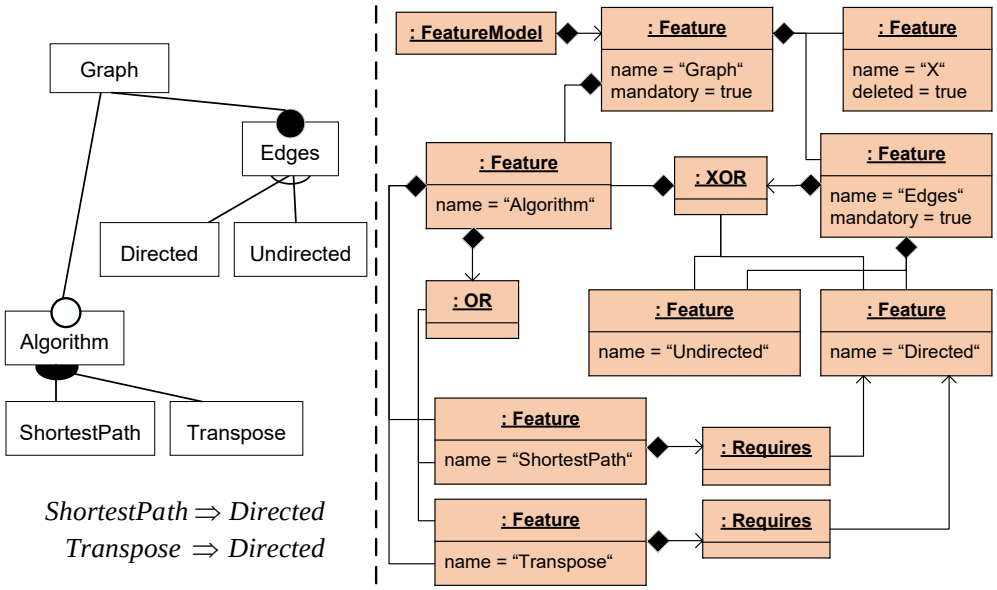


Figure 9.11: Example feature model in concrete and abstract syntax.

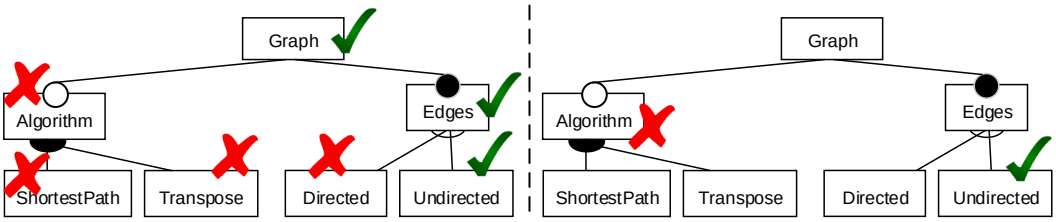


Figure 9.12: Example feature choice and feature ambition.

and defaults are derived (we use the initial letters to address feature names):

$$O_f = \{f_G, f_E, f_D, f_U, f_A, f_S, f_T, f_X\}$$

$$\mathcal{J}_f = (f_G) \wedge (f_G \Leftrightarrow f_E) \wedge (f_D \Rightarrow f_E) \wedge (f_U \Rightarrow f_E) \wedge (f_X \Rightarrow f_G) \wedge \big((f_D \vee f_U \Leftrightarrow f_E) \wedge \neg(f_D \wedge f_U)\big) \wedge (f_A \Rightarrow f_G) \wedge (f_S \Rightarrow f_A) \wedge (f_T \Rightarrow f_A) \wedge (f_S \vee f_T \Leftrightarrow f_A)$$

$$\mathcal{P}_f = \{(f_G, f_E)\}$$

$$\mathcal{D}_f = \{(f_G, true), (f_X, false)\}$$

In Figure 9.12, a feature choice and feature ambition, both basing on the feature model introduced above, are shown. The version selections are mapped to these bindings:

$$c_f = \{(f_G, true), (f_E, true), (f_D, false), (f_U, true), (f_A, false), (f_S, false), (f_T, false)(f_X, false)\}$$

$$a_f = \{(f_U, true), (f_A, false)\}$$

9.5 Preliminary Editing Model

After having explained in detail the mapping of revision graphs and feature models to the low-level version space defined in the core metamodel, we combine these building blocks into an initial *editing model*. After its semi-formal definition in Section 9.5.1, an example is given in Section 9.5.2. A formally complete description of the conceptual framework's *consistency-preserving dynamic filtered editing model* is contributed in Chapter 11.

9.5.1 Semi-Formal Definition

Check-Out. The check-out operation is used to specify a unique version (i.e., revision and variant) of the domain model, and a unique revision of the feature model, which both populate the workspace.

1. The user selects a revision j from the revision graph. After applying the automatic completions presented in (**9.6**) a *revision choice* ${}^{\mathcal{PD}}c_r$ is derived.
2. The multi-version feature model is *filtered* by ${}^{\mathcal{PD}}c_r$. The filtered feature model is exported into the workspace. [7]
3. The user specifies a *feature configuration* in the filtered feature model. From this, a *feature choice* ${}^{\mathcal{PD}}c_f$ is derived as specified by (**9.8**).
4. The *effective choice* c is calculated as the union ${}^{\mathcal{PD}}c = {}^{\mathcal{PD}}c_r \cup {}^{\mathcal{PD}}c_f$.
5. The multi-version domain model is *filtered* by the effective choice ${}^{\mathcal{PD}}c$ and then exported into the workspace.
6. The choice ${}^{\mathcal{PD}}c_f$ is memorized in order to reproduce the checked-out variant of the workspace during the subsequent commit.

Modify. In the workspace, the user applies modifications to the filtered domain model and/or to the filtered feature model.

Commit. On commit, the performed modifications are written back to the multi-version product space part of the repository. All changes shall be made conditional to a newly created revision. Furthermore, changes to the domain model are scoped by a *feature ambition*, such that they become visible only in affected variants.

1. The head revision h is retrieved. The *head choice* ${}^{\mathcal{PD}}c_h$ is obtained according to (**9.6**). The updated choice ${}^{\mathcal{PD}}c$ is defined as follows based on the memorized feature choice: ${}^{\mathcal{PD}}c := {}^{\mathcal{PD}}c_h \cup {}^{\mathcal{PD}}c_f$. The *most recent states* of feature model and domain model are reproduced by applying ${}^{\mathcal{PD}}c_h$ and ${}^{\mathcal{PD}}c$ to the multi-revision feature model and to the multi-version domain model, respectively.
2. By comparing the current workspace contents with the most recent states, differences are deduced as lists of inserted elements $e_{ins} \in E_{ins}$ and deleted elements $e_{del} \in E6_{del}$.

7 The operation *filter* as well as the multi-version representation of the feature model are precisely explained in Chapter 10.

3. The revision graph is managed automatically: A new revision k is introduced with user-specified details (i.e., commit message) and added as successor of the head revision h. A corresponding revision option r_k is added to O_r. Furthermore, a default, invariant, and preference are introduced as defined in Table 9.1.
4. The *revision ambition* is automatically set: $a_r = \{(r_k, true)\}$ (cf. (**9.7**)).
5. The user specifies a *feature ambition* that delineates the scope of the change to the domain model. From this, an option binding a_f is derived according to (**9.9**).
6. The applied modifications are written back to the multi-version representations of the product space under suitable ambitions. For changes to the feature model, $\hat{a} := r_k$; for changes to the product space, $\hat{a} := r_k \wedge \hat{a}_f$. Each modified (inserted or deleted) element is processed as follows (cf. Section 8.4.2):
   - Inserted elements e_{ins} are appended to the respective product space (domain or feature model) of the repository. Their visibility is set to: $v_{ins} := \hat{a}$.
   - Deleted elements e_{del} remain in the multi-version product space. Their visibility is set to $v_{del} := v_{old} \wedge \neg\hat{a}$.

9.5.2 Example

We illustrate the editing model by means of an example that refers to an evolving product line of *flow diagrams*, directed graphs with a dedicated start node (no incoming, one outgoing control flow, cardinality 0/1), multiple activity nodes (1/1), binary decision nodes (1/2), join nodes (+/1), and end nodes (1/0). Start and end nodes are represented by rounded rectangles, decision nodes by diamonds, and join nodes by circles, respectively.

The product line is developed in subsequent steps: In revision 1, the product space is initialized by performing a universal change that only affects the revision space. Next, two independent changes are applied that correspond to two mutually exclusive features. In the last revision, a change is re-assigned to a new feature, mapping an evolutionary increment to a logical feature retrospectively. [8]

Initialization. In the beginning, the version space consists of a revision graph that only contains the initial revision 0 and a feature model with a mandatory root feature R. The product space consists of an empty flow diagram. From the rules defined in Tables 9.1 and 9.2, the following low-level options, invariants, preferences, and defaults are derived [9]:

$$O^0 := O^0_r \cup O^0_f = \{r_0\} \cup \{f_R\}$$

$$\mathcal{J}^0 := (\mathcal{J}^0_r) \wedge (\mathcal{J}^0_f) = (r_0) \wedge (f_R)$$

$$\mathcal{P}^0 := \emptyset$$

$$\mathcal{D}^0 := \mathcal{D}^0_r \cup \mathcal{D}^0_f = \emptyset \cup \{(f_R, true)\}$$

8 When compared to the original version of the example, presented in [Schwä+15], the employed feature ambitions are slightly modified here, resulting in less complex visibilities.

9 We use superscripts in order to delineate different historical versions of the option set and the rule base, as well as the choices and ambitions used in each step.

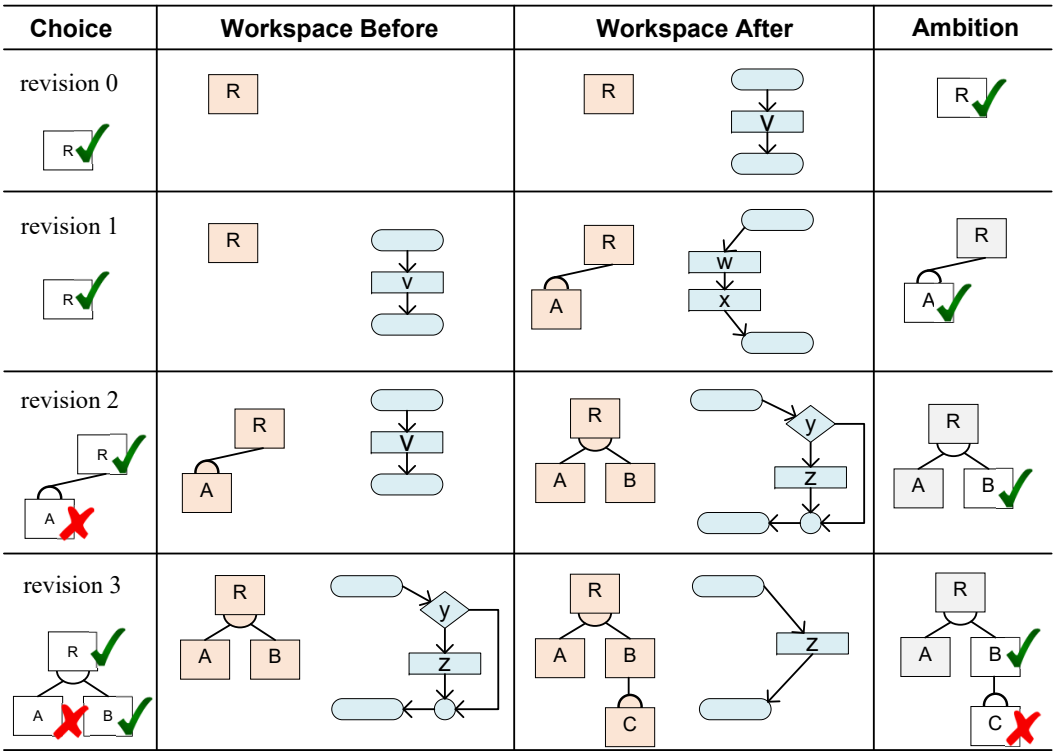


Figure 9.13: Specified choices and ambitions as well as modifications performed during subsequent iterations. Based on [Schwä+15, Figure 6].

Step 1. In order to populate the empty workspace, a choice must be specified. At the moment, there is only one choice allowed, including revision 0 as well as a feature configuration where the mandatory root feature R is selected. From this version selection, the following choice is derived:

$$^{\mathcal{PD}}c^1 := \{(r_0, true), (f_R, true)\}$$

Figure 9.13, step 1, shows the modifications applied: the insertions of a start node, an activity node v and an end node, as well as flows between those. The change is committed as a new revision 1. In the feature ambition, R is selected in order to explicitly connect this change to the root of the product line. This results in the following effective ambition:

$$a^1 := \{(r_1, true), (f_R, true)\}$$

As a side effect, the following option, invariant, preference, and default are added to the low-level option set O_r and rule base $\mathcal{R}_r$ transparently:

$$O_r^1 := O_r^0 \cup \{r_1\}$$

$$\mathcal{J}_r^1 := \mathcal{J}_r^0 \wedge (r_1 \Rightarrow r_0)$$

$$\mathcal{P}_r^1 := \mathcal{P}_r^0 \cup \{(r_0, r_1)\}$$
$$\mathcal{D}_r^1 := \mathcal{D}_r^0 \cup \{(r_1, false)\}$$

Analogous constraints are inserted for subsequent revisions; they are omitted below in favor of a better readability. Moreover, in this iteration, the low-level rule base of the unmodified feature model remains unchanged.

Step 2. As there is no variability defined in the feature model yet (since the selection of f_R is mandatory), it suffices to select a revision. A selection of revision 1 generates the choice

$${}^{\mathcal{PD}}c^2 := \{(r_0, true), (r_1, true)(f_R, true)\}$$

In the checked-out workspace, v is deleted and a sequence of activity nodes consisting of w and x is inserted (cf. Figure 9.13, step 2). Furthermore, an optional feature A is introduced below R. The following option set and the invariants are derived from this modification:

$$O_f^2 := O_f^0 \cup \{f_A\}$$
$$\mathcal{J}_f^2 := \mathcal{J}_f^0 \wedge (f_A \Rightarrow f_R)$$

For the commit, we specify a feature ambition with A selected, resulting in the following ambition:

$$a^2 := \{(r_2, true), (f_A, true)\}$$

Step 3. Once again, the latest revision is chosen. In the feature space, A is deselected, which generates

$${}^{\mathcal{PD}}c^3 := (r_0, true), (r_1, true), (r_2, true), (f_R, true), (f_A, false)\}$$

Thus, the checked-out domain model version is not affected by the deletion of v in step 2. As shown in Figure 9.13, step 3, an additional feature B is introduced, which excludes A due to an XOR group added above both features. This results in the following derived modifications to elements of the version space base layer:

$$O_f^3 := O_f^2 \cup \{f_B\}$$
$$\mathcal{J}_f^3 := \mathcal{J}_f^2 \wedge (f_B \Rightarrow f_R) \wedge \big((f_R \Leftrightarrow f_A \vee f_B) \wedge \neg(f_A \wedge f_B)\big)$$

Within the workspace, we perform a corresponding realization: the replacement of v by a new activity node z guarded by a conditional node y. Finally, we commit the change under a feature ambition in which B is selected:

$$a^3 := \{(r_3, true), (f_B, true)\}$$

Step 4. After revision 3 has been committed, we come to the conclusion that the guard y should not realize feature B but rather an optional child feature C. This retrospective feature assignment can be performed by checking out the latest revision with feature

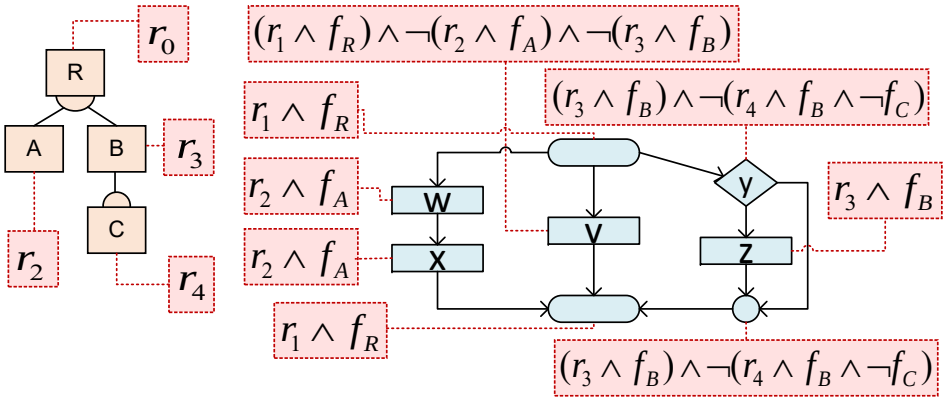


Figure 9.14: The transparent multi-variant product space of the example.

B included:

$$^{\mathcal{PD}}c^4 := \{(r_0, true), \ldots (r_3, true), (f_R, true), (f_A, false), (f_B, true)\}$$

Then, a feature C is introduced as an optional child of B:

$$O_f^4 := O_f^3 \cup \{f_C\}$$

$$\mathcal{J}_f^4 := \mathcal{J}_f^3 \wedge (f_C \Rightarrow f_B)$$

In the domain model made available in the workspace, the guard as well as the join node are deleted (cf. Figure 9.13, step 4). The domain model change is associated with a selection of B and a deselection of C:

$$a^4 := \{(r_4, true), (f_B, true)(f_C, false)\}$$

As a consequence, since revision 4, the presence of guard y is restricted to product line instances that include C. In case revision 3 is restored, however, y's logical visibility still depends on B only.

Inspecting the Repository. Taking the consecutive steps of the example together, we now investigate the user-invisible contents of the repository. Apart from the revision graph, which is merely a sequence connecting revisions 0 until 4 in this example, the repository contains a transparent product space consisting of feature model and domain model. Figure 9.14 depicts the product space in a superimposition representation, including visibilities organized by the filtered editing model. The fact that the feature model evolves only along the historical dimension is reflected by visibilities of features being composed of revision options only, whereas domain model elements' visibilities reference both revision and feature options. Visibility updates caused by deletions, affecting elements such as v or y, are characterized by multiple revision options occurring inside an expression.

9.6 The Change Space and its Mapping to the Base Layer

One of the key characteristics of the conceptual framework is the automated management of visibilities that determine version membership of product space elements. The mechanism of writing back changes using a global ambition (per product space dimension) results, however, in option expressions corresponding to the ambition $\hat{a}$ repeatedly appearing in the visibility of all affected elements. These expressions, on the one hand, increase the size of the repository. On the other hand, in case the user requests to revise an ambition retrospectively (see AMEND operation in Section 11.6.4), corrections of derived option expressions would be necessary in several places.

To this end, we introduce as optimization the *change space*, which is internally managed as a third version dimension in addition to the revision graph and the feature model. In contrast to the other dimensions, the change space is entirely transparent to the end user.

Altogether, the revision graph, the feature model, and the change space disjointly divide up the option set and the rule base, i.e., invariants, preferences and defaults:

$$O = O_f \dot{\cup} O_r \dot{\cup} O_\Delta \tag{9.10}$$

$$\mathcal{R} = \mathcal{R}_f \dot{\cup} \mathcal{R}_r \dot{\cup} \mathcal{R}_\Delta = \big((\mathcal{J}_f \wedge \mathcal{J}_r \wedge \mathcal{J}_\Delta), (\mathcal{P}_f \dot{\cup} \mathcal{P}_r \dot{\cup} \mathcal{P}_\Delta), (\mathcal{D}_f \dot{\cup} \mathcal{D}_r \dot{\cup} \mathcal{D}_\Delta)\big) \tag{9.11}$$

9.6.1 Structural Design

In the repository, the change space is represented as an instance of the metamodel depicted as class diagram in Figure 9.15. The change space organizes a sequence of change sets – representing commit actions triggered by the user –, which in turn are composed of several changes that refer to one product dimension each and that are scoped with an individual ambition. Each change is mapped to a triple of option, invariant, and preference based on the mapping described below.

In general, the change space may abstract from arbitrarily composed version spaces. When assuming the concrete three-layered repository architecture presented so far, each change set would consist of two changes, one referring to the feature model dimension and

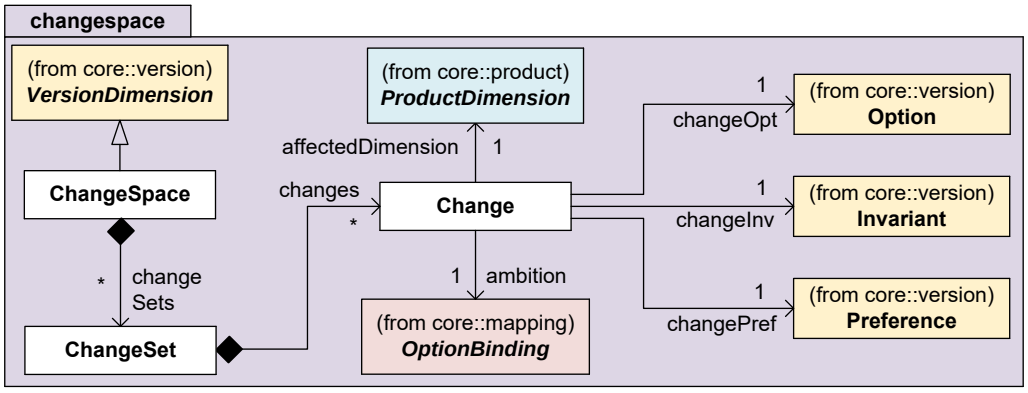


Figure 9.15: Metamodel for the transparent change space.

Table 9.3: Mapping concepts of the change space to low-level rule base elements.

Pattern		Transformation	Metamodel
change to the feature model	option	Δ_f	changeOpt
for revision k	invariant	$\Delta_f \Leftrightarrow r_k$	changeInv
	preference	(Δ_f, r_k)	changePref
change to the domain model	option	Δ_d	changeOpt
for revision k	invariant	$\Delta_d \Leftrightarrow r_k \wedge \hat{a_f}$	changeInv
under feature ambition a_f	preference	$(\Delta_d, r_k \wedge \hat{a_f})$	changePref

carrying an ambition that refers to the newly introduced revision, and another one referring to the domain model and associating the effective ambition.

9.6.2 Formal Mapping

Rather than referring to metaphors exposed to the user – i.e., revision graphs and feature models – the formal mapping for the change space refers to modifications (i.e., insertions and deletions) detected in a specific product dimension. A *write set* connects a difference to a user-specified ambition. [10]

Table 9.3 specifies how write sets are mapped to changes. Since the decision whether or not a change set shall be applied is boolean, it is adequate to map each write set to a change option Δ_i. Furthermore, each change abstracts from a user-specified, potentially multi-dimensional ambition a_i, such that preferences of the form (Δ_i, a_i) ensure that the change is activated if and only if the ambition includes the user-selected choice. Specific invariants ensure consistency of change option and ambition.

9.6.3 Version Selection

When seen from the user's perspective, version selection is not affected by the change space optimization. Behind the scenes, the transparent mapping to low-level choices and ambitions is altered as follows.

Change Choices. First of all, the user is asked for a combined choice in the revision graph and in the feature model as usual, and preferences and defaults are applied. We refer to this choice as ${}^{\mathcal{PD}}c' := {}^{\mathcal{PD}}c_r \cup {}^{\mathcal{PD}}c_f$. From this choice, a *change choice* ${}^{\mathcal{PD}}c_\Delta$ is derived automatically by applying the preferences derived from write sets:

$$\begin{aligned} {}^{\mathcal{PD}}c_\Delta &= \{(\Delta_1, b_1), \ldots, (\Delta_n, b_n)\},\ \Delta_i \in O_\Delta,\ i \in \{1, \ldots, n\}, \\ b_i &= \hat{a}_i({}^{\mathcal{PD}}c'),\ a_i \text{ is the ambition assigned to the change mapped by } \Delta_i \end{aligned} \tag{9.12}$$

10 A precise definition of write sets is given in Section 10.8.1 in the context of product-level difference computation.

Thus, the choice is applied to the conjunctive representation of the change ambitions. [11]

The *effective choice* ${}^{\mathcal{PD}}c$, which is used for filtering the product space, is unionized as follows:

$$ {}^{\mathcal{PD}}c = {}^{\mathcal{PD}}c_r \cup {}^{\mathcal{PD}}c_f \cup {}^{\mathcal{PD}}c_\Delta \tag{9.13} $$

Change Ambitions. After an ambition $a' := \{(r_k, true)\} \cup a_f$ has been specified by the user, rather than using r_k for visibility updates to the feature model, the modifications are persisted as follows:

1. A new change option Δ_f is introduced to O_Δ.
2. The invariant $\Delta_f \Leftrightarrow r_k$ is added to $\mathcal{J}_\Delta$.
3. The preference (Δ_f, r_k) is added to $\mathcal{P}_\Delta$.
4. The change to the feature model is committed under the change ambition $a_{\Delta_f} := \{(\Delta_f, true)\}$.

In analogy, for visibility updates to the domain model, a' is replaced by a change option:

1. A new change option Δ_d is introduced to O_Δ.
2. The invariant $\Delta_d \Leftrightarrow \hat{a}'$ is added to $\mathcal{J}_\Delta$.
3. The preference $(\Delta_d, \hat{a}')$ is added to $\mathcal{P}_\Delta$.
4. The change to the domain model is committed under the change ambition $a_{\Delta_d} := \{(\Delta_d, true)\}$.

Table 9.4: Low-level change space elements by the flow chart example.

Rev.	Dimension	Option	Invariant	Preference
0	feature	Δ_{0f}	$\Delta_{0f} \Leftrightarrow r_0$	(Δ_{0f}, r_0)
0	domain	Δ_{0d}	$\Delta_{0d} \Leftrightarrow r_0$	(Δ_{0d}, r_0)
1	feature	Δ_{1f}	$\Delta_{1f} \Leftrightarrow r_1$	(Δ_{1f}, r_1)
1	domain	Δ_{1d}	$\Delta_{1d} \Leftrightarrow (r_1 \wedge f_R)$	$(\Delta_{1d}, (r_1 \wedge f_R))$
2	feature	Δ_{2f}	$\Delta_{2f} \Leftrightarrow r_2$	(Δ_{2f}, r_2)
2	domain	Δ_{2d}	$\Delta_{2d} \Leftrightarrow (r_2 \wedge f_A)$	$(\Delta_{2d}, (r_2 \wedge f_A))$
3	feature	Δ_{3f}	$\Delta_{3f} \Leftrightarrow r_3$	(Δ_{3f}, r_3)
3	domain	Δ_{3d}	$\Delta_{3d} \Leftrightarrow (r_3 \wedge f_B)$	$(\Delta_{3d}, (r_3 \wedge f_B))$
4	feature	Δ_{4f}	$\Delta_{4f} \Leftrightarrow r_4$	(Δ_{4f}, r_4)
4	domain	Δ_{4d}	$\Delta_{4d} \Leftrightarrow (r_4 \wedge f_B \wedge \neg f_C)$	$(\Delta_{4d}, (r_4 \wedge f_B \wedge \neg f_C))$

11 With the same semantics but with a higher computational complexity involved, we could have written $\hat{c}' \Rightarrow \hat{a}_i$.

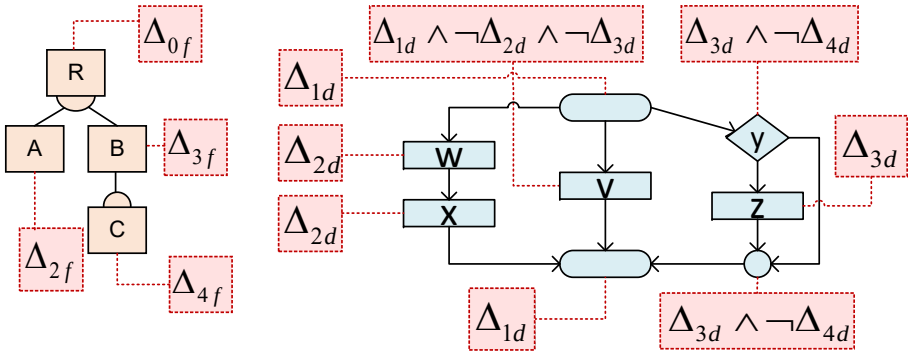


Figure 9.16: The multi-variant domain model of the flow chart example with change space optimization enabled.

Efficiency Estimation. The visibilities created by repeated application of this commit strategy correspond to logical conjunctions of delta options, which may occur in positive or negated form. By this optimization, visibilities grow linearly with the number of changes applied to the respective elements, and in particular, do not depend on the complexity of the ambition expression derived from selections in the feature model. Therefore, the complexity is bound to $\mathcal{O}(|a|)$ (where $|a|$ denotes the number of bindings in the ambition), which is in most cases below the complexity of the unoptimized strategy, $\mathcal{O}(|E| \cdot |v_{avg}|)$ (where $|E|$ denotes the number of modified elements and $|v_{avg}|$ refers to the average complexity of the visibilities before commit).

9.6.4 Example

The change space optimization is illustrated by the example introduced above in Section 9.5.2. We assume that the same version history is replayed, such that the user does not observe any change in behavior. Transparently, however, the change space manages new version space elements. Table 9.4 lists the options, invariants, and preferences transparently introduced before committing the change during subsequent commit operations.

Furthermore, as visualized in Figure 9.16, the transparently organized visibilities differ inasmuch as they exclusively refer to change options. In particular, multiple occurrences of option expression terms such as $r_4 \wedge f_B \wedge \neg f_C$ are condensed into single option references, e.g., Δ_{4d}. Particularly when considering larger change sets, this saves both memory when storing the superimposition as well as run-time when evaluating option expressions.

9.7 Visibility Forest

A similar yet orthogonal optimization was mentioned in Section 9.2.3, where the metamodel for option expressions has been explained. According to this, the *visibility forest* of a repository is a global data structure for visibilities, avoiding repeated duplication. A further practical purpose of visibility forests is the simplified representation of *merged visibilities*, which comes into play as soon as different copies of a repository need to be synchronized; this is a subject of Section 12.4.4.

9.7.1 Structure and Functionality

The reuse of visibilities is mainly achieved by two design decisions reflected in the meta-model for feature expressions, whose class diagram is depicted in Figures 9.5 and 9.7. First, option expressions are not contained by their versioned elements but cross-referenced by those only. The container for option expressions is a global VisibilityForest. Second, although option expressions are essentially self-contained, the class OptionExprRef makes possible that a sub-expression is a reference to an existing expression managed by the same visibility forest. This way, identical sub-trees can be factored out.

A non-optimized visibility update strategy would behave as follows: Create a new feature expression by applying the visibility update rule (cf. Section 9.5.1) and insert it into the visibility forest. Then, reference it from the affected product space element. The strategy does not make use of expression references at all, such that the visibility forest degenerates into a flat collection of self-contained option expressions.

In contrast, the optimized strategy applies the following mode of procedure for a given non-empty write set:

1. At the beginning of the processing of each write set, create an option expression that represents the conjunctive form of the ambition $\hat{a}$ as a self-contained instance of OptionExpr. Insert it into the visibility forest.
2. In case the write set contains deletions, create and insert into the visibility forest an expression representing the negation of the ambition, $\neg\hat{a}$. Reuse the ambition expression node from step 1 using an OptionExprRef.
3. For each element insertion, reference the ambition expression (cf. step 1) from the inserted element.
4. For each element deletion, create a new And expression. As its first operand, insert an OptionExprRef that targets the old expression $\hat{a}_{old}$. As second operand, create an OptionExprRef to the expression created for the negated ambition $\neg\hat{a}$ (cf. step 2). Reference the And expression from the element affected by the deletion.
5. Avoid the repeated creation of identical And expressions in step 4. To this end, keep track of the old visibilities of processed deleted elements. If the same visibility (represented by the identical "old" forest node) was already processed, re-use the expression by referencing it from the deleted element. [12]

Efficiency Estimation. When considering only insertions, the size of a so created forest would be directly proportional to the number of commits (cf. steps 1 and 3 above). Depending on the original visibilities of affected elements, deletions may involve the creation of individual new visibilities, whose maximum number is bound to the number of deleted elements. Nevertheless, it must be taken into consideration that the created "new" visibilities consist of an And expression with two subordinate option expression references only. Altogether, the number of visibility nodes added to the forest is bound to $\mathcal{O}(|E_{del}|)$, whereas the unoptimized strategy exposes a complexity of $\mathcal{O}(|E| \cdot |a|)$.

12 This can be straightforwardly implemented with the help of an associative auxiliary data structure.

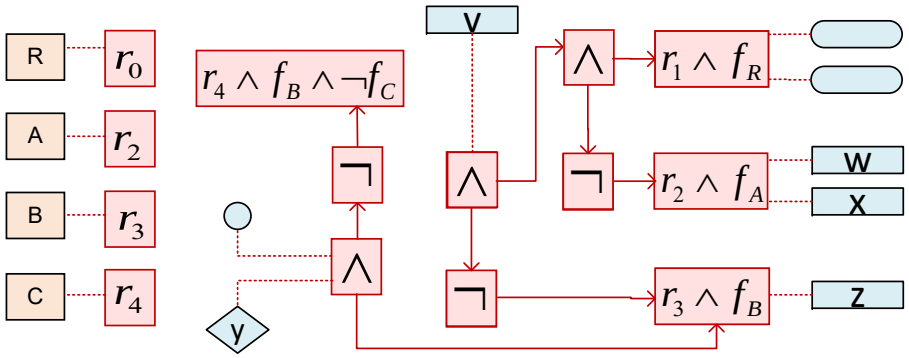


Figure 9.17: The multi-variant domain model of the flow chart example with visibility forest optimization enabled.

9.7.2 Example

The visibility forest optimization is also demonstrated by the flow chart example from Section 9.5.2. For a better comprehensibility, we fade out the orthogonal change space optimization here.

Figure 9.17 illustrates the optimized usage of the visibility forest. For clearness, particular details of the product space, e.g., flows, are omitted from the view. Each of the referenced visibilities (represented using solid boxes) is contained by the used instance of VisibilityForest. Dashed connectors indicate instances of the reference visibility, whereas solid arrows represent instances of OptionExprRef.

In multiple places, the same feature expression is referenced from different product space elements rather than occurring in multiple copies; e.g., nodes w and x, which are inserted in the same iteration under the same ambition, share an option expression as their visibility. The expression referenced by v was composed by applying the new strategy for element deletions twice. Furthermore, the same And expression is referenced by decision node y and its corresponding join node, which were both added and removed under the same logical and historical scope.

When compared to the unoptimized usage of the visibility forest (see Figure 9.14), the number of nodes contained in the forest is larger, however, their internal complexity is much lower. This advantage becomes more significant when increasing the number of elements affected in an iteration, or the complexity of ambitions used, or both.

9.8 Related Work

The conceptual framework has been influenced by several related theoretical approaches, first and foremost UVM and its precursors. We also present concepts related to the building blocks of the editing model, namely integrated repository architectures, the mapping of feature model to propositional logic, feature ambitions, as well as the optimizations change space and visibility forests.

Integrated Repository Architectures. The hybrid repository architecture with the feature model being part of both the version space and the product space is unique in the literature. Nevertheless, there exist a couple of related architectures that allow for versioned version control metadata in a similar way.

In [WMC01], a layered architecture for UVM-based software repositories is described. On the lowest level, the *instrumentable version engine* combines a version model and version rules (roughly corresponding to the option set and invariants presented here) with an abstract delta storage. On top of this, *transaction support* is added, which organizes the historical evolution of both versioned data and versioning metadata using a time-stamping mechanism.

As mentioned in Section 6.4.2, the VCS *Adele* [EC94] realizes an *asymmetric* architecture towards integrated versioning. On the base layer, the version model is versioned independently of the data model using directed deltas. The mapping between problem space and solution space is contained in the data model itself – but also has to be managed manually –, such that the necessity for versioned visibilities does not arise.

The feature logic approach implemented in the ICE system [ZS97] shares the property of the new conceptual framework according to which extensional versioning is realized on top of intensional versioning. As already pointed out in Section 6.4.4, *change features* represent historical evolution steps. A superordinate transaction protocol, however, does not exist, such that the management of the historical dimension is not automated to the extent that is usual in VCS.

DeltaEcore [SSA14b] is an integrated approach towards variability (in space) and evolution (referred to as variability in time). The connection of variable and evolving parts of the product line to the underlying product model is made by explicit directed deltas, which are divided up into *configuration deltas*, which are specific to the selection of a feature, and *evolution deltas*, which describe the product-level change between two historical revisions, respectively. Rather than adding the feature model to historical version control, *hyper feature models* (see Section 6.3.2) come into play. Consequently, the selection order between features and revisions is inverted when compared to our approach. For the dedicated end user, long sequences of revisions may become difficult to understand given the multitude of evolution deltas that must be kept in mind by the developer.

The approaches presented in [Fis+15; LELH16] manage a set of product variants without intrinsically representing their differences in a finer-grained way. In this way, they realize *intensional on top of extensional versioning*, which contrasts with the strategy applied here (see design decision **D10** on page 137).

Specialized Rule Base. In contrast to plain UVM (see Section 8.4), the *specialized rule base* presented in Section 9.2.2 defines a disjoint partition of version rules into *invariants*, *preferences*, and *defaults*. This distinction differs from [CW97] inasmuch as invariants are called *constraints*[13]. Furthermore, in [CW97], defaults are weaker than here, being "only applied when otherwise no unique selection could be performed".

In the context of *change-oriented versioning* (CoV), the terms have entirely different interpretations. *Preferences* are weights attached to options that are used for the heuristic inference of a choice from an incomplete specification. *Defaults* are bindings created from

13 Here, we intentionally use the term "constraint" for the editing model exclusively.

system-environment settings.

In [Mun+93], two additional related rule base concepts have been defined. First, a *stability* is a logical expression determining choices for which the associated versions cannot be changed on product level. Second, a *validity* determines which choices safely produce valid or completed versions. Both concepts may be used for defining and guaranteeing the consistency of released historical and logical versions.

Mapping Feature Models to Logic. The mapping of feature models to the specialized rule base provided in Section 9.4.2 can be generalized to a mapping to *propositional logical formulas* when confining to invariants. A so obtained mapping is equivalent to the semantic descriptions provided in [SHT06].

In [Bat05], an additional group type, AND, is introduced, and all features except for the root are forced to be contained in a group. Given that the feature model dialect used here can be straightforwardly translated into the format used in [Bat05], the resulting mappings to propositional logic are also equivalent.

In the formal introduction of feature models provided in [Ap+13b], requires/excludes relationships are generalized into arbitrary logical expressions; apart from this, the described mapping is semantically equivalent.

Feature Ambitions. A novel contribution is the concept of feature ambitions, partial selections in the feature model which describe the logical scope of a change.

Partial feature configurations have been used for diverse purposes in the related work. For instance, *staged feature configurations* [CHE04] are used for stepwise refinement during application engineering and require additional consistency properties such as parent-child co-selection. In contrast, feature ambitions as used here merely require consistency with the feature configuration defined as choice; see Chapter 11.

In the flow chart example, most ambitions are of type (b) when referring to Figure 9.2, i.e., one feature is selected positively. This matches well the paradigm of *feature-driven development* (FDD) [PF01], where each increment should strictly associate with one feature. Feature ambitions generalize from this by allowing for more complex scoping of changes such as feature interaction or product-specific changes. Several related approaches lie within the continuum between strictly FDD-like scope definition and feature ambitions. For instance, in *Feature Mapper* [HKW08], one or more features may be positively selected. Similarly, *views* are defined in CIDE [Käs+09].

Conversely, there exist related concepts having the same or different expressiveness than feature ambitions, but reside at a lower level of abstraction by exposing configuration options to the user. For instance, in UVM [WMC01], ambitions are represented as arbitrary conjunctions of feature options occurring in a positive or negated form. In [Stă+16], a tool relying on the partially filtered editing model by [WO14] is presented. On check-out, a view is defined using a textual representation of the *choice calculus* [EW11]. In contrast to feature ambitions, this formalism is situated at a lower level of abstraction but has a higher expressiveness being fully equivalent to propositional logical expressions.

Change Space and Visibility Forest. Both optimizing techniques presented above, the change space and the visibility forest, are based on the observation that conjunctive forms of ambitions repeatedly appear in the visibilities of elements added and removed in the same iteration. Similar solutions for related problems can be found in the literature.

As reported in [WMC01], the EPOS system, on top of which UVM had been prototypically implemented, provides a data structure called *visibility tree* [Mun96]. In contrast to the visibility forest presented here, these trees are not capable of reusing leaves by cross-references, which still may lead to repeated duplication of option expressions derived from the ambition. The strict tree structure may also become problematic when retrospectively modifying ambitions; in contrast, the *change space* optimization presented here guarantees that no ambition ever needs to be structurally copied internally.

9.9 Summary

The hybrid version model is at the heart of the new conceptual framework for the integration of version control and SPLE based on MDSE. The version space base layer is defined by means of an abstract metamodel and formal definitions for version definition and selection concepts, including options and version rules. Relying on propositional logic, the base layer demands for more abstract representations as soon as the user is involved. To this end, we have introduced a mapping to revision graphs (for historical, extensional versioning) and to feature models (for logical, intensional versioning). A preliminary editing model, which was only sketched superficially in this chapter, combines both version dimensions.

Coming back to the requirements established in Section 2.3, both the historical and the variant dimension are covered, such that **R1** and **R5** (*revision graphs* and *feature models*) as well as **R2** and **R6** (*extensional revision selection* and *intensional variant selection*) are satisfied by the individual version dimensions. As far as cross-cutting requirements are concerned, a *uniform mechanism* (**R15**), which is aware of the *overlap* between historical and logical versioning (**R14**), has been presented. The feature model may evolve in parallel with the domain model (**R13**).

The change space and the data structure of visibility forests are two disjoint optimizations serving a similar purpose—avoiding duplication of visibilities created from the same ambition. Despite their high similarity, it makes sense to use them both at a time, since they have individual advantages that the opposite does not have. The change space allows for retrospective amendments to ambitions, whereas visibility forests form a global data structure that allows to uniquely identify visibilities; this becomes crucial as soon as several copies of the repository (and the contained visibilities) need to be synchronized.

This chapter has been dedicated to the version management perspective on the conceptual framework. In Chapter 10, we move the focus to the internal product representation perspective and complete the presentation of the feature model's dual role. The preliminary editing model is revisited and formally specified in Chapter 11, making it consistency-preserving. Revision graphs, as introduced here in Section 9.3, are extended in such a way that they meet the requirements of multi-user operation in Chapter 12. Last, metadata management, which was anticipated in this chapter, is explained in Chapter 13.

Well before the seventies have run to completion, we shall be able to design and implement systems that are now straining our programming ability, at the expense of only a few percent in man-years of what they cost us now. [...] These systems will be virtually free of bugs.

EDSGER W. DIJKSTRA (1972)

Chapter 10

Extensible Extrinsic Product Model

Abstract

We move the focus from the version space to the product space, whose structure is also defined using the formalism of Ecore class diagrams. The product space base layer – which essentially arranges versioned elements that carry visibility annotations in a tree – is instantiated in multiple places. First, a general representation for any kind of ordered collection of product space elements is given. Then, a versioned representation for file hierarchies is introduced, before concrete specializations for files – text files and EMF model instances – are presented. Furthermore, a mapping between feature models and the product space base layer is explained in order to complete the explanation of its hybrid role in the conceptual framework. Generic algorithms are introduced for the operations of matching, differencing, and (asymmetric two-way raw) merging. These algorithms rely on element-specific sameness criterion definitions. Last, related concepts to extrinsic product space representation as well as approaches to feature model versioning are compared.

Contents

10.1 Characterization

In the previous chapter, we have explained *how* products are versioned—in an integrated way that takes into consideration both historical evolution and logical variation. This chapter is centered around the question *what* is versioned, and presents data structures and algorithms for the transparently versioned contents of the repository. The presented product model is *extensible* and *extrinsic*. These properties demand for further explanation.

Extensible. The framework makes only minimal assumptions about the contents of the repository. In particular, it is assumed that the versioned information can be decomposed into a tree of fine-grained elements, each capable of carrying its own visibility. This way, the described framework represents transparent annotative variability. These assumptions are reflected in the product space base layer, which in turn is instantiated by several product dimensions, e.g., versioned file hierarchies (consisting of text files and EMF resources) and feature models as assumed in the default architecture. The framework may be extended in order to support, e.g., diverse file content types or entirely different product space models such as databases.

Extrinsic. Inside the repository, the versioned contents are represented in a different way than they are in the workspace. This way, the multi-version artifacts need not comply to single-version well-formedness rules imposed by tools and interchange formats used in the workspace. In particular, multi-version EMF models may violate single-version metamodel rules, achieving *unconstrained variability* in the repository (see design decision **D6.1** on page 136). Workspace contents are presented *intrinsically* (**D6.2**), making the approach tool-independent.

Figure 10.1 illustrates the connection between extrinsic and intrinsic representation by an abstract and minimalistic example. The workspace, depicted on the right hand side, comprises a copy of the feature model as well as a file hierarchy that includes three nested folders and two files, a text file and a model file containing a simple state diagram. These are the contents available for editing during the MODIFY phase. On the left hand side, a corresponding repository-internal representation is shown. Every product dimension is essentially a tree of elements, each of which may or may not carry an individual visibility that refers to options of the superordinate version dimensions. The structure of the tree is defined by several *product space metamodels*, which are refined further below. Sequences, e.g., text files, are mapped to *directed graphs*. The figure contains several operations that constitute building blocks of higher-level commands such as check-out and commit: IMPORT and EXPORT convert between the intrinsic and extrinsic representation. MATCH, DIFF, and MERGE are defined extrinsically, such that they may be used for the comparison of workspace (after importing) and repository on the one hand, and of multiple copies of the repository in distributed versioning (see Chapter 12) on the other hand.

The remainder of this chapter is organized as follows: First, the product space base layer

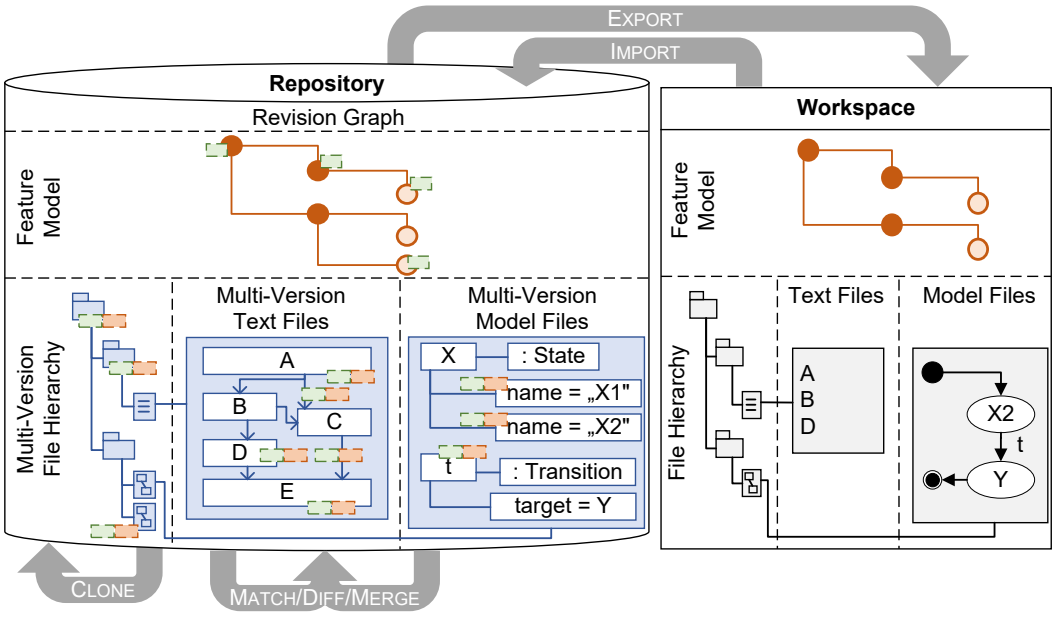


Figure 10.1: relation between the extrinsic product representation used in the repository and the intrinsic, workspace-level representation.

is formally introduced by means of a generic metamodel and several function definitions. Special emphasis is put on *sameness criteria*, which are used to match individual objects of two copies of a concrete product dimension. Moreover, a generic multi-version representation for *ordered collections* is discussed in Section 10.3. Next, two of those concrete dimensions are presented: a versioned file hierarchy – which may comprise both text files (cf. Section 10.5) and EMF model instances (10.6) –, and moreover, feature models in their role as an additional product dimension. To this end, the feature metamodel introduced in Section 9.4 is refined in Section 10.7. We return to the generic extrinsic representation in Section 10.8, where data structures and algorithms for matching, differencing, and merging diverged copies of a product space version are defined. Last, related work is presented.

10.2 Product Space Base Layer

Like the version space, the product space is organized along several dimensions, allowing for a flexible repository architecture. The structure of each dimension in turn is defined by a metamodel that extends the product space core metamodel. This in turn structurally defines the product space base layer.

10.2.1 Structural Design

Figure 10.2 depicts a class diagram that refines the package core::product declared in Figure 9.5. A product dimension contains a *tree* of *fine-grained* (**D18**) versioned elements, which in turn refer to a visibility from the visibility forest (not shown here, see Section 9.7). Versioned elements carry a *transaction number*, which is relevant for collaborative versioning; see Section 12.3.1.

Root elements are directly contained by the dimension, whereas non-root elements have a unique *parent* element. By *cross-links*, non-hierarchical dependency relationships such as applied occurrences are represented. Both kinds of relationships are redefined for all

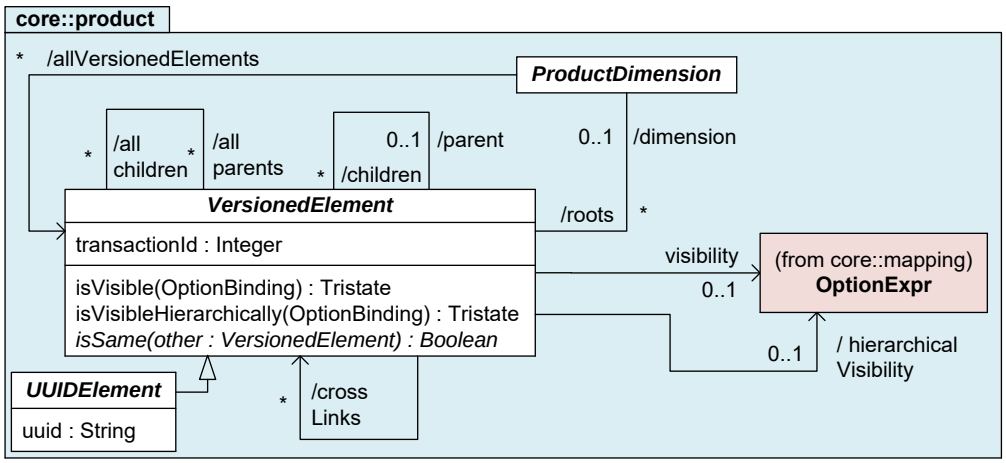


Figure 10.2: Ecore class diagram representing the product space core.

individual product elements; unless stated otherwise, children corresponds to the values of all containment, and crossLinks to non-containment references defined in instances of all specializations of this abstract product space metamodel.

The operation isVisible applies the given option binding to the referenced visibility and returns the result. The semantics of hierarchical visibility evaluation (see operation isVisibleHierarchically and derived reference hierarchicalVisibility) is provided below.

Last, the polymorphic operation isSame is accessed during the comparison of two copies of a product dimension. The operation is formally declared below, and its implementation depends on the specific subclass of VersionedElement. In many cases, artificial string-valued UUIDs are used here; for this purpose, the abstract base class UUIDElement is provided.

10.2.2 Operations on the Product Space

For our formal reasoning about the extrinsic representation of product dimensions, we assume that the elements arranged in the hierarchy are taken from a base set P (which corresponds to the values of the derived reference allVersionedElements):

$$P = \{e_1, \ldots, e_q\} \tag{10.1}$$

Furthermore, each versioned element is instance of a specific, non-abstract *class* that corresponds to the name of the respective subclass of VersionedElement in the structural view. This is here defined by the following function signature:

$$class : P \to \mathcal{C} \tag{10.2}$$

Here and below, $\mathcal{C}$ denotes the domain of non-abstract *classes* defined in the extrinsic repository metamodels.

The fact that the elements of the base set P are arranged in a tree is expressed by the following function, which reflects the reference parent in the structural view:

$$parent : P \to (P \cup \{\epsilon\}) \tag{10.3}$$

Root elements of a product dimension do not have a parent element; in this case, the function returns the null element ϵ.

In addition to parent/child relationship, we allow for *cross-links* between elements. In concrete instantiations of the core model, these represent applied occurrences of elements. The following relation is equivalent to the metamodel reference crossLinks:

$$crossLinks : P \to \mathcal{P}(P) \tag{10.4}$$

The functions $class$, $parent$, and $crossLinks$ are, on the low product space layer, version-insensitive. For specific version dimensions, in particular EMF models, however, versioning of containment relationships or cross-links becomes relevant. This is achieved by additional subclasses of VersionedElement organizing these links; see Section 10.6.1.

The notion of *visibility* must be understood by taking the semantics of the tree hierarchy into consideration. Since a child element existentially depends on its parent element, the

child's visibility must be "at least as restricted as" the parent's. It is this property that is additionally satisfied by the *hierarchical visibility* v_i^* of an element e_i:

$$v_i^* = \begin{cases} v_i & \text{if } parent(e_i) = \epsilon \\ v_i \wedge v_p^* & \text{otherwise, where } e_p = parent(e_i) \end{cases} \quad \textbf{(10.5)}$$

This definition avoids repeated copies of visibility terms of parent elements to children, such that visibility updates to the root of a sub-tree affect all direct and indirect contents. Also, operations on the visibilities of the product space may skip a sub-tree in case its root element need not be processed either.

According to the hierarchical (re-) definition of visibility, the new *filter* operator (previously introduced for UVM in Section 8.4) performs the following hierarchy-aware projection on a given element set E using the specified choice c:

$$E|_c^* = \{e_i \in E | v_i^*(c) = true\} \quad \textbf{(10.6)}$$

As mentioned before, these definitions hold only for the *extrinsic* product space representation within the repository. The *intrinsic* representation of workspace contents, however, is content-specific and therefore not covered by the base layer of the conceptual framework (except for ordered collections, which are formalized in Section 10.3.2). We assume generic operations to convert between the extrinsic and intrinsic representation:

- The operation IMPORT converts the currently available workspace contents into an element set E conforming to the extrinsic definition. It is part of the higher-level operation COMMIT.
- The operation EXPORT, which is invoked during CHECKOUT, takes as input an element set E – which is supposed to represent a single product version –, converts it into the corresponding intrinsic representation, and makes this available in the workspace. This involves dedicated well-formedness management, which is discussed in Chapter 13.

Last, operation CLONE creates an exact copy of an extrinsic element set E.

10.2.3 Sameness Criteria

During the operation COMMIT, it is necessary to *compare* two states of an (extrinsically represented) product space. To this end, in Section 10.8, we specify generic matching and differencing algorithms that rely on a *comparison-based* differentiation strategy (see design decision **D3**). These algorithms make use of a two-valued predicate $same$ (cf. polymorphic operation isSame in Figure 10.2), which states whether or not two versioned elements part of disjoint product spaces mutually correspond to each other.

$$same : P_1 \times P_2 \rightarrow \{true, false\},\ P_1 \cap P_2 = \emptyset \quad \textbf{(10.7)}$$

Sameness criteria are always applied locally in the hierarchy, such that a pair of elements may be the same only in case their parent elements are the same:

$$same(e_i, e_j) \Rightarrow same(parent(e_i), parent(e_j)) \vee parent(e_i) = parent(e_j) = \epsilon \quad \textbf{(10.8)}$$

Notice that the inverse implication is not required, such that the contents – i.e., children – of "same" elements may be mutually different.

The exact semantics of this predicate depends on the concrete class of elements. In the subsequent sections, we specify the predicate for concrete classes of the diverse product space dimensions utilizing the Object Constraint Language (OCL; see Section 3.4.2). The definitions assume that the elements to be compared are of the same class:

$$same(e_i, e_j) \Rightarrow class(e_i) = class(e_j) \tag{10.9}$$

In many cases, the sameness criterion relies on artificially generated, string-valued UUIDs. Correspondingly, for subclasses of UUIDElement, the following operation is defined:

$$uuid : P_{UUID} \rightarrow \mathcal{S}, \quad P_{UUID} \subseteq P \tag{10.10}$$

By $\mathcal{S}$, the domain of alphanumeric character strings is denoted.

10.3 Mapping Sequences to the Product Space Base Layer

After having explained the core mechanisms, we move onward to concrete product dimensions. As a first building block, *ordered collections* are presented here. Section 8.2.3 has described how version-aware ordered collections may be represented by *multi-version digraphs*. This idea is adapted to the product model below.

10.3.1 Structural Design

A metamodel for multi-version ordered collections is depicted as class diagram in Figure 10.3. An ordered collection is defined by means of a vertex set (vertices, corresponding to V in the formalization) and an edge set (edges, corresponding to E, respectively). Vertices represent the applied occurrence of an elsewhere defined versioned element (cf. occurringElement) at a specific relative position in the collection. Each edge connects a source to a target vertex. Both the occurrence of an element itself and the mutual order between occurrences are subject to version control. Last, we define that vertices carry UUIDs, which are, however, not persisted in the workspace.

Instances of this metamodel are used in specific product dimensions in several places where ordered collections must be versioned. An example referring to text files – ordered collections of lines – is provided in Section 10.5.3.

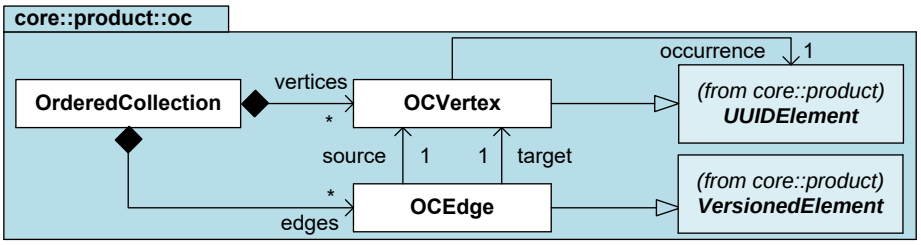


Figure 10.3: Metamodel for multi-version ordered collections.

10.3.2 Import and Export Transformations

Independently of concrete artifact types the user is confronted with in the workspace, we here specify in a generic form the IMPORT and EXPORT transformations, employed during commit and check-out, for ordered collections. Without loss of generality, we assume that the workspace contains *single-version sequences* (cf. Section 8.1.2), which are translated from and into *multi-version digraphs* (Section 8.2.3 and 8.2.4).

Import. Converting a non-versioned sequence $\vec{S}$ into an *intensional* multi-version digraph g^* is straightforward and consists of two steps:

– Initialize the graph as a *chain*: $g_{\vec{S}} = chain(\vec{S})$ (cf. (**8.23**) on page 145).
– Convert $g_{\vec{S}} : (V, E)$ into a multi-version graph: $g^*_{\vec{S}} = (V, E, \mathcal{R}, vis)$, where $vis(e) := \hat{a}$ for all $e \in V \cup E$.

Here, $\hat{a}$ is the conjunctive form of the ambition specified by the user for commit, which is assigned to all vertices and edges homologously. Moreover, $\mathcal{R}$ represents the current state of the rule base.

Export. Converting between the repository and the workspace is comparably more complicated. Here, a multi-version digraph g^* has to be linearized into a single-version sequence. The digraph may be arbitrarily shaped and may, in particular, contain cycles.

– Select a single-version view of the multi-version graph: $g = g^*|_c$ (cf. (**8.38**) on page 148). This is usually achieved by a preceding *filter* step.
– *Linearize* the graph g into a sequence: $\vec{S} = linearize(g)$. (cf. Algorithm 8.1 on page 145).

Above, c denotes the tuple-represented choice inferred from the version specification made by the user during check-out. The non-deterministic selection step included in Algorithm 8.1 is resolved using the framework's a-posteriori product analysis mechanism by signaling an *order conflict*; this is a subject of Section 13.3.1.

10.3.3 Sameness Criteria

The metamodel for ordered collections contains two non-abstract classes, OCVertex and OCEdge, for which the sameness criterion formally introduced in Section 10.2.3 remains to be defined. We here assume that vertices carry UUIDs, which can serve as sameness criterion. The graph heuristic matching algorithm shown below explains how these UUIDs are obtained.

Sameness Criterion 1 (*OCVertex*)— Two vertices are the same if their UUIDs are equal.

```
context OCVertex body isSame(other : OCVertex):
self.uuid = other.uuid
```

Sameness Criterion 2 (*OCEdge*)— Two edges are the same if their respective referenced source and target vertices are the same.

```
context OCEdge body isSame(other : OCEdge):
self.source.isSame(other.source) and self.target.isSame(other.target)
```

procedure OCMATCH($g_1^* : (V_1, E_1, VER_1, vis_1), g_2^* : (V_2, E_2, VER_2, vis_2)$)
  $M = \emptyset$
  for all $v_1 \in V_1$ **do**
    $e \leftarrow v_1$.occurringElement
    if $\nexists v_{1x} \in V_1 \big(v_{1x} \neq v_1 \wedge v_{1x}$.occurringElement $= e\big)$ **then**
      $V_{2same} \leftarrow \{v_2 \in V_2 | same(v_1, v_2)\}$
      if $|V_{2same}| = 1$ **then**
        $v_2 \leftarrow$ lone element of V_{2same}
        $M \leftarrow M \cup \{(v_1, v_2)\}$
  for all $m : (v_1, v_2) \in M$ **do**
    $M \leftarrow M \cup$ EXPANDPRED$(v_1, v_2, g_1^*, g_2^*, M)$
    $M \leftarrow M \cup$ EXPANDSUCC$(v_1, v_2, g_1^*, g_2^*, M)$
  for all $m : (v_1, v_2) \in M$ **do**
    if v_1.uuid->oclIsUndefined() **then**
      if v_2.uuid->oclIsUndefined() **then**
        $uuid \leftarrow$ generate new UUID
        v_1.uuid $\leftarrow uuid$
        v_2.uuid $\leftarrow uuid$
      else
        v_1.uuid $\leftarrow v_2$.uuid
    else if v_2.uuid->oclIsUndefined() **then**
      v_2.uuid $\leftarrow v_1$.uuid
function EXPANDPRED$(v_1, v_2, g_1^*, g_2^*, M)$
  for all $\{v_{1p} \in V_1 | (v_{1p}, v_1) \in E_1\}$ **do**
    for all $\{v_{2p} \in V_2 | (v_{2p}, v_2) \in E_2\}$ **do**
      if $same(v_{1p}$.occurringElement$, v_{2p}$.occurringElement$)) \wedge$
          $\nexists m : (v_{1x}, v_{2x}) \in M | v_{1x} = v_{1p} \vee v_{2x} = v_{2p}$ **then**
        $M \leftarrow M \cup \{(v_{1p}, v_{2p})\}$
        $M \leftarrow M \cup$ EXPANDPRED$(v_{1p}, v_{2p}, g_1^*, g_2^*, M)$
  return M
function EXPANDSUCC$(v_1, v_2, g_1^*, g_2^*, M)$
  for all $\{v_{1s} \in V_1 | (v_1, v_{1s}) \in E_1\}$ **do**
    for all $\{v_{2s} \in V_2 | (v_2, v_{2s}) \in E_2\}$ **do**
      if $same(v_{1s}$.occurringElement$, v_{2s}$.occurringElement$) \wedge$
          $\nexists m : (v_{1x}, v_{2x}) \in M | v_{1x} = v_{1s} \vee v_{2x} = v_{2s}$ **then**
        $M \leftarrow M \cup \{(v_{1s}, v_{2s})\}$
        $M \leftarrow M \cup$ EXPANDSUCC$(v_{1p}, v_{2p}, g_1^*, g_2^*, M)$
  return M

Algorithm 10.1: Heuristic Matching of two multi-version ordered collections.

10.3.4 Heuristic Graph Matching

Since graphs imported from single-version collections, represented as sequences in the workspace, do not carry UUIDs, it becomes necessary to infer them by comparison. To this end, a heuristic graph matching strategy, which takes two instances of OrderedCollection as input, is formalized by means of an algorithm. For each vertex matching identified, it is ensured that the corresponding vertices carry equal UUIDs. This matching phase is executed in advance to difference calculation; see Section 10.8.3.

Algorithm 10.1 takes as input the two multi-version graphs that represent versioned sequences extrinsically[1]. The intermediate result of the algorithm is a matching, represented as a set of tuples, each containing a pair of matching vertices from the first and second version. The underlying idea is a generalization of *Heckel's Algorithm* (see Section 4.3.3) from sequence comparison to graph comparison. The heuristics underlying Heckel's Algorithm suit well the particular graph matching scenario considered here, since the graphs are created by chaining and unionizing single-version sequences; this produces rather weakly branched, chain-like graphs. Like in the original form of Heckel's Algorithm, a two-step procedure is applied: First, vertices that uniquely refer to the same element in both versions are identified. Second, immediate predecessors and successors of matched vertices are also matched in case they refer to the same elements. The second step is applied repeatedly until no more "same" predecessors or successors are retrieved. [2]

After having been matched, vertices get the corresponding UUID of the opposite version assigned. In case none of the vertex versions carries a UUID, a new identifier is artificially generated and assigned to both versions.

10.3.5 Example

Figure 10.4 illustrates a concrete application of Algorithm 10.1 to an abstract example, in which it is assumed that the upper graph represents an extrinsically represented version of a collection selected for check out, whereas the lower graph corresponds to an imported collection. Vertex labels represent the elements occurring in the multi-version order defined by the graph. Visibilities, being irrelevant for matching, have been omitted.

Solid blue lines represent vertices matched in the first phase, i.e., which have the same, unique content in both versions. The correspondence between the rightmost occurrences of J is established by expanding the matching of S to predecessors. Conversely, recursive EXPANDPRED/EXPANDSUCC calls produce matches referring to vertices with contents T,

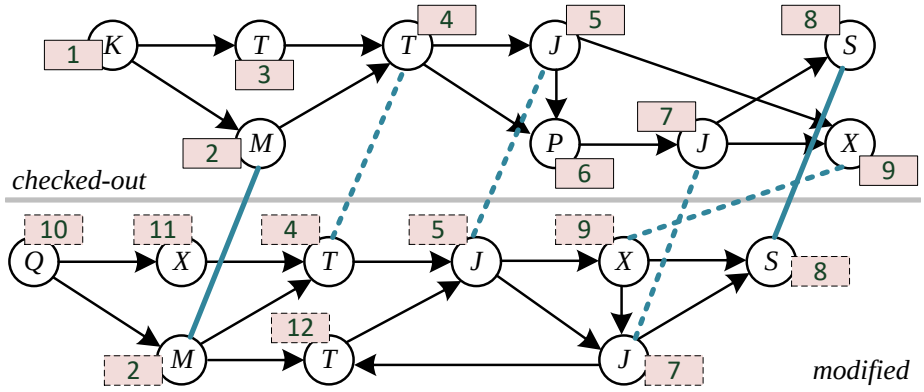


Figure 10.4: Example input and output for heuristic graph matching.

1 Here and in algorithms provided subsequently, expressions written in a typewriter font comply to OCL.

2 For optimization, both phases can also be iterated until the matching does not grow. Furthermore, to increase the number of matchings in the first phase, the comparison window might be increased from one to two elements. Both optimizations were investigated and implemented in [Schi17].

J, and X. In the non-optimized version of the algorithm, one additional pair of instances of T remains unmatched as there remain no equivalent predecessors or successors, whose expansion would reach this pair.

Having been imported from an intrinsic representation, the vertices of the modified version do not initially carry UUIDs. For matching elements, they are copied from the checked-out version, whereas for inserted elements, new UUIDs (10 or greater) are generated.

10.4 Mapping File Hierarchies to the Product Space Base Layer

Although having been referred to as *domain model* in the overview in Section 9.1.2, it would be a too idealistic assumption for the primary versioned artifact to be defined by one single model instance in the workspace. Rather than this, it must be taken into account that realistic model-driven projects consist of several *heterogeneous interconnected artifacts* (**D19**). In this section, a metamodel for *versioned file hierarchies* is presented.

Assumptions. The applicability is potentially restricted by the following items:

– The names of files and folders are stable and its contents are never moved; otherwise, this is treated as a deletion and insertion of a file and all of its contents. This restriction is due to the fact that we cannot make the premise that the underlying workspace file system assign UUIDs to all files and folders.

– The content type of a file is stable in the sense that, e.g., a text file is never converted into a model file, or vice versa.

10.4.1 Structural Design

As shown in the class diagram in Figure 10.5, files and folders are organized hierarchically, supported by the *Composite* design pattern [Gam+95]. The abstract class Resource defines a name for both files and folders. Class File is abstract and not further decomposed here; it is used as a base class for modeling different concrete file types in subsequent sections. References rootResources and contents determine the values of the derived references roots and children introduced in Section 10.2.1.

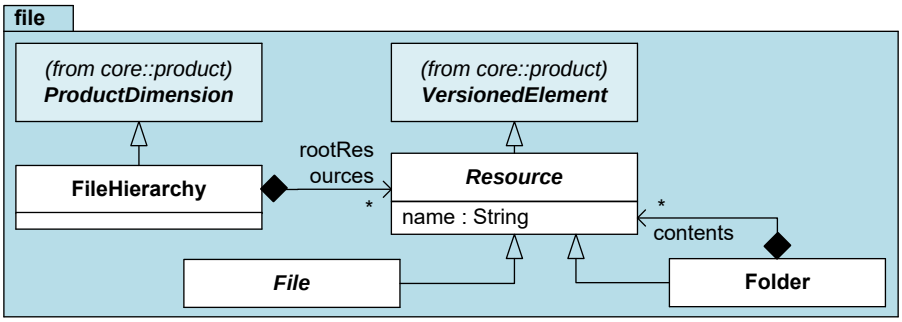


Figure 10.5: Metamodel for versioned file hierarchies. Based on [Schwä+16, Figure 5].

10.4.2 Sameness Criteria

Due to the locality of the predicate *same* in the element hierarchy, as well as due to the fact that file and folder names must be unique inside a parent folder (in usual file systems), the sameness for both files and folders can be broken down to the equality of their names defined in the abstract class VersionedResource:

Sameness Criterion 3 (*VersionedResource*)— Resources are the same if their names match.

```
context VersionedResource body isSame(other : VersionedResource):
self.name = other.name
```

10.5 Text Files

A first concrete instantiation of versioned files is multi-version text files, which are essentially represented as ordered collections of text lines. Extrinsically, *texts* are *represented as models* according to design decision **D11**.

10.5.1 Structural Design

The metamodel for multi-version text files, shown in Figure 10.6, extends the basic file metamodel and utilizes multi-version ordered collections as introduced in Section 10.3. A text file comprises a base set of lines, whose versioned order is contained as an instance of OrderedCollection; multiple occurrences of a line with the same content are handled by reusing the same line instance as occurrences of multiple collection vertices.

Assumptions. The object granularity is fixed to text lines here. On the one hand, this may cause pseudo-conflicts when it comes to concurrent modifications of disjoint regions of the same text line. On the other hand, large blocks of text lines are not consolidated into a single versioned element, which would provide for a better scalability.

10.5.2 Sameness Criterion

The list of sameness criteria is extended by a trivial item:

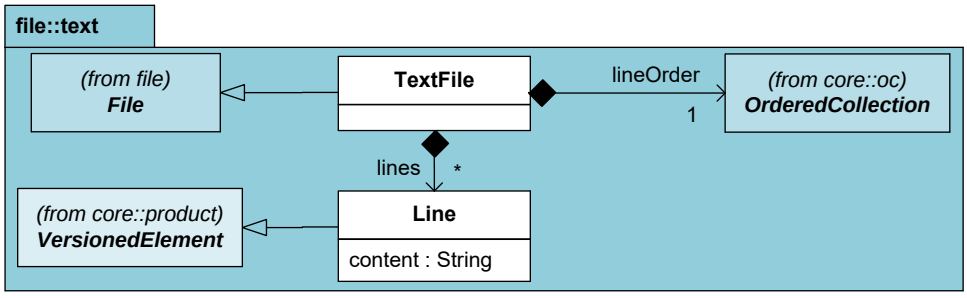


Figure 10.6: Metamodel of multi-version text file representation.

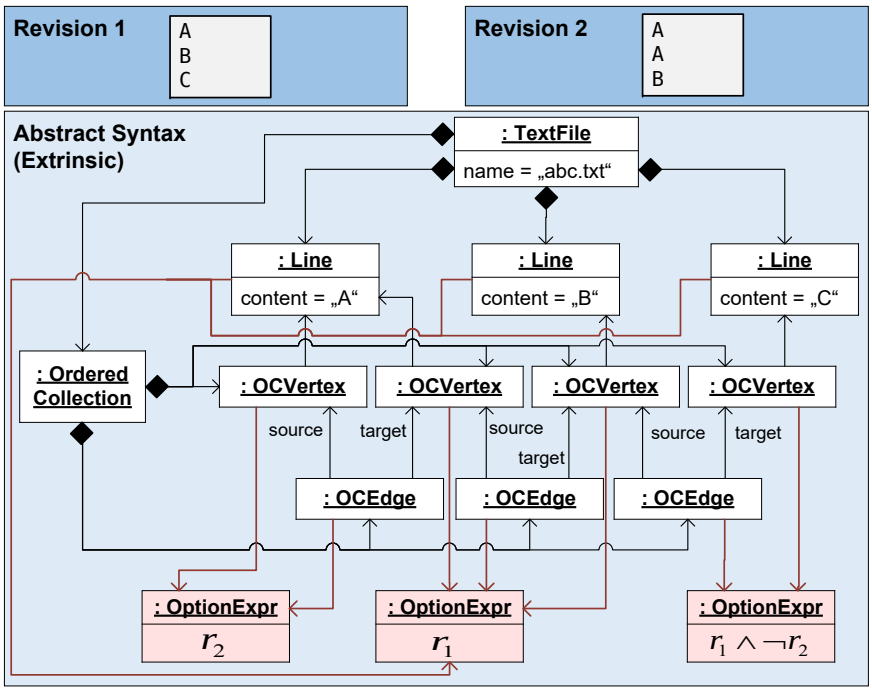


Figure 10.7: Example of multi-version representation of a text file.

Sameness Criterion 4 (*Line*)— Two lines are the same if their contents are equal.

```
context Line body isSame(other : Line):
self.content = other.content
```

10.5.3 Example

The example depicted in Figure 10.7 demonstrates how an instance of the metamodel shown in Figure 10.6 is capable of representing multiple versions – here, two revisions – of the same text file extrinsically. Ensuing from the base contents – three lines, `A`, `B`, and `C` – the second revision is derived by removing the trailing `C` and by adding another occurrence of `A` at the beginning of the text file. Through the mechanisms explained in Chapter 9, unmodified contents obtain the visibility r_1, inserted elements r_2, and deleted elements $r_1 \wedge \neg r_2$, respectively. Option expressions are represented in a condensed form in Figure 10.7. The example demonstrates that new occurrences (here, of line `A`) are created rather than duplicating identical line contents. Furthermore, by deleting the occurrence of `C`, the adjacent edge (`B`,`C`) is deleted, too.

10.6 EMF Model Instances

In Sections 6.2 and 6.3, it was argued that line-oriented representation of model instances – e.g., based on the OMG XMI standard – is not the adequate representation for the operations of VC and of SPLE, respectively. This makes it indispensable to introduce a structured

multi-version model representation which we here refer to as *extrinsic*. Technically speaking, in the repository, models are not represented as instances of their metamodel, but as instances of a universal, version-aware metamodel for EMF instances.

Assumptions. Firstly, the extrinsic representation allows to add *visibilities* to model elements. Secondly, EMF well-formedness rules would be too restrictive for representing multi-version models in a superimposed way. In particular, the following restrictions hold for the *intrinsic*, and are suspended by the *extrinsic* representation:

– The location of an object in the containment hierarchy of an EMF model must be fixed. This disallows storing the result of a move operation in a superimposed form without creating multiple copies of the object.
– The multiplicities of structural features are defined with single-version semantics in the metamodels. In particular, it is not possible to represent multiple values for single-valued attributes or references. This, however, is crucial when allowing rename operations.
– The class of an object is fixed after creation. Although model editing tools do not usually offer a command "change class", this should be taken into account when supporting co-evolution of model and metamodel. For instance, a new revision of a metamodel may be organized under a different package URI (see below).

As a consequence, the subsequently presented extrinsic EMF metamodel incorporates many design decisions such as a flattened containment tree, versioned metadata (i.e., classes and structural features), and fine-grained versioning (i.e., each individual value of every structural feature carries its own visibility).

10.6.1 Structural Design

The metamodel[3] shown in Figure 10.8 reflects these considerations by representing not only objects but every possibly varying detail of objects, such as their classes and values of structural features, as versioned elements. Furthermore, since the location of an object in the EMF containment hierarchy is allowed to vary, the containment tree is flattened, such that an EMF resource is extrinsically interpreted as a flat set of objects.

References to classes and structural features are further divided up into *internal* and *external*. Here, internal means that the model is co-versioned with its metamodel, such that the "instance of" relationship can be represented as an object link extrinsically. In the case of external class and feature definition – which is the standard case, assuming that metamodels are typically contained in separate resources outside the version control – , the reference to metadata is made indirect by storing implicit addresses such as package URI, class name, and feature name, which in turn refer to metamodels and models of the intrinsic workspace.

Similarly, yet orthogonally to the above distinction, values for references can be defined in an internal or external way. An internal reference value targets a versioned object available in the same resource set (i.e., file hierarchy), whereas an external reference value maps an object stored elsewhere in the local file system by its persistent URI. The values of

3 This metamodel was heavily influenced by the metamodels underlying the tools *BTMerge* [SUW13b] (which addresses three-way merging) and *FAMILE* [BS12b] (an MDSPLE tool based on annotative variability).

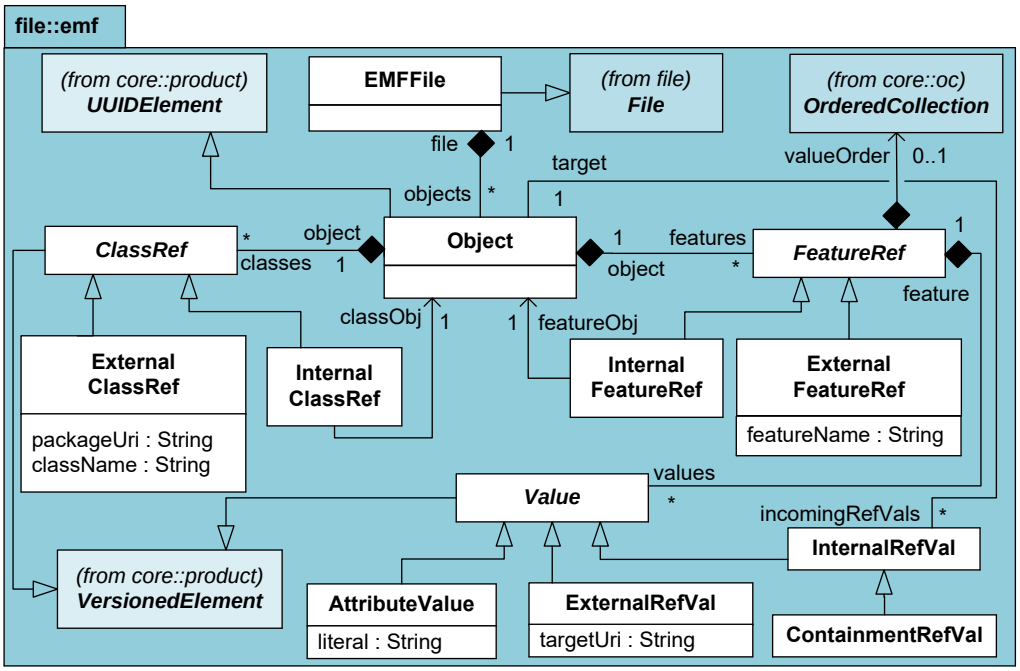


Figure 10.8: Metamodel of multi-version EMF model representation. Based on [Schwä+16, Figure 6].

containment references are made explicit by subclass ContainmentRefVal. In turn, attribute values are always serialized in their custom string-valued literal representation.

In case the metamodel defines a multi-valued structural feature as *ordered*, the versioned set of values is supported by a valueOrder using the multi-version representation of sequences described in Section 10.3. This order is taken into consideration when importing or exporting the corresponding single-version collection.

10.6.2 Sameness Criteria

We define sameness criteria for all concrete subclasses of VersionedElement located in the package file::emf. To begin with, objects are matched based upon their UUIDs. Internal and external class references are matched by their metadata expressed. Accordingly, feature references are handled. Last, the sameness criteria for structural feature values depend on the specific value representation.

Sameness Criterion 5 (*Object*)— Two objects are the same if their UUIDs are equal.

```
context Object body isSame(other : Object):
self.uuid = other.uuid
```

Sameness Criterion 6 (*ExternalClassRef*)— Two external class references are the same if their referenced package URI and class name are equal.

```
context ExternalClassRef body isSame(other : ExternalClassRef):
self.packageUri = other.packageUri and self.className = other.className
```

Sameness Criterion 7 (*InternalClassRef*)— Two internal class references are the same if the internal objects they are represented by are the same.

```
context InternalClassRef body isSame(other : InternalClassRef):
self.classObj.isSame(other.classObj)
```

Sameness Criterion 8 (*ExternalFeatureRef*)— Two external structural feature references are the same if their names are equal.

```
context ExternalFeatureRef body isSame(other : ExternalFeatureRef):
self.featureName = other.featureName
```

Sameness Criterion 9 (*InternalFeatureRef*)— Two internal structural feature references are the same if the internal objects they are represented by are the same.

```
context InternalFeatureRef body isSame(other : InternalFeatureRef):
self.featureObj.isSame(other.featureObj)
```

Sameness Criterion 10 (*AttributeValue*)— Two attribute values are the same if the literals serialized from them are equal.

```
context AttributeValue body isSame(other : AttributeValue):
self.literal = other.literal
```

Sameness Criterion 11 (*ExternalRefVal*)— Two external reference values are the same if their target objects' URIs are equal.

```
context ExternalRefVal body isSame(other : ExternalRefVal):
self.targetUri = other.targetUri
```

Sameness Criterion 12 (*InternalRefVal*)— Two internal (containment or non-containment) reference values are the same if the referenced objects are the same.

```
context InternalRefVal body isSame(other : InternalRefVal):
self.target.isSame(other.target)
```

10.6.3 Example

In Figure 10.9, a simplistic example of a state diagram is shown in both intrinsic – i.e., as instance of its regular single-version metamodel – and extrinsic – i.e., as instance of the extrinsic EMF metamodel introduced in this section – form. In the latter case, each model element is represented as an instance of Object; its class is versioned externally. Furthermore, the values of the structural features – names of both states and transition, source and target of the transition – are represented as individual fine-grained versioned elements.

The example demonstrates that the extrinsic representation is finer grained than the intrinsic. We have refrained from using visibilities when compared to the example shown in Figure 10.7. Furthermore, there exists no ordered collection in this example, which would have resulted in a considerably larger extrinsic model representation.

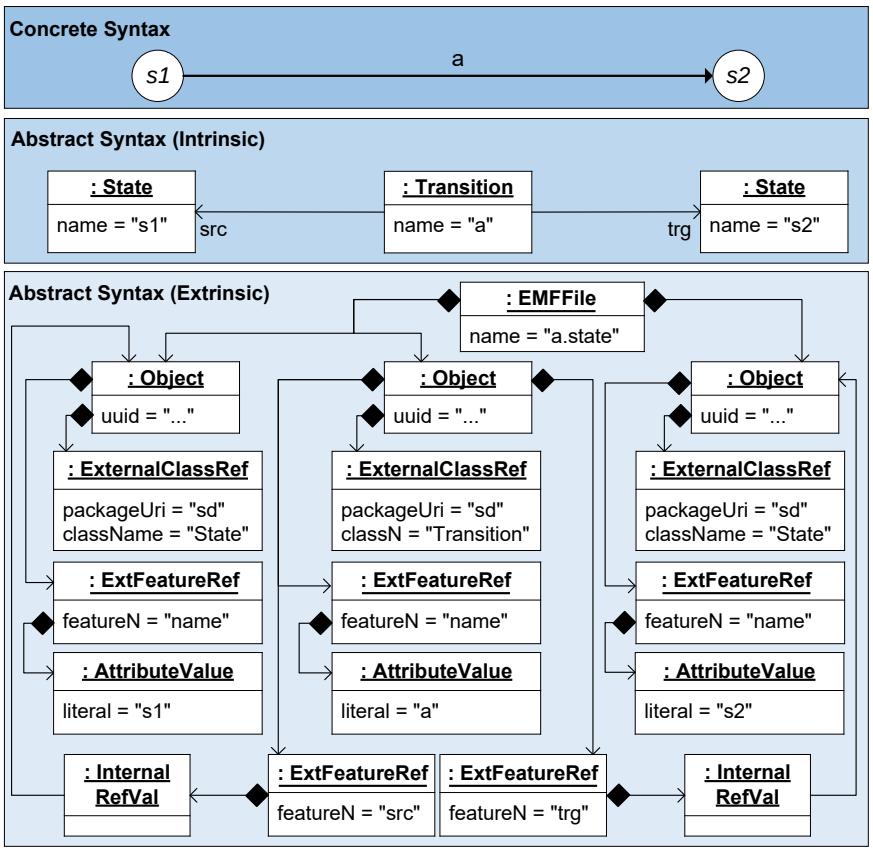


Figure 10.9: Example of extrinsic EMF model representation.

10.7 Mapping Feature Models to the Product Space Base Layer

After having presented feature models as a part of the version space in Section 9.4, we here revisit them in their dual role as an additional product dimension.

There are two reasons why it would be inadequate to reuse an existing single-version metamodel for feature models and to extrinsically apply the multi-version EMF representation defined in the previous subsection to it. First, in addition to the syntactic structure, the semantics, which is expressed by low-level rule base concepts, must be managed, too. Second, an individual multi-variant feature metamodel provides for more precise conflict detection and resolution; see Section 13.3.3.

Assumptions. The subsequently presented metamodel and sameness criteria take into account that the following evolutionary modifications have to be supported:

– insertion and deletion of features, groups, and requires/excludes relationships;

– renaming of features;

– change of the mandatory/optional state of features;

– deletion of features;

– changes in the parent/child hierarchy;

– definition of a different root feature;

– changes to the feature set contained in a group.

As a consequence, all pieces of information potentially changed through such modifications should be version-aware. Therefore, even ordinary attributes like name, which suffice for single-version representation, have been refined into classes – inheriting from VersionedElement in order to carry a visibility – in the multi-version format.

10.7.1 Structural Design

In Figure 9.10, a simplified view on the feature metamodel was provided. The class diagram presented in Figure 10.10 refines this view, fading out version space related details such as references to the base layer.

Moreover, parent/child relationships, group memberships, as well as the decision which feature represents the root of the feature model, are variable. This is reflected in additional classes such as ChildRelationship, GroupMembership, and RootRelationship. Motivated by unconstrained variability, the multi-variant representation permits that features be part of multiple groups or that they have multiple parents. Similarly to the extrinsic EMF model representation, the hierarchy of feature models is flattened in their multi-version representation.

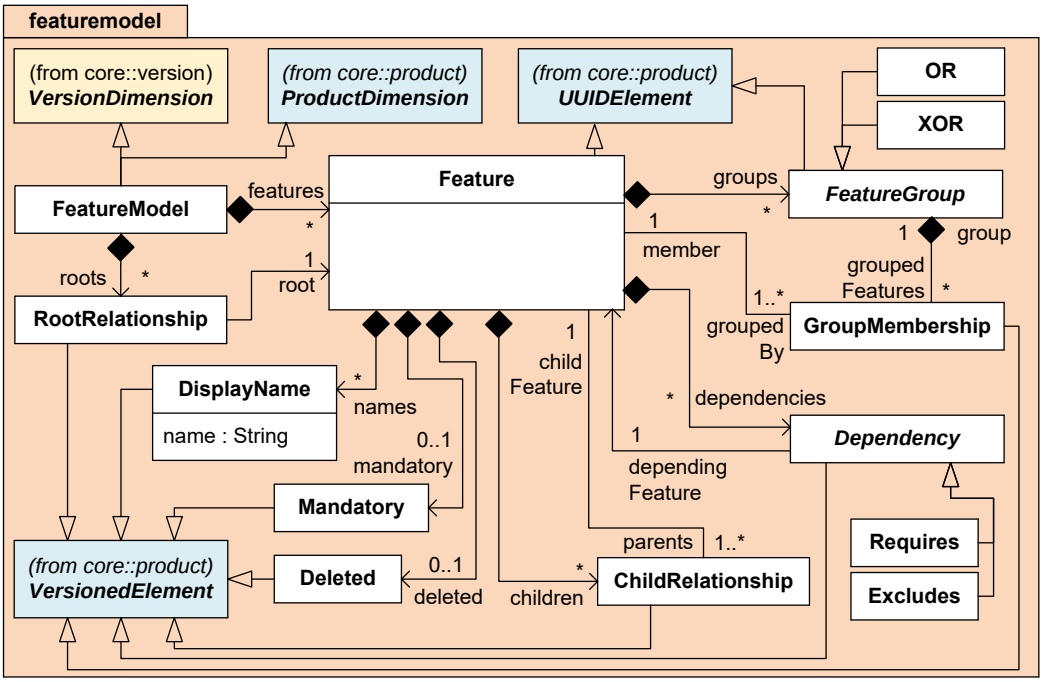


Figure 10.10: Product centric view on the metamodel for multi-version feature models. Based on [Schwä+16, Figure 8].

The existence of the versioned element class Deleted is justified by the fact that deleted features are only graphically hidden in the workspace feature model (see Section 13.2.4); therefore, deletions should not be made effective by the filtering mechanism, but rather, the information in which revision a feature has been deleted must be versioned.

10.7.2 Sameness Criteria

Features and groups are matched based upon their UUIDs. Root and child relationships as well as group memberships are the same in case their referenced root, child, or member value correspond. Moreover, dependencies are matched based on the sameness of their dependingFeature. Display names are matched by name, whereas versioned Mandatory and Deleted flags have a trivial sameness criterion since there is only one such object contained per feature. (Recall that sameness is local to the element hierarchy.)

Sameness Criterion 13 (*Feature*)— Two features are the same if their UUIDs are equal.

```
context Feature body isSame(other : Feature):
self.uuid = other.uuid
```

Sameness Criterion 14 (*RootRelationship*)— Two root relationships are the same if their referenced features are the same.

```
context RootRelationship body isSame(other : RootRelationship):
self.root.isSame(other.root)
```

Sameness Criterion 15 (*ChildRelationship*)— Two child relationships are the same if their referenced children are the same.

```
context ChildRelationship body isSame(other : ChildRelationship):
self.childFeature.isSame(other.childFeature)
```

Sameness Criterion 16 (*FeatureGroup*)— Feature groups (instances of OR or XOR) are the same if their UUIDs are equal.

```
context FeatureGroup body isSame(other : FeatureGroup):
self.uuid = other.uuid
```

Sameness Criterion 17 (*GroupMembership*)— Two group memberships are the same if their referenced members are the same.

```
context GroupMembership body isSame(other : GroupMembership):
self.member.isSame(other.member)
```

Sameness Criterion 18 (*Dependency*)— Two feature dependencies are the same if their depending features are the same.

```
context Dependency body isSame(other : Dependency):
self.dependingFeature.isSame(other.dependingFeature)
```

Sameness Criterion 19 (*DisplayName*)— Display names are the same if their texts are equal.

```
context DisplayName body isSame(other : DisplayName):
self.name = other.name
```

Sameness Criterion 20 (*Mandatory*)— Flags for mandatory features are always the same.

```
context Mandatory body isSame(other : Mandatory): true
```

Sameness Criterion 21 (*Deleted*)— Two deletion flags are always the same.

```
context Deleted body isSame(other : Deleted): true
```

10.8 Matching, Differencing, and Merging

After having portrayed two different concrete product dimensions, let us now return to the product space base layer, focusing on three low-level VCS operations that are used as building blocks of CHECKOUT, COMMIT, et cetera.

The editing model semi-formally introduced in Section 9.5.1 implicitly includes the problems of matching, differencing, and merging two copies of an extrinsically represented repository (cf. steps 2 and 6 in the COMMIT phase). After having concretized the representation of different product dimensions, we are now able to define a data structure as well as generic algorithms for matching, differencing, and asymmetric two-way raw merging.

10.8.1 Structural Design

In Figure 10.11, a metamodel for *matchings* and *differences* of two product space instances is depicted. For the two-way comparison, two versions left and right are assumed, where the

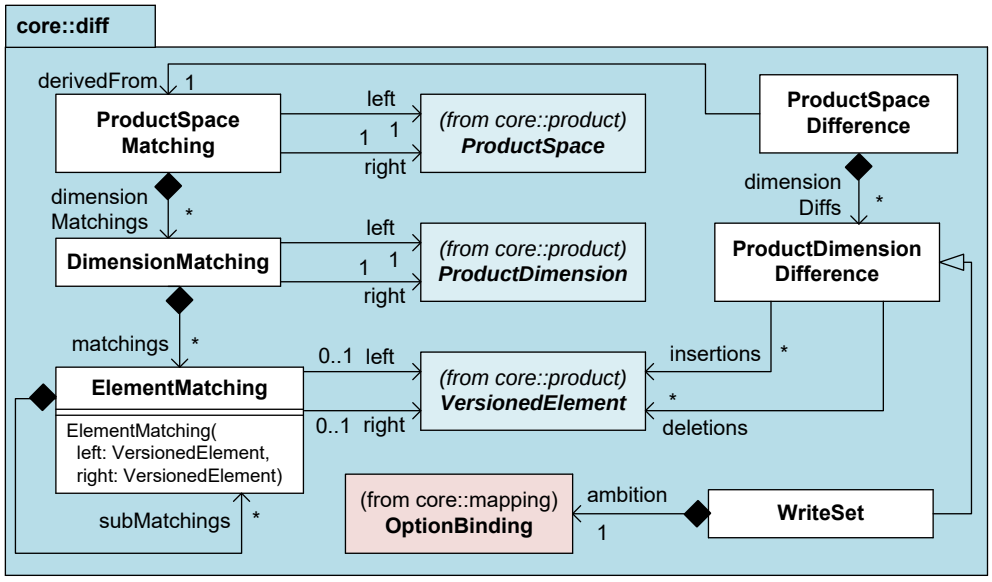


Figure 10.11: Metamodel for matchings, differences, and write sets.

```
function MATCH(leftPs, rightPs)
    psm ← new ProductSpaceMatching
    for all d ∈ {1, ..., number of product dimensions} do
        pdm ← new DimensionMatching
        pdm.left ← leftPs.dimensions.at(d)
        pdm.right ← rightPs.dimensions.at(d)
        rightMatches ← ∅
        for all l ∈ pdm.left.roots do
            leftMatch ← false
            for all r ∈ pdm.right.roots do
                if class(l) = class(r) ∧ same(l, r) then
                    pdm.matchings ← pdm.matchings ∪ {MATCHREC(l, r)}
                    leftMatch ← true
                    rightMatches ← rightMatches ∪ {r}
                    break
            if ¬leftMatch then
                pem ← new ElementMatching(l, ε)
                pdm.matchings ← pdm.matchings ∪ {pem}
        for all r ∈ (pdm.right.roots \ rightMatches) do
            pem ← new ElementMatching(ε, r)
            pdm.matchings ← pdm.matchings ∪ {pem}
        psm.dimensionMatchings ← psm.dimensionMatchings ∪ {pdm}
    return psm
function MATCHREC(l, r)
    pem ← new ElementMatching(l, r)
    rightMatches ← ∅
    for all lc ∈ l.children do
        leftMatch ← false
        for all rc ∈ r.children do
            if class(lc) = class(rc) ∧ same(lc, rc) then
                if class(lc) = class(rc) = OrderedCollection then
                    GRAPHMATCH(lc, rc)                          ▷ Algorithm 10.1
                pem.subMatchings ← pem.subMatchings ∪ {MATCHREC(lc, rc)}
                leftMatch ← true
                rightMatches ← rightMatches ∪ {rc}
                break
        if ¬leftMatch then
            cpem ← new ElementMatching(lc, ε)
            pem.subMatchings ← pem.subMatchings ∪ {cpem}
    for all rc ∈ (r.children \ rightMatches) do
        cpem ← new ElementMatching(ε, rc)
        pem.subMatchings ← pem.subMatchings ∪ {cpem}
    return pem
```

Algorithm 10.2: Generic matching for two given product space versions.

difference is calculated under the assumption that left is more recent than right. [4]

A product space matching consists of a matching of mutually corresponding dimensions. Each dimension matching in turn defines a tree of element matchings, each referring to a versioned element from the left and/or right version. In case an element does not have a matching partner in the opposite version, the corresponding reference is left unset.

Differences represent the same information as matchings, but in a further processed form. An instance of ProductDimensionDifference refers to inserted and deleted sub-trees of versioned elements. These differences are organized in a flat list rather than hierarchically as in the corresponding matching. In this structural representation, a *write set* – semi-formally introduced in Section 9.6.2 – is a product dimension difference decorated with an ambition based on which the visibilities of inserted and deleted elements are to be updated.

10.8.2 Generic Matching Algorithm

Usually, a matching is computed based on two related versions of the same artifact, here, product space. Based upon the predicate $same$, we are now able to give a specification of a generic matching operation, which is shown in Algorithm 10.2.

The algorithm proceeds as follows: First, it is assumed that equivalent product dimensions are matched by position[5]. Then, the sets of root elements of both versions are compared in a pair-wise way. As soon as the sameness criterion holds for a pair, this is recorded as a matching. In addition, unmatched elements are captured as matchings with a missing left or right side. In the case of ordered collections, Algorithm 10.1 enhances the elements of both versions of the graph with UUIDs. Function MATCHREC describes a recursive descent, which is applied for the children sets of matched elements.

10.8.3 Generic Differencing Algorithm

As mentioned before, differences can be derived from given matches. In the generic Algorithm 10.3, a product space matching is converted into a product space difference, and so are contained product dimension matchings transformed into product dimension differences. In contrast to the matching, the structure of referenced elements is not hierarchical, but each detected modification is represented as an instance of the references insertions and deletions directly ensuing from the corresponding dimension difference. By convention, insertions refer to the left, and deletions to the right version only.

10.8.4 Generic Asymmetric Two-way Raw Merging

The product-level two-way merge operation defined in Algorithm 10.4 relies on two assumptions: First, one of the copies to be compared, namely the right version, is considered as a "master" product space, such that differences are intended to be copied from left to right, but not vice versa (*raw* merging). Second, deletions are ignored because they are made effective not by modifying the contents of the version dimensions, but by updating the visibilities of

4 This convention is assumed in many model comparison tools including EMF Compare; see Section 6.2.1.

5 This implies that product dimensions cannot be dynamically added and removed in the course of the MODIFY step.

```
function DIFF(psm)
    psDiff ← new ProductSpaceDifference
    for all pdm ∈ psm.dimensionMatchings do
        pdDiff ← new ProductDimensionDifference
        for all pem ∈ pdm.matchings do
            if pem.left = ε then
                pdDiff.deletions ← pdDiff.deletions∪{pem.right}
            else if pem.right= ε then
                pdDiff.insertions ← pdDiff.insertions∪{pem.left}
            else
                DIFFREC(pdDiff, pem)
        psDiff.dimensionDiffs ← psDiff.dimensionDiffs∪{pdDiff}
    return psDiff
procedure DIFFREC(pdDiff, pem)
    for all cpem ∈ pem.subMatchings do
        if cpem.left= ε then
            pdDiff.deletions ← pdDiff.deletions∪{cpem.right}
        else if cpem.right= ε then
            pdDiff.insertions ← pdDiff.insertions∪{cpem.left}
        else
            DIFFREC(pdDiff, cpem)
```

Algorithm 10.3: Generic differencing based on a given product space matching.

affected elements as described in Section 9.5.1. This way, this special asymmetric type of two-way raw merge algorithm effectively realizes what has been informally described as "append inserted elements to the product space".

In contrast to the previous operations, Algorithm 10.4 contains more informal steps since its details can hardly be specified on a purely conceptual level without making assumptions about the dimension-specific implementation of this operation. Insertions are transferred by finding the corresponding location – i.e., the containing structural feature of the parent element – in the right version and creating a copy of the inserted element.

```
procedure MERGE(leftPs, rightPs, psDiff, psMatch)
    for all d ∈ {1, ..., number of product dimensions} do
        leftDim ← leftPs.dimensions.at(d)
        rightDim ← rightPs.dimensions.at(d)
        pdMatch ← psMatch.dimensionMatchings.at(d)
        pdDiff ← psDiff.dimensionDiffs.at(d)
        for all leftIns ∈ pdDiff.insertions do
            leftParent ← leftIns.parent
            rightParent ← find match for leftParent in pdMatch
            sf ← structural feature by which leftParent contains leftIns
            rightIns ← create copy of leftIns
            resolve cross-links in contents of rightIns
            add rightIns as new value to the structural feature sf of rightParent
```

Algorithm 10.4: Generic asymmetric two-way raw merging.

10.9 Related Work

The concepts presented in this chapter have their equivalences mainly in the field of MD-SPLE, but further specific algorithms are also VCS-related. Below, we list approaches offering similar solutions for versioned file hierarchies, extrinsic model representation, and versioned feature models. Furthermore, approaches belonging to the category of transformational variability are surveyed, as in this line of research, related problems are solved with largely different techniques.

Versioned File Hierarchies. Here, a metamodel for versioned file hierarchies was contributed; this enables version management for entire file hierarchies, explicitly supporting two content types: plain text files and model resources complying to the Eclipse Modeling Framework. Some related approaches support similar mechanisms.

The *Leviathan* file system [Hof+10] incorporates a toolchain-agnostic variability management solution for hierarchies of source code files containing preprocessor annotations. Based on a variant definition specified by the user, the presented file contents are filtered. To this end, low-level system calls such as *mount*, *open*, *read*, or *write*, are re-implemented with a variability-aware semantics. In contrast to our approach, there exists no *workspace* where the filtered contents are temporarily made available, but all operations directly alter the underlying multi-variant representation through the filter.

Similarly, *Rational MultiVersion File System*[6] (MVFS) allows to access versioned file hierarchies through a (read-only) filter. The solution has been implemented at operating system level, such that every change to the filter requires a re-mount. Nevertheless, this is considered as the most general solution towards versioned file hierarchies.

ICE [ZS97] is equipped with a *virtual* file system whose back-end relies on C preprocessor variability. Rather than re-implementing system calls as done in the Leviathan and Rational MultiVersion systems, ICE relies on a version management layer on top, which is part of the configuration management toolchain.

Multi-Version Sequences. The idea of representing multi-version sequences as graphs is not unique in the literature. In [SBK88], a multi-version text file is represented as a set of *threads*—ordered paths of text fragments, which may in turn be shared among different threads. Effectively, the superimposed structure corresponds to a directed graph.

[EWC13] introduce a generic intensional versioning concepts for directed graphs based on the *choice calculus* [EW11]. In addition to the structural representation, many *graph algorithms* are translated into a variability-aware form, such that they operate on multi-variant graphs. Sequence linearization, however, is not covered by this partially filtered approach.

Graph Comparison Algorithms. The matching strategy shown in Algorithm 10.1 assumes a directed graph whose specific properties – connectedness, sparseness, weak branching, and potential cyclicity – are due to the fact that the graph versions to be compared have been created from a superimposition of single-version sequences.

6 http://www-01.ibm.com/support/docview.wss?uid=swg21230196

Related to this algorithm are generic *graph matching* methods, which are divided up into *exact* and *non-exact* algorithms [Con+04], as well as heuristics such as *similarity flooding* [MGMR02]. These algorithms are more general and well-optimized, but not enough specific to the here discussed multi-version sequence matching problem.

Extrinsic Multi-Version Model Representation. The extrinsic representation of EMF model instances presented here relieves multi-version model instances from single-version metamodel restrictions. Related approaches have their origins both in MDSPLE and in model version control.

In [Wes14], an extrinsic representation for superimposed EMF model instances has been proposed: *versioned model graphs*. Although originally intended for three-way model merging, this structure can be generalized into arbitrary multi-version scenarios by transitioning from extensional to intensional versioning (cf. Section 8.2.4). Merged model graphs also consider that the class as well as the container of an object may vary; a design decision that has been transferred to the extrinsic EMF metamodel presented in Section 10.6.1.

The idea of representing models not as instances of their regular metamodel but of a generic metamodel that allows to overcome context-free restrictions such as cardinalities, was investigated in a more general form in the context of *deep modeling* [AK15], where an explicit distinction between a *linguistic* and an *ontological* metamodel is made. On the linguistic level, models may violate metamodel constraints; the ontological compliance check is orthogonal and optional.

Several approaches aim at overcoming metamodel restrictions – in particular, the multiplicity of single-valued structural features – with virtual extensions to a multi-version domain model, which is represented as an ordinary intrinsic metamodel instance. For instance, the MDSPLE tool FAMILE offers *alternative mappings* [BS16c], which are physically contained in an explicit mapping model (cf. Section 6.1.1). Similarly, in the context of MVC, the tool AMOR [Alt+08] represents alternative values for the resolution of three-way merge conflicts as model annotations.

Versioned Feature Models. The refined feature metamodel presented here offers fine-grained revision control for feature models in the sense that every detail – feature names, parent/child relationships, dependencies, et cetera – are allowed to evolve. Related approaches to feature model versioning have different individual properties.

As anticipated in Section 6.3.2, *hyper feature models* [SSA14a] add an individual revision graph to each feature. This way, however, only the connection to the product space (here, model deltas) is versioned, but not the properties (such as name, optionality, etc.) of the feature itself.

Transformational Variability. The conceptual framework presented here is an example of the usage of *annotative variability* to manage fine-grained model versioning. In the literature, there are representatives of *transformational variability* to organize the multi-version domain model.

DeltaEcore [SSA14b] is a delta-oriented language for model-driven transformational variability. The approach distinguishes between two types of deltas, which are explicitly written in a *delta dialect* that must be defined in advance. Model-level deltas constitute a natural combination of transformational variability and *forward deltas*. Yet, using (dialects of) explicit delta languages requires to deviate from the actual modeling language (see Section 7.1.4). In more recent research, it was attempted to mitigate the extent of explicit user effort by inferring delta dialects as well as evolution deltas by change recording [Wil+17].

In DeltaEcore, deltas are connected to features of the feature model. In contrast, the tool *SiPL* [Pie+15] allows for a more general connection to the version space by defining *presence conditions* for deltas. Furthermore, as already explained in Section 6.1.2, product-level deltas are deduced by model comparison rather than being hand-written.

10.10 Summary

The conceptual framework presented in this thesis makes a general distinction between a version space, which has been the subject of the preceding chapter, and a product space, which has been presented in the current chapter. In general, the framework makes only few assumptions about the structure of the product space; it is supposed to consist of a tree of elements, each capable of carrying a visibility. Furthermore, for each type of element, a sameness criterion is defined, which is used for comparing, differencing, and merging two versions of a product space—corresponding algorithms have been provided at the end of this chapter.

Two concrete version dimensions have been investigated. A versioned file hierarchy may reflect an arbitrary cut-out of a file system. Supported file types comprise plain text and EMF files, but new types such as XML can be added to the framework retrospectively. The second dimension corresponds to the feature model, which also forms the intersection set between version space and product space. This way, the extensible product space satisfies the requirement of *generality* (cf. **R11** from Section 2.3).

Both multi-version EMF models and feature models are represented inside the repository in an *extrinsic* way; their representation differs from the workspace – where *automatically derived* contents (**R9**) can be edited using single-version tools (**R10**) – inasmuch as they allow for *unconstrained variability*, such that *fine-grained* revision and variant control (cf. **R16**) is supported.

The algorithms MATCH, DIFF, and MERGE, as well as the translation operations IMPORT and EXPORT are taken up in the subsequent Chapter 11, where the consistency-preserving dynamic editing model of the conceptual framework is formally defined. Furthermore, in Chapter 13, mechanisms for ensuring the *well-formedness* of product space artifacts to be exported from the extrinsic representation into the single-version workspace are presented. The metamodels presented in this chapter also serve as foundation for the model-driven implementation of the tool *SuperMod* explained in Chapter 14.

There is not only
the current version to change,
but also last year's version
(which is still supported)
and next year's version
(which almost runs).

MARC J. ROCHKIND (1975)

Chapter 11

Consistency-Preserving Dynamic Editing Model

Abstract

The antagonists of the conceptual framework, namely the repository, which combines version space and product space, as well as the workspace, where single-version artifacts are organized, are combined by a filtered editing model. This editing model is dynamic inasmuch as it allows for different kinds of evolution: The feature model and the domain model may be edited concurrently. Features, which define the workspace contents or the scope of the current change – represented by the ambition –, may be introduced or deleted. Furthermore, the ambition may be revised until commit. The dynamism of this filtered editing model raises consistency problems concerning the evolving relationships between the version space, the choice, the ambition, and the modified workspace contents. In this chapter, nine consistency constraints are formalized based on the version space base layer. Furthermore, three consistency-preserving algorithms for the workspace operations check-out, commit, as well as for a new operation, migrate, are presented. Last, a generalized editing model, whose amount of dynamism can be adjusted by the user, is sketched.

Contents

11.1 Problem Statement

In the preceding chapters, it has been argued that supporting both evolution and variability at a time is far from trivial. Things become even more complicated when allowing the variability model, as well as the connection between variability model and product, to co-evolve. This chapter is dedicated to consistency questions dealing with these relationships. It contributes both consistency constraints and consistency-preserving algorithms that enable *dynamic filtered editing* (DFE). [1]

This chapter is organized as follows: After motivating and classifying the evolution scenario in the remainder of the current Section 11.1, consistency constraints are formalized upon the version space base layer in Section 11.2. Next, algorithms for the operations CHECKOUT and COMMIT, which have been sketched semi-formally in Section 9.5.1, are precisely defined in Section 11.3. In addition, we introduce a new operation, MIGRATE, which prepares the current workspace contents for the subsequent iteration, obviating repeated check-outs. In Section 11.4, we deal with the otherwise neglected revision graph, supplying proof that it is managed automatically in a consistent way. The dynamic editing

1 A greater part of the problem motivation, as well as the formalization provided in Chapters 11.2 and 11.3, have been pre-published in [SW17b].

model is exemplified in multiple steps in Section 11.5. More informally, a generalized editing model, which allows to mix the static and the dynamic style of filtered editing, is presented in Section 11.6. Before concluding with a summary, we present differences to related approaches to filtered editing in Section 11.7.

11.1.1 Static vs. Dynamic Filtered Editing

The peculiarities of the here contributed *dynamic* editing model are best explained by a comparison with the so considered conventional approach, *static filtered editing* (SFE). Concrete representatives are discussed in the related work section; we here use the vocabulary used in the own conceptual framework and continue under the assumption that design decisions **D4** (*fully filtered editing*), **D5** (*hidden variation points*), and **D7** (*transactional filtered editing*) have to be satisfied by both approaches.

Commonalities. The filtered editing (FE) approaches considered here have in common that they operate in an *iterative* way, where each iteration is a transaction begun with *check-out* and concluded with *commit*. In between, workspace contents may be modified. The workspace view is defined by a *choice* – or read filter –, which is a *unique* version selection. By an *ambition* – a *partial* write filter –, the versions affected by the change are defined. This way, the version described by the choice is *representative* for the set of versions described by the ambition.

Static Filtered Editing. In SFE, both the choice and the ambition are defined in one step at the beginning of a workspace transaction, i.e., at check-out. Typically, the ambition is defined first as a partial version selection, and further configured top-down into a unique choice. Both choice and ambition, as well as the version space itself, are immutable during MODIFY. After COMMIT, the transaction is closed and the workspace is cleared; subsequent transactions must be started explicitly. Furthermore, changes to the version space are allowed as long as no workspace transaction is active.

Dynamic Filtered Editing. With DFE, we here denote that the scope of a change – represented by the ambition – is specified only at COMMIT time. Furthermore, the version space (or parts thereof, namely the feature model) is made available in the workspace for modification. This way, it is possible to introduce those features to which a change is relevant while the change is actually performed. Another property of the dynamic approach is that a new transaction is started immediately after *commit*. For the new transaction, it is assumed that the same choice as in the previous iteration shall be used—an assumption that is obtained by generalizing the VCS workflow. It is, however, possible to define a different view explicitly by CHECKOUT.

The here presented framework primarily assumes DFE, but it also allows to step back to SFE in case a more restrictive workflow is desired. This is explained in Section 11.6 in the context of the *generalized editing model*. As far as there, the given explanations assume that purely dynamic filtered editing is applied.

11.1.2 Unproblematic Iterations of Dynamic Filtered Editing

To begin with, let us reconsider two previously presented examples that already make the benefits of DFE obvious.

In Figure 2.5 on page 30, an uncolored labeled graph is checked-out. Then, the workspace is modified by adding an attribute weight to class Edge. This modification might have been initially planned as an evolution step connected to an existing feature; however, right before committing, the user might realize that this actually represents a new feature. Since it is allowed to alter the feature model, he/she may introduce a new feature weighted, which is used in the ambition thereafter.

A second example has been shown in Section 9.5.2, more precisely in Figure 9.13 on page 179. In all revisions 1 until 3, features used for selection in the ambition are introduced in the course of the MODIFY phase. The example also illustrates the aforementioned assumption that it is usual to stay in the current workspace view for the subsequent iteration. Merely, after having committed revision 1, new workspace content is generated by check-out.

11.1.3 Consistency Violations by Example

It is not difficult, however, to construct less benevolent cases where the dynamism implied by the considered editing model becomes problematic. We here sketch representative instances of consistency violations, illustrated by Figure 11.1, which are avoided by the consistency-preserving DFE model. These constraints are further formalized in a generalized evolution scenario subsequently.

Non-Unique or Inconsistent Choice. Let us assume the feature model depicted in Figure 11.1(a), from which a version is to be selected for CHECKOUT. Then, the feature configuration shown in (b) is *non-unique*, since features Vertices and Colored do not have a selection state assigned. Moreover, (c) represents an *inconsistent* choice: The mandatory feature Vertices is deselected. In contrast, the feature configuration shown in (d) is both unique and consistent, thus it is assumed for the subsequent steps.

Disallowed Feature Model Modification. During MODIFY, the feature model may be edited, however, not arbitrarily. For example, in (e), features Weighted and Directed are made mandatory and at the same time arranged in an XOR group. This contradicts the semantics of feature models. A different problem is illustrated in (f): Feature Weighted, which is currently selected in the active choice (d), is deleted. In the workspace, however, elements connected to this features are still present.As a consequence, the workspace contains elements which could bot be selected by any future choice.

Non-Represented or Inconsistent Ambition. Moving further on to the COMMIT phase, in Figure 11.1(g), a user-specified feature ambition is depicted. Since the mandatory feature Vertices is bound negatively there, the ambition represents an *inconsistent* set of versions. Similarly, the ambition depicted in (h) is not in line with the proposition that the choice must be a *representative* of it: Feature Weighted, which is positively selected in the choice (d), has a negative selection state assigned in the ambition (g). Below, we assume that the valid ambition depicted in (j) has been selected.

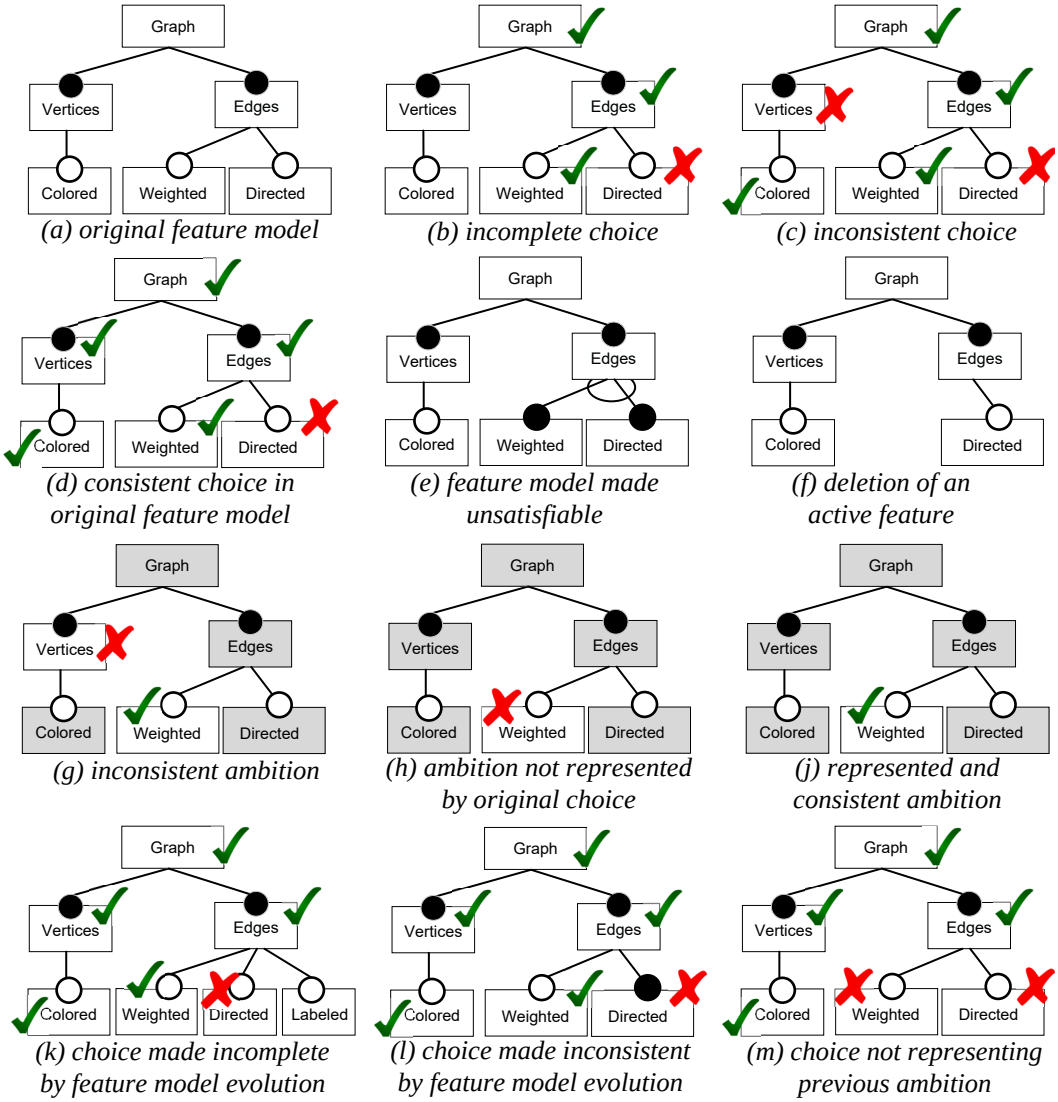
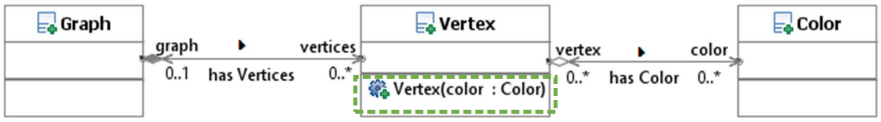
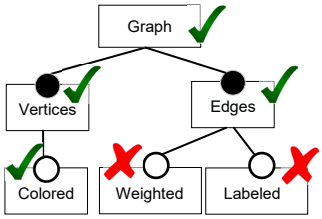


Figure 11.1: Examples of consistency violations connected to feature models, choices, and ambitions.

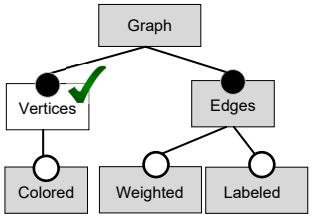
Choice not Suitable for Next Iteration. Unless the user interrupts this workflow, the DFE model continues with the next iteration based on the current choice. The choice may, however, become invalid for several reasons. First, (k) assumes that the original feature model has been extended by a new feature Labeled, for which the original choice, however, does not define a binding. Similarly, in (l), feature Directed is made mandatory, but excluded from the current choice, such that this becomes invalid for the next iteration. Last, in (m), a new user-defined choice is depicted. This choice, however, disagrees with the ambition in the binding for Weighted, whose corresponding product artifacts are still present in the workspace. Thus, it becomes necessary to re-generate the workspace contents by check-out, such that an unweighted graph is presented.



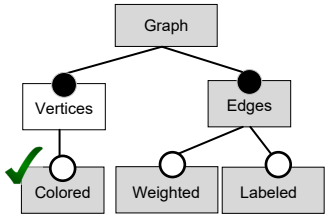
(q) product-level change



(p) choice



(r) too unspecific ambition



(s) sufficiently specific ambition

Figure 11.2: Example of a non-representative product-level change, described with a too unspecific ambition. In (q), artifacts belonging to Edges and sub-features thereof are hidden.

All the problems discussed above are related exclusively to version concepts, namely version space (i.e., feature model), choice, and ambition. There exists an additional potential source of inconsistency that is, in contrast, also connected to the product space:

Too Unspecific Ambition. Figure 11.2 depicts an iteration based on a choice that represents a colored graph (p). The product-level-change shown in (q) consists in the introduction of a new constructor to class Vertex. As constructor parameter type, the existing class Color, whose visibility is restricted by feature colored, is chosen. An attempt to use (r) as ambition for this change should fail, for the following reason: The choice should be representative for all versions in which the change can be applied. Albeit, the constructor would not be valid in variants that exclude feature colored, since class Color is not available as parameter type then. A more *specific* ambition, which adequately describes the set of versions in which the change is applicable, should be used; the most general yet sufficiently specific ambition is shown in (s).

11.1.4 General Evolution Scenario

The generalized *evolution scenario*, forming the problem statement of this chapter, is illustrated in Figure 11.3. The vertical dashed green lines divide the figure into four parts, dedicated to the phases CHECKOUT, MODIFY, COMMIT, and MIGRATE, respectively. Green arrows indicate evolution. Gray arrows represent consistency relationships and are labeled with the same numbers as the constraints introduced in Section 11.2. For abstraction, the domain model is not shown, and versioning of the feature model is not illustrated explicitly. In contrast to the negative examples shown in the previous subsection, this general problem description also considers the revision graph (whose consistency is, however, managed automatically; see Section 11.4).

Check-Out. Unless the choice is produced automatically by migration (see below), an iteration of the DFE model starts with an initial revision graph, from which a specific

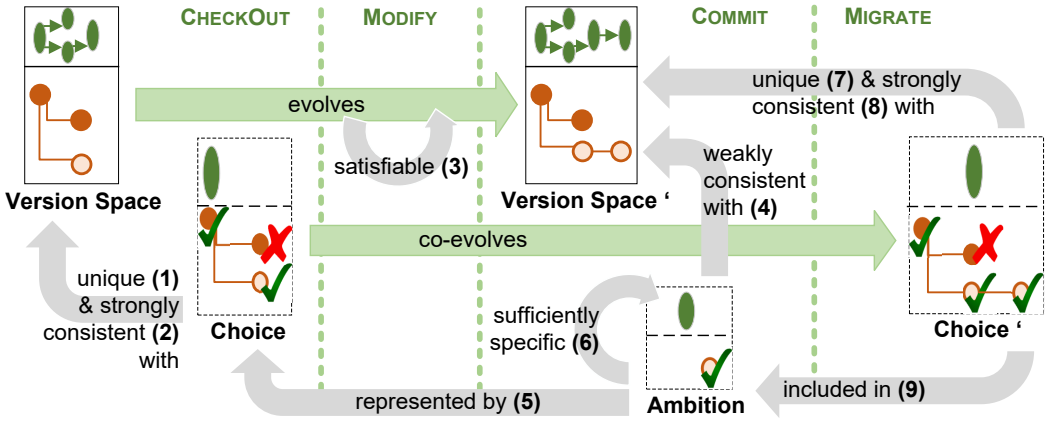


Figure 11.3: Evolution scenario generalizing the problem statement of this chapter. Based on [SW17b, Figure 3].

revision is selected. In the resulting revision of the feature model, all features need to be either selected or deselected. The version defined by the revision and feature configuration is referred to as the *choice*. The provided CHECKOUT operation has to ensure that the choice is *unique* – in particular, that no features from the feature model may remain unbound – and *strongly consistent*—all version rules defined in the feature model must hold.

Modify. In the workspace, the feature model may be edited as required, provided that it remains *satisfiable*, i.e., non-contradictory. Moreover, in order to avoid conflicts that might occur during commit, it ought to be ensured that active features must never be deleted.

Commit. The COMMIT operation produces a revised version space, whose revision graph is extended with a new revision for the performed change. The feature model may have evolved, as well. For commit, the user needs to define a *feature ambition*; the combination of feature ambition and the new revision is called *ambition*. This ambition must be checked for *weak consistency* with the new revision of the feature model, which means that there must be no contradiction to the version rules implied by the feature model (such that there exists at least one valid version within the set of versions described by the ambition). Furthermore, the ambition defined at commit time must be *represented by* the choice at check-out time, which means that there must be no contradiction between the old choice and the ambition (i.e., the ambition must neither select a feature which was inactive nor deselect a feature which was active in the old choice). Last, it must be ensured that the ambition is *sufficiently specific*, such that the applied change would have been equally applicable in all other choices included in it.

Migrate. Finally, the choice must *co-evolve* with the change: A new choice has to be established which satisfies the same constraints as the old choice, since the end of the previous iteration coincides with the start of the next iteration. To this end, we introduce a novel operation MIGRATE, which is applied transparently after commit.

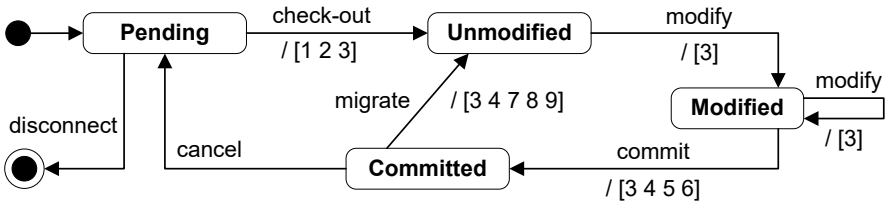


Figure 11.4: Workspace operations as transitions in a state diagram.

It essentially extends the old choice with the ambition, ensures *uniqueness* of the migrated choice and *strong consistency* with respect to the new feature model, as well as *inclusion* of the migrated choice in the ambition. Consequently, executing a new check-out under the migrated choice would result in a copy of the workspace as present at commit time.

Figure 11.4 summarizes the underlying editing cycle as state diagram. Initially, the workspace is Pending, i.e., not populated yet. On check-out, a specific version is selected from the repository. After modifying the workspace and committing, the user may either continue with the subsequent iteration in the same view, requiring to migrate the choice, or migration is canceled, triggering a transition back into state Pending. To re-populate the workspace, a new choice must be specified then in the context of a regular check-out. Operation disconnect – to be further explained in Section 12.1.3 – breaks the editing cycle.

The constraints that are guaranteed by the respective transitions are depicted as post-conditions. They refer to the constraint numbers introduced in Figure 11.3.

The figure underlines two important properties of the dynamic editing model: First, except for the initial version, and as long as choice migration is never aborted, check-out is an optional operation. Second, choice migration is a shortcut taking for granted that the user wants to perform the subsequent iteration using the current workspace as starting point, as usual in version control.

11.2 Dynamism-Aware Consistency Constraints

In this section, the consistency constraints mentioned in Figure 11.3 are elaborated on the basis of the formal foundations [2]. from Sections 8.3 and 9.2.2. Superscripts (^{ch} = check-out, ^{mo} = modify, ^{cm} = commit, ^{mi} = migrate) delineate the phases of each iteration.

11.2.1 Check-Out

In filtered editing, a choice designates an unambiguous version to describe the workspace contents to be checked-out. Therefore, unbound options must not occur.

Constraint 1— The option binding c^{ch} specified as choice during check-out must be *unique* with respect to the global option set O^{ch} defined at check-out time.

$$\forall o \in O^{ch} : (\exists (o, s) \in c^{ch} : s \in \{true, false\}) \quad \textbf{(11.1)}$$

2 Recall in particular the tuple representation (b) and conjunctive form ($\hat{b}$) of an option binding. See page 150.

In the following, we assume in all constraints that choices are unique.

Moreover, the choice must comply with the rules derived by, e.g., feature dependencies:

Constraint 2— The choice c^{ch} defined at check-out time must be *strongly consistent* with the invariants of the rule base $\mathcal{R}^{ch}$ present at check-out time.

$$\mathcal{J}^{ch}(c^{ch}) = true \tag{11.2}$$

Here, $\mathcal{J}^{ch}(c^{ch})$ denotes the evaluation[3] of the invariants $\mathcal{J}^{ch}$ under the choice c^{ch}.

11.2.2 Modify

By editing the feature model, the user indirectly modifies parts of the option set and of the rule base. It must be avoided that the user introduces rules disallowing consistent version selection in future check-outs.

Constraint 3— After each modification to the version space, the invariants of the rule base $\mathcal{R}^{mo}$ must be satisfiable, such that there exists any strongly consistent choice:

$$\exists c : \mathcal{J}^{mo}(c) = true \tag{11.3}$$

The aforementioned restriction, stating that active features must not be deleted, is beyond the means of formalization available for the base layer, and therefore not presented as an explicit constraint here. See Section 11.3.2.

11.2.3 Commit

The ambition defined at commit time describes a set of versions that should comply with the rule base: At least one choice c must exist which agrees with a^{cm} in all common option bindings ($c \Rightarrow a^{cm}$) such that all rules hold under c.

Constraint 4— An ambition a^{cm} specified during commit must be *weakly consistent* with the invariants of the rule base $\mathcal{R}^{cm}$ available at commit time:

$$\exists c : (\hat{c} \Rightarrow \hat{a}^{cm}) \wedge (\mathcal{J}^{cm}(c) = true) \tag{11.4}$$

In the version determined by the choice, a change is applied *representatively* for the ambition. Thus, there must not be any contradiction between option bindings of the check-out time choice and the commit time ambition inferred from feature selections: [4]

Constraint 5— The ambition a^{cm} must be *represented by* the check-out choice c^{ch}.

$$\forall (o, s) \in a^{cm} : (o, \neg s) \notin c^{ch}, \quad s \in \{true, false\} \tag{11.5}$$

3 Recall that $\mathcal{J}$ denotes a conjunction rather than a set of invariants.

4 This form of representativity is weaklier defined than the implication $\hat{c} \Rightarrow \hat{a}^{cm}$ required by Constraints 4 and 9. Precisely, Constraint 5 allows to commit against a newly introduced feature that was not bound in c^{ch}.

Requiring no contradictions between choice and ambition does, however, not guarantee that the modifications performed between check-out and commit are representative at product space level. To this end, it must be ensured that the performed change – here represented as a *write set* of inserted and deleted elements $E_{mod} = E_{ins} \dot{\cup} E_{del}$ – could have been equally applied in any other version contained in the ambition:

Constraint 6— The ambition a^{cm} must be *sufficiently specific* to the write set E_{mod}.

$$\forall e \in E_{mod} : \big(\forall e' \in P^{cm} : (e \xrightarrow{d} e') \Rightarrow v'({}^{\mathcal{PD}}a^{cm}) = true\big) \tag{11.6}$$

The symbols used in the equation above require further clarification. First, with $e' \in P^{cm}$, we denote any element part of the commit time product space. The premise $e \xrightarrow{d} e'$ checks whether an element of the write set *depends* on e'. Last, v' denotes the visibility of e' before commit.

The completed – yet not in general satisfying the *unique* constraint – ambition ${}^{\mathcal{PD}}a^{cm}$ is obtained by applying preferences and defaults[5] to the original ambition a^{cm}, such that $v'({}^{\mathcal{PD}}a^{cm})$ evaluates to true if and only if e' is visible in all versions included in the completed ambition. Taken together, the constraint checks whether all elements on which any inserted or deleted element depends are visible in all affected versions.

The *depends* operator $e \xrightarrow{d} e'$, where $e \in E_{mod}$ and $e' \in P$, remains to be defined upon the product space base layer. Informally, e depends on e' whenever at least one of the following conditions hold:

- e is a deleted element and e equals e'. (Intuition: Elements must be visible in order to be deletable.)
- e is an inserted element and e' contains e. (Intuition: The insertion location, i.e., the container of an inserted element, must be visible.)
- e is an inserted element and e' is cross-referenced from e. (Intuition: When an inserted element represents the applied occurrence of an existing element, the latter must be visible.)

Formally:

$$\begin{aligned} e \xrightarrow{d} e' \Leftrightarrow \big(e \in E_{del} \wedge e = e'\big) \vee \\ \big(e \in E_{ins} \wedge (e' = parent(e) \vee e' \in crossLinks(e))\big) \end{aligned} \tag{11.7}$$

11.2.4 Migrate

Transitioning to post-commit time, it is assumed that the operation MIGRATE, to be formally defined later, produces a choice that is used for the next iteration, tying on the non-disruptive workflow provided by VCS. The same workspace can be reused with the migrated choice.

Due to modifications of the rule base, the choice c^{ch} specified at check-out time may become *non-unique* with respect to the option set O^{cm}, and/or *inconsistent* with the invariants

5 In this way, a "more complete" ambition is obtained, which represents, however, the same set of product versions as a^{cm}. The options additionally included in ${}^{\mathcal{PD}}a^{cm}$ may occur in visibilities v', therefore $v'({}^{\mathcal{PD}}a^{cm})$ will less likely return $undefined$.

of the rule base $\mathcal{J}^{cm}$ at commit time. Such temporary inconsistencies are explicitly allowed in order to support feature model evolution. However, before starting the subsequent iteration, it is required that the version to be modified must be uniquely and consistently identified by c^{mi}.

Constraint 7— The option binding c^{mi} describing the choice after migration must be *unique* with respect to the commit time option set O^{cm}:

$$\forall o \in O^{cm} : (\exists (o, s) \in c^{mi} : s \in \{true, false\}) \tag{11.8}$$

Constraint 8— The migrated choice c^{mi} must remain *strongly consistent* with the invariants of the rule base $\mathcal{R}^{cm}$ available at commit time:

$$\mathcal{J}^{cm}(c^{mi}) = true \tag{11.9}$$

Apart from this, it is required that the migrated choice c^{mi} must still comply with ambition a^{cm}, which represents changes applied to the current workspace. Since all newly introduced options are mandatory to be selected or deselected for the next choice, total *inclusion* (implemented by propositional logical implication in the opposite direction) is required:

Constraint 9— An ambition a^{cm} must *include* the migrated choice c^{mi} describing the workspace contents for the subsequent iteration:

$$\hat{c}^{mi} \Rightarrow \hat{a}^{cm} \tag{11.10}$$

11.3 Consistency-Preserving Algorithms

In this section, we contribute detailed algorithms for the operations CHECKOUT, MODIFY, COMMIT, and MIGRATE, having been mentioned in Figure 11.3 and partly presented in a semi-formal way in Section 9.5.1. Taken together, they realize the dynamic filtered editing model. In addition to algorithmic descriptions, their properties are discussed and run-time considerations are made. The algorithms contain interactive statements, which have been underlined in the descriptions below.

11.3.1 Check-Out

The obvious purpose of CHECKOUT is to populate an empty (i.e., Pending, cf. Figure 11.4) workspace with a consistent product version uniquely defined by the user with the help of the abstractions of revision graph and feature model.

Algorithm 11.1 first asks the user for a revision selection. Using preferences and defaults introduced during COMMIT (cf. Table 9.1 on page 171), it is ensured that options of the selected revision as well as all predecessors are bound to $true$, whereas remaining options are bound to $false$, making the revision choice unique.

Next, the feature model, whose elements' visibilities exclusively refer to revision options, is filtered by the revision choice. In the filtered feature model, the user specifies a feature configuration; invisible options for deleted features are bound to $false$ by corresponding

defaults (cf. Table 9.2 on page 174). The effective choice c^{ch} is calculated by union of revision and feature choice, before preferences and defaults are applied to it. Next, the feature choice is checked for uniqueness and strong consistency.

After a well-formedness analysis, which has been intentionally omitted in the description provided here as it is presented in Chapter 13, the workspace is populated with filtered versions of feature and domain model. The feature choice is memorized to enable a later re-construction of the checked-out workspace.

Properties. If successful, Algorithm 11.1 transitions the workspace into state Unmodified and produces a choice both unique and strongly consistent with respect to the check-out time rule base, such that Constraints 1 and 2 are ensured. In case the user specifies a non-unique or inconsistent choice, the action is canceled and the workspace remains Pending.

Complexity Estimation. Applying preferences and defaults requires to iterate over the preference set $\mathcal{P}$ and the default set $\mathcal{D}$; the size of both is proportional to $|O|$. Ensuring Constraints 1 and 2 requires iterations over O and $\mathcal{J}$, respectively; the size of the latter is – when assuming the rule base mappings provided in Chapter 9 and that the number of *requires*/*excludes* constraints ensuing from a feature is linear – proportional to $|O|$. While filtering, each element of E is considered. The total complexity is therefore $\mathcal{O}(|O| + |E|)$.

11.3.2 Modify

The consistency of the domain model is supposed to be ensured by the respective single-version editing tools employed. Feature model editing, however, is restricted. Rather than giving a full specification of feature model editing, we explicitly define these restrictions by providing redefined algorithms for SAVEFEATUREMODEL and DELETEFEATURE.

---

procedure CHECKOUT
- $r_i^{ch} \leftarrow$ option in O_r^{ch} belonging to a selected revision i
- $c_r^{ch} := (r_i^{ch}, true)$
- ${}^{\mathcal{PD}}c_r^{ch} \leftarrow$ COMPLETE(c_r^{ch}) ▷ Algorithm 9.3
- $P_F^{ch} \leftarrow P_f^{ch}|^*_{{}^{\mathcal{PD}}c_r^{ch}}$ ▷ Filter the feature model; (**10.6**)
- EXPORT(P_F^{ch})
- $c_f^{ch} \leftarrow$ define feature configuration in the exported filtered feature model
- $c^{ch} \leftarrow c_r^{ch} \cup c_f^{ch}$
- ${}^{\mathcal{PD}}c^{ch} \leftarrow$ COMPLETE(c^{ch}) ▷ Algorithm 9.3
- **if not** $(\forall o \in O^{ch} : (\exists (o, s) \in^{\mathcal{PD}} c^{ch} : s \in \{true, false\}))$ **then** ▷ Constraint 1
  - **return** error "Choice is not unique."
- **else if not** $(\mathcal{J}^{ch}({}^{\mathcal{PD}}c^{ch}) = true)$ **then** ▷ Constraint 2
  - **return** error "Choice is not strongly consistent."
- $P_D^{ch} \leftarrow P_d^{ch}|^*_{{}^{\mathcal{PD}}c^{ch}}$ ▷ Filter the domain model; (**10.6**)
- EXPORT(P_D^{ch})
- Memorize c_f^{ch} for the subsequent commit

Algorithm 11.1: Consistency-preserving CHECKOUT. From [SW17b, Algorithm 1].

```
procedure SAVE(P_F^mo)
    R_f^mo ← rule base derived from (P_F^mo)                    ▷ Table 9.2
    if not ∃c : J_f^mo(c) = true then                           ▷ Constraint 3
        return error "Feature model is not satisfiable."
    else
        Persist P_F^mo in its current state
```

Algorithm 11.2: Redefined operation SAVEFEATUREMODEL in feature model editor.

```
procedure DELETEFEATURE(D)
    o_D ← option belonging to feature D to be deleted
    c_f^ch ← feature choice memorized during preceding check-out or migration
    if (o_D, true) ∈ c_f^ch then
        return error "Cannot delete feature active in current choice."
    else
        for all C ∈ D.children do
            if DELETEFEATURE(C) ≠ success then
                Undo all modifications related to D.children
                return error "Error during deletion of child C."
        set the deleted flag of D to true
        D_f^mo := D_f^mo ∪ {(D, false)}                         ▷ Table 9.2
```

Algorithm 11.3: Redefined operation DELETEFEATURE in feature model editor.

Feature Model Editing. Constraint 3 must be enforced; otherwise, no consistent variant can be specified in subsequent check-outs. Rather than checking for feature model satisfiability in the course of COMMIT, it is preponed to MODIFY in order to give feedback to the user as early as possible. To this end, we redefine the SAVEFEATUREMODEL operation of the feature model editor in Algorithm 11.2 in a way that only satisfiable feature models can be persisted in the workspace.

Feature Deletion. Furthermore, *deletion* of features in the workspace version of the feature model is redefined (see Algorithm 11.3): First, the operation is only applicable to features bound to $false$ in the current choice; otherwise, the feature model would become unsatisfiable, or *choice migration* (see Section 11.3.4) would transfer the positive selection state to the choice to be derived for the next iteration, where the deleted feature and corresponding realization artifacts are supposed to be hidden.

Second, rather than persistently deleting a feature, it is merely hidden from the user's display and thus not available in the current and future revisions of the editable feature model (cf. Section 10.7.1). Nevertheless, its feature option, which still may occur in visibilities of domain model elements, remains. To maintain *uniqueness* of future choices, a default is introduced. This default also affects version rules referencing the feature; their evaluation is automated for future satisfiability checks in the feature model.

In order to maintain the hierarchical consistency of the feature model, feature deletion is recursively applied to all child features.

Properties. Constraint 3 is actively enforced by Algorithm 11.2. Whenever the SAVE operation has been applied successfully, the workspace enters (or remains in) state Modified.

By approximating the satisfiability of the feature model in conjunction with the current choice, redefined feature deletion indirectly affects Constraint 8 inasmuch as certain situations in which migration would fail are prevented. Nevertheless, the satisfiability of neither Constraint 8 nor any other constraint can be guaranteed.

Complexity Estimation. The satisfiability check described by Constraint 3 performed after each save is *NP-complete*.

Given the worst case, where the entire feature tree is attempted to be deleted, the run-time of feature deletion is linear with the number of features: $\mathcal{O}(|O_f|)$.

11.3.3 Commit

A consistency-preserving COMMIT operation is formalized in Algorithm 11.4. The revision graph is handled automatically, introducing a new revision option along with a preference and a default ensuring that a single revision selection will yield unique and consistent revision choices in future. By using the latest revision as reference point, a linear version history is enforced. Through repeated application of preferences of the form (r_i, r_{i+1}), the selection is propagated back until the initial revision. Defaults, having a lower priority, are applied to unbound revision options thereafter, such that remaining revisions are deselected.

Besides, the user specifies a *feature ambition*. In case the domain model is unmodified[6], however, this step can be skipped. It is ensured by corresponding checks that feature ambitions must be weakly consistent with the rule base (Constraint 4) and represented by the previous choice (Constraint 5).

Next, the latest version of the workspace state (taking into consideration the logical choice used for check-out) is reconstructed and differentiated with its commit time version. Based on the deduced difference, it is now ensured that the specified ambition is sufficiently specific to the performed change (Constraint 6).

In case all constraints are passed, inserted elements are appended to the product space with the help of the generic asymmetric two-way raw MERGE operator (cf. Algorithm 10.4 on page 213). Next, visibilities of inserted and deleted elements are updated as defined by Algorithm 11.5. For visibility updates applied to the feature model, the new revision option serves as ambition. For the domain model, a conjunction of the new revision option and the chosen feature ambition is used, such that *presence conditions* are *implicitly versioned* (design decision **D17**). Transparently, the change space optimization as presented in Section 9.6.2 is applied.

Properties. If successful, Algorithm 11.4 transitions the workspace into the state Committed, while ensuring Constraints 4, 5, and 6 for the specified ambition.

Otherwise, the workspace remains in state Modified; in this case, the user may re-attempt the commit with a different ambition.

6 Such a situation can be detected by an a-priori comparison, e.g., based on hashing. See Section 14.5.2.

Complexity Estimation. Reproducing the checked-out workspace and updating the visibilities both imply a run-time proportional to the size of the product space, i.e., $|P|$. Constraint 5 checks a maximum of $|O|$ options.

Assuming that the number of elements that depend ($\xrightarrow{d}$) on a modified element is constant, we obtain a complexity of $|P|$ for Constraint 6.

Run-time is, in theory, dominated by the *NP-complete* satisfiability check issued during the evaluation of Constraint 4. In practice, however, we expect that the Filter and Export operations applied to the domain model, which is typically considerably larger than the feature model, become the bottleneck.

---

procedure COMMIT
- $c_f^{ch} \leftarrow$ feature choice memorized during preceding check-out or migration
- $r_i \leftarrow$ option of most recently committed revision i (*head*)
- $c^{ch} \leftarrow c_f^{ch} \cup \{(r_i, true)\}$
- ${}^{\mathcal{PD}}c^{ch} \leftarrow$ COMPLETE(c^{ch}) ▷ Algorithm 9.3
- $i+1 \leftarrow$ new revision, successor of i, with user-specified details (commit message, etc.)
- $r_{i+1} \leftarrow$ new revision option for revision $i+1$
- $O_r^{cm} \leftarrow O_r^{ch} \cup \{r_{i+1}\}$
- $\mathcal{J}_r^{cm} \leftarrow \mathcal{J}_r^{ch} \wedge (r_{i+1} \Rightarrow r_i)$ ▷ Table 9.2
- $\mathcal{P}_r^{cm} \leftarrow \mathcal{P}_r^{ch} \cup \{(r_i, r_{i+1})\}$ ▷ Table 9.2
- $\mathcal{D}_r^{cm} \leftarrow \mathcal{D}_r^{ch} \cup \{(r_{i+1}, false)\}$ ▷ Table 9.2
- $P_F^{ch} \leftarrow P_f^{ch}|^*_{\mathcal{PD}_{c_r^{ch}}}$ ▷ Reproduce latest revision of feature model
- $P_D^{ch} \leftarrow P_d^{ch}|^*_{\mathcal{PD}_{c^{ch}}}$ ▷ Reproduce latest revision of selected variant of domain model
- $P_F^{cm} \leftarrow$ IMPORT current workspace version of the feature model
- $P_D^{cm} \leftarrow$ IMPORT current workspace version of the domain model
- **if** $P^{cm} = P^{ch}$ **then** ▷ Domain model is unmodified
  - $a_f^{cm} \leftarrow \emptyset$ ▷ No feature ambition is needed
- **else**
  - $a_f^{cm} \leftarrow$ define feature ambition in the current workspace version of the feature model
- $a^{cm} \leftarrow a_f^{cm} \cup \{(r_{i+1}, true)\}$
- **if not** $(\exists c : (\hat{c} \Rightarrow \hat{a}^{cm}) \wedge (\mathcal{J}^{cm}(c) = true))$ **then**
  - **return** error "Ambition is not weakly consistent." ▷ Constraint 4
- **else if not** $(\forall (o,s) \in a^{cm} : (o, \neg s) \notin c^{ch})$ **then**
  - **return** error "Ambition is not represented by choice." ▷ Constraint 5
- $match \leftarrow$ MATCH(P^{cm}, P^{ch}) ▷ Algorithm 10.2
- $diff \leftarrow$ DIFF$(match)$ ▷ Algorithm 10.3
- $E_{mod} \leftarrow$ all inserted or deleted elements according to $diff$
- ${}^{\mathcal{PD}}a^{cm} \leftarrow$ COMPLETE(a^{cm}) ▷ Algorithm 9.3
- **if not** $(\forall e \in E_{mod} : (\forall e' \in P_D^{cm} : (e \xrightarrow{d} e') \Rightarrow v'({}^{\mathcal{PD}}a^{cm}) = true))$ **then**
  - **return** error "Ambition is not sufficiently specific to the change." ▷ Constraint 6
- MERGE$(P^{cm}, P^{ch}, diff, match)$ ▷ Algorithm 10.4
- $cs \leftarrow$ new change set
- UPDATEVISIBILITIES$(diff_f, a_r^{cm}, cs)$ ▷ Algorithm 11.5
- UPDATEVISIBILITIES$(diff_d, a^{cm}, cs)$ ▷ Algorithm 11.5

Algorithm 11.4: Consistency-preserving COMMIT. From [SW17b, Algorithm 2].

procedure UPDATEVISIBILITIES(diff, a, cs)
  $\Delta \leftarrow$ new change under cs connected to ambition a
  $o_\Delta \leftarrow$ new change option for Δ
  $O^{cm}_\Delta \leftarrow O^{cm}_\Delta \cup \{o_\Delta\}$
  $\mathcal{J}^{cm}_\Delta \leftarrow \mathcal{J}^{cm}_\Delta \wedge o_\Delta \Leftrightarrow \hat{a}$
  $\mathcal{P}^{cm}_\Delta \leftarrow \mathcal{P}^{cm}_\Delta \cup \{(o_\Delta, \hat{a})\}$
  for all $e_{ins} \in \mathit{diff}$`.insertions` **do**
    $v_{ins} \leftarrow \Delta$ ▷ Initialize visibility v_{ins} of element e_{ins}
  for all $e_{del} \in \mathit{diff}$`.deletions` **do**
    $v_{del} \leftarrow v_{del} \wedge \neg\Delta$ ▷ Update visibility v_{del} of element e_{del}

Algorithm 11.5: Visibility update using change space optimization.

11.3.4 Migrate

The operation MIGRATE prepares the workspace choice for the subsequent iteration, proceeding under the assumption that the user prefers to stay in the current view. Unlike CHECKOUT and COMMIT, this operation is not triggered explicitly by the user but automatically after COMMIT. Conversely, it makes the subsequent CHECKOUT optional, tying on the unobtrusive revision control workflow.

Algorithm 11.6 considers options that remain unbound in the choice to be migrated. If a corresponding option has been bound in the ambition, the binding is transferred to the choice. Otherwise, by the COMPLETE operator, preferences and defaults are triggered as far as applicable, with the aim to make c^{mi} unique. As a "last resort", a binding state of remaining unbound options is obtained non-deterministically. Since the new option has been ignored in the ambition, there cannot exist any reference to it in updated visibilities. Therefore, it is immaterial for the subsequent choice whether or not the option is selected. At this point, it is not known how new (and still unbound) features will be incorporated in

procedure MIGRATE
  for $(o, s^{mi}) \in a^{mi}$ **do**
    if $\nexists (o, s_x) \in c^{mi} : s_x \in \{true, false\}$ **then**
      $c^{mi} \leftarrow c^{mi} \cup \{(o, s^{mi})\}$
  $c^{mi} \leftarrow$ COMPLETE(c^{mi}) ▷ Algorithm 9.3
  for $o \in O^{cm}$ **do**
    if $\nexists (o, s_x) \in c^{mi} : s_x \in \{true, false\}$ **then**
      $s^{mi} \leftarrow$ user selection for o
      if $s^{mi} = undefined$ **then**
        return error "Operation was canceled by the user."
      else
        $c^{mi} \leftarrow c^{mi} \cup \{(o, s^{mi})\}$
  if not $\mathcal{J}^{cm}(c^{mi}) = true$ **then** ▷ Constraint 8
    return error "Cannot migrate to a consistent choice."
  else
    Memorize c^{mi}_f for the subsequent commit ▷ Obviate check-out

Algorithm 11.6: Consistency-preserving MIGRATE. From [SW17b, Algorithm 3].

the next iteration. Therefore, the user may choose among the set of choices describing the current workspace contents equivalently.

Properties. If migration succeeds, a *strongly consistent choice* (Constraint 8) is actively enforced by the algorithm. Theorems 11.1 and 11.2 prove that Constraints 7 and 9 are satisfied, respectively. Conjecture 11.1 explicitly refers to the relationship between MIGRATE and CHECKOUT.

Notice that there are three possible causes for failure of this operation. First, there may be no correct solution regardless of the user selections performed[7]; an example is provided in Section 11.5.3. Second, the user might introduce a contradiction although a different selection would have provided a correct migrated choice[8]. Third, the user may cancel intentionally.

If migration succeeds, the workspace immediately enters state Unmodified. Otherwise, entering state Pending triggers an exceptional check-out, forcing the user into specifying a new choice.

Theorem 11.1— After having applied MIGRATE successfully, Constraint 7 is satisfied.

Proof. The algorithm iterates over all options in O^{cm}, which equals O^{mi} as no options can be introduced between commit and migrate by any operation. In each iteration, either $true$ or $false$ are assigned to bindings missing in c^{mi}. Therefore, unless the migration is aborted, c^{mi} is *complete* (as required by Constraint 7). ■

Theorem 11.2— After having applied MIGRATE successfully, Constraint 9 is satisfied.

Proof. Being its descendant, c^{mi} includes c^{ch}. Moreover, a^{cm} is weakly consistent with c^{ch} (cf. Constraint 5). Thus, no contradictions exist between c^{mi} and a^{cm}. Bindings for missing options are transferred from the ambition, or if not applicable, in a way that does not contradict with any ambition binding. Altogether, the migrated choice c^{mi} is included in the ambition a^{cm} (such that $\hat{c}^{mi} \Rightarrow \hat{a}^{cm}$, as required by Constraint 9). ■

Conjecture 11.1— It is taken as premise that migration has been applied successfully, yielding the migrated choice. c^{mi}. Then, an explicit check-out using c^{mi} as choice produces the same workspace content as already present.

Comment. This conjecture cannot be (dis)proved without making assumptions about the bijectivity of the operations IMPORT and EXPORT, which we cannot guarantee here. In fact, the conjecture might be disproved as soon as it comes to product-level well-formedness violations (see Chapter 13) introduced during the iteration preceding the migrate operation. (Then, it depends on the resolution decisions whether the conjecture holds.)

7 In such a case, newly introduced feature model rules prevent the product version available in the workspace from being reproduced by further check-outs. The performed modifications are, however, valid for different versions included in the ambition.

8 In SuperMod, all missing bindings are requested from the user at the same time in an interactive dialog, where validation is applied in the background. This helps preventing such deadlock situations.

Complexity Estimation. Assuming that the option bindings underlying choices and ambitions have been implemented with an associative data structure, iterating over all options requires a maximum run-time proportional to $|O|$. Preferences and defaults, whose number is proportional to $|O|$, are taken into account exclusively for newly introduced options $o_{new} \in O_{new}$. Therefore, when neglecting user interaction, total complexity is $\mathcal{O}(|O| \cdot |O_{new}|)$.

11.4 Automatic and Consistent Revision Graph Management

Above, it has been repeatedly claimed that the consistency of the revision graph – faded out in all examples – is guaranteed automatically, such that the user is not accosted with constraint violations in this dimension. In this section, we supplement proof for the satisfaction of the constraints defined in Section 11.2 by the historical dimension in isolation.

The remainder of this section is structured by phases, of which MODIFY has been omitted since it does not affect the revision graph.

11.4.1 Check-Out

It has to be proved that the choice inferred from the selection of a single revision in Algorithm 11.1 satisfies the check-out time consistency constraints.

Theorem 11.3— A revision choice derived at check-out is *unique* according to Constraint 1.

Proof. Preferences and defaults are applied in advance to filtering. According to Table 9.1, a default of the form $(r_i, false)$ is introduced for each revision i, such that no unbound revision option remains after having applied all defaults. ■

Theorem 11.4— A revision choice derived at check-out is *strongly consistent* as required by Constraint 2.

Proof. According to Table 9.1, there are two types of invariants to be potentially violated: initial revision invariants (r_0) and predecessor invariants ($r_{i+1} \Rightarrow r_i$).

Except for the option r_i of the user-selected revision, all other revision options are bound by preferences or defaults. In Algorithm 11.4, it is ensured that together with each invariant $r_{i+1} \Rightarrow r_i$, a preference (r_i, r_{i+1}) is created. Through repeated application of this preference, after a revision option r_i has been selected, all predecessor revisions are bound positively. For no predecessor of r_i, a negative binding will be created since defaults have a lower priority than preferences. Therefore, all predecessor invariants are satisfied.

Given the premise of a linear and acyclic revision graph, repeated application of predecessor preferences will propagate to r_0, regardless of which revision has been selected. Therefore, $(r_0, true)$ will occur in every binding derived this way, such that the initial revision constraint r_0 is satisfied. ■

11.4.2 Commit

During COMMIT (cf. Algorithm 11.4), a new revision with option r_{i+1} is introduced for the successor of revision i. We prove that the derived revision ambition $\{(r_{i+1}, true)\}$ satisfies the constraints associated with the commit phase.

Theorem 11.5— A revision ambition derived at commit is *weakly consistent* as defined in Constraint 4.

Proof. In order to be weakly inconsistent, it would be required that $\{(r_{i+1}, true)\}$ contradicts with any invariant in $\mathcal{J}_r^{cm}$. The only invariant in which r_{i+1} can appear is the newly introduced $r_{i+1} \Rightarrow r_i$. Though, $(r_i, false) \notin a_r^{cm}$, thus weak consistency is given. ■

Theorem 11.6— A revision ambition derived at commit is *represented* by the check-out time choice as required by Constraint 5.

Proof. Not yet existing at check-out time, r_{i+1} is unbound in the revision choice c_r^{ch}, such that no contradiction with $\{(r_{i+1}, true)\}$ can occur. ■

Theorem 11.7— A revision ambition derived at commit is *sufficiently specific* to describe the historical component of a workspace change; see Constraint 6.

Proof. We represent the visibilities v of all elements as conjunctions $v_f \wedge v_r$. Furthermore, we assume that all elements e' on which modified elements $e \in E_{mod}$ may depend, have passed the choice: $v'_r(c^{ch}) = true$. After applying option binding completion, it is granted that ${}^{\mathcal{PD}}a_r^{cm} \supset c_r^{ch}$. As a consequence, $v'_r({}^{\mathcal{PD}}a_r^{cm}) = true$ for all v'_r. ■

11.4.3 Migrate

During MIGRATE (cf. Algorithm 11.4), the binding tuple $(r_{i+1}, true)$ is transferred from the ambition to the choice. We first consider the common case that the selected revision equals the *head*; for the subsequent proofs, we may therefore presume $c_r^{mi} = c_r^{ch} \cup a_r^{cm}$, hence $\hat{c}_r^{cm} \Rightarrow \hat{a}_r^{cm}$.

Theorem 11.8— The migrated revision choice is *unique*; cf. Constraint 7.

Proof. c_r^{ch} is unique with respect to O_r^{cm} except for the only new option r_{i+1}, which is, however, bound in the ambition a_r^{cm} and therefore transferred from there. Thus, $c_r^{mi} = c_r^{ch} \cup a_r^{cm}$ is unique. ■

Theorem 11.9— The migrated revision choice is *strongly consistent*; cf. Constraint 8.

Proof. c_r^{ch} is consistent with respect to $\mathcal{J}_r^{ch}$. We may assume that $\mathcal{J}_r^{cm} = \mathcal{J}_r^{ch} \wedge (r_{i+1} \Rightarrow r_i)$. From the choice, we know that $r_i = true$. From the ambition, $r_{i+1} = true$. Taken together $(true \Rightarrow true)$, the new predecessor invariant is fulfilled. ■

Theorem 11.10— The migrated revision choice *is included in* the revision ambition as demanded by Constraint 9.

Proof. Since the binding tuple $(r_{i+1}, true)$ is transferred from the ambition to the migrated choice, $a_r^{cm} \subset c_r^{mi}$. ■

In case the check-out time choice did not equal the latest revision, however, MIGRATE will definitely fail because $r_{i+1} \Rightarrow r_i$ is violated due to the negative selection state of r_i set by the revision default. In this case, an explicit CHECKOUT, including a consistent revision selection, is enforced (cf. Fig. 11.4).

11.5 Examples

We exemplify the consistency constraints and the consistency-preserving algorithms in three ways. First, in Section 11.5.1, a cut-out of an editing history, comprising four iterations, is investigated, where the editing model imposes significant benefits. In Section 11.5.2, we refer back to the consistency violations motivated in Section 11.1.3, explaining how they are detected formally. A more sophisticated example demonstrating circumstances under which the migrate operation may fail is supplied in Section 11.5.3.

11.5.1 Unobtrusive and Consistent Dynamic Filtered Editing

Figure 11.5 continues the Graph example and illustrates the user-visible artifacts of four successful iterations of the editing model. Once more, to reduce complexity, the revision graph has been entirely faded out.

Step 1. In the first revision, a colored, labeled, unweighted graph is chosen. Feature Labeled remains to be realized, which is done during the MODIFY phase of the first iteration. The change is committed against the existing feature Labeled. Since the feature model is not modified, no new feature bindings need to be added during choice migration.

Step 2. Since we intend to stay in the current workspace view, an explicit check-out is not necessary, such that the specification of a feature configuration is skipped. We concurrently modify the domain model and the feature model: In the latter, a new XOR feature group Direction, organizing the mutually exclusive features Directed and Undirected, is introduced. The depicted domain model changes refer to the realization of the feature Directed by two UML associations representing the source and the target vertices of an edge, respectively. During MIGRATE, bindings for the new features are inferred partly: The positive selection of Directed is transferred from the ambition; for the superordinate feature Direction, a preference is applicable, setting it to true. The binding for Undirected remains to be defined by the user[9], who decides to cancel this step anyway, since the current workspace view does not represent a suitable variant in which the subsequently intended change can be made.

Step 3. In the beginning of iteration three, therefore, a feature configuration is specified explicitly during CHECKOUT. It represents an uncolored, unweighted, labeled, and undirected graph. Let us assume that the feature Colored shall be abandoned. The

9 Actually, only a negative binding comes into question, as negative selection would violate the rule base. This, however, cannot be recognized by the MIGRATE algorithm in its form presented above.

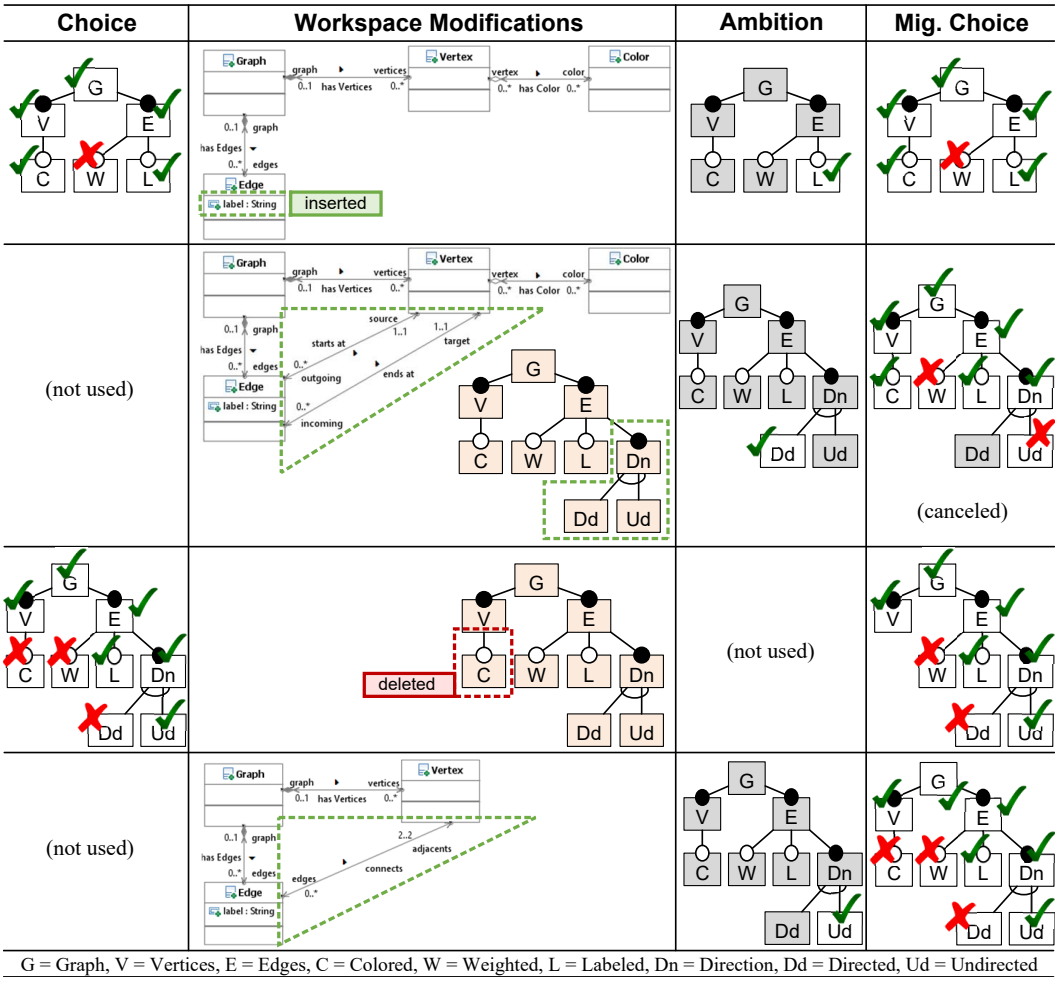


Figure 11.5: Four iterations of unobtrusive and consistent dynamic filtered editing.

corresponding deletion during MODIFY is allowed as the feature is neither mandatory nor selected in the current configuration, nor is it required by another feature due to a cross-tree relationship. Since no change is performed to the domain model, no ambition needs to be specified[10]. For the same reason, the migrated choice equals the check-out choice (when confining to the feature model).

Step 4. We use the current workspace view for the fourth and last iteration. Within this, we commit the realization of feature Undirected: a UML association referencing two vertices part of an undirected edge.

Observations. This cut-out illustrates three properties of the editing model that, altogether, ensure an unobtrusive workflow. First, in two of four cases, no explicit check-out is required thanks to the MIGRATE operation. In case, however, a different workspace view is desired by the user, migration can be canceled at any time. Second, feature

10 For check-out, a feature configuration needs to be defined though. By *feature model transactions*, introduced in Section 11.6, the user could be relieved from this task.

deletion behaves consistently by also hiding corresponding realization artifacts from the user. Third, in case the domain model is not modified, no feature ambition is requested, such that the user actions necessary during COMMIT are reduced to defining a commit message.

11.5.2 Consistency Violations Formally Revisited

Let us now return to the consistency violations informally explained in Section 11.1.3 in order to re-formulate them in terms of negative evaluations of Constraints 1 until 9 introduced in Section 11.2. Subscripts refer to corresponding sub-figures of Figure 11.1.

Non-Unique or Inconsistent Choice. The feature model depicted in Figure 11.1(a) can be mapped to the following option set:

$$O^{ch}_{(a)} = \{o_G, o_V, o_E, o_C, o_W, o_D\}$$

Furthermore, the following invariants are derived from it:

$$\mathcal{J}^{ch}_{(a)} = (o_G) \wedge (o_V \Leftrightarrow o_G) \wedge \ldots \wedge (o_C \Rightarrow o_V) \wedge \ldots$$

The choice expressed by (b) is mapped to:

$$c^{ch}_{(b)} = \{(o_G, true), (o_E, true), (o_W, true), (o_D, false)\}$$

Then, Constraint 1 is violated as neither (o_V, s_V) nor $(o_C, s_C) \in O^{ch}_{(a)}$, where $s_V, s_C \in \{true, false\}$ (cf. (**11.1**)).

Choice (c) is internally represented as:

$$c^{ch}_{(c)} = \{(o_G, true), (o_V, false), (o_E, true), (o_C, true), (o_W, true), (o_D, false)\}$$

Due to the contradiction

$$(o_V = false) \wedge (o_C = true) \wedge (o_G) \wedge (o_V \Leftrightarrow o_G) \wedge (o_C \Rightarrow O_V)$$

the application $\mathcal{J}^{ch}_{(a)}(c^{ch}_{(c)})$ evaluates to false (cf. (**11.2**)), such that Constraint 2 is violated.

The unique and consistent choice selected by the user in (d) is mapped to:

$$c^{ch}_{(d)} = \{(o_G, true), (o_V, true), (o_E, true), (o_C, true), (o_W, true), (o_D, false)\}$$

Disallowed Feature Model Modification. Algorithm 11.2 would actively prevent the feature model depicted in Figure 11.1(e) from being saved. Since the propositional logical representation is a contradiction, Constraint 3 would be violated.

$$\mathcal{J}^{mod}_{(e)} = o_G \wedge \ldots \wedge (o_E \Leftrightarrow o_G) \wedge \neg(o_W \wedge o_D) \wedge (o_W \Leftrightarrow o_E) \wedge (o_D \Leftrightarrow o_E)$$

Similarly, the deletion of feature Weighted in (f) is disallowed according to Algorithm 11.3 because $(o_W, true) \in c^{ch}_{(d)}$.

Non-Represented or Inconsistent Ambition. We still assume (a) as feature model and $c^{ch}_{(d)}$ as active choice, and consider the ambitions depicted in Figure 11.1(g, h):

$$a^{cm}_{(g)} = \{(o_V, false), (o_W, true)\}$$

$$a^{cm}_{(h)} = \{(o_W, false)\}$$

On the one hand, according to (**11.4**), there must exist choice c inside $a^{cm}_{(g)}$ that satisfies $\mathcal{J}^{ch}_{(a)}$. Such c cannot exist due to the following contradiction:

$$o_G \wedge (o_G \Leftrightarrow o_V) \wedge (o_V = false)$$

Therefore, Constraint 4 fails here.

On the other hand, $(o_W, false) \in a^{cm}_{(h)}$ and $(o_W, true) \in c^{ch}_{(d)}$ contradict (**11.5**), such that a violation of Constraint 5 indicates that the ambition specified in (h) is not represented by the choice defined in (d).

Too Unspecific Ambition. Figures 11.2(p) and (r) externally represent the following choice and ambition:

$$c^{ch}_{(p)} = \{(o_G, true), (o_V, true), (o_E, true), (o_C, true), (o_W, false), (o_L, false)\}$$

$$a^{cm}_{(r)} = \{(o_V, true)\}$$

After applying the COMPLETE operator, we obtain:

$${}^{\mathcal{PD}}a^{cm}_{(r)} = \{(o_V, true), (o_G, true), (o_E, true)\}$$

In order to reason about the representativity of the performed product-level change, we need to explicitly introduce three elements into our formal representation: e_V and e_C represent the UML class Vertex and Color, respectively, whereas $e_{VC1} \in E_{ins}$ corresponds to the constructor inserted into the list of operations of e_V. We start from the premise that the existing elements carry the following visibilities before the commit is made effective:

$$v^{ch}_V := o_V; \quad v^{ch}_C := o_C; \quad v^{ch}_{VC1} := true$$

Furthermore, we assume that the dependencies $e_{VC1} \xrightarrow{d} e_V$ and $e_{VC1} \xrightarrow{d} e_C$ have been inferred from the higher-level model structure; class Vertex contains the constructor, which in turn has a cross-link to its parameter type, class Color.

Formally, we can detect a violation of Constraint 6 by means of the following conclusions being deduced from (**11.6**):

$$v_V({}^{\mathcal{PD}}a^{cm}_{(r)}) = true \wedge v_C({}^{\mathcal{PD}}a^{cm}_{(r)}) = true$$

Albeit, when evaluating $v_C({}^{\mathcal{PD}}a^{cm}_{(r)})$ under three-valued logic, we obtain $undefined$ since o_C is unbound in ${}^{\mathcal{PD}}a^{cm}_{(r)}$. This suffices to invalidate (**11.6**).

Migrated Choice not Suitable for Next Iteration. When being compared to (a), the feature model underlying the choice shown in Figure 11.1(k) contains an additional option Labeled, internally represented by o_L:

$$O^{mi}_{(k)} = \{o_G, o_V, o_E, o_C, o_W, o_D, o_L\}$$

The current choice (d) does not include a binding for the new option for feature Labeled: $(o_L, s_L) \notin c^{mi}_{(d)}$, where $s_L \in \{true, false\}$. This is, however, required by (**11.8**). Therefore, Constraint 7 is violated.

For (l), we use $O_{(l)} = O_{(a)}$, but the rule base $\mathcal{J}_{(l)}$ contains an additional rule taking into account that Directed was made mandatory:

$$\mathcal{J}_{(l)} = \mathcal{J}_{(a)} \wedge (o_E \Leftrightarrow o_D)$$

Due to the contradiction $(o_E = true) \wedge (o_D = false) \wedge (o_E \Leftrightarrow o_D)$, the application of $\mathcal{J}_{(l)}(c^{mi}_{(l)})$ evaluates to false (**11.9**), thus Constraint 8 is violated.

Last, (**11.10**), when applied to this example, requires that $\hat{c}^{mi}_{(m)} \Rightarrow \hat{a}^{cm}_{(m)}$, thus:

$$o_G \wedge o_V \wedge o_E \wedge o_C \wedge \neg o_W \wedge \neg o_D \Rightarrow o_W \qquad (\textbf{11.11})$$

By this contradiction, a violation of Constraint 9 is detected formally.

11.5.3 Inapplicable Migration

We conclude the examples section by giving a (maliciously constructed and minimalistic) scenario where choice migration actually fails, such that it is not possible for Algorithm 11.6 to terminate successfully. In particular, the produced choice is both unique and including the ambition, but has become strongly inconsistent with the evolved rule base $\mathcal{J}^{mo}$. The descriptions given below are illustrated by Figure 11.6.

The feature model depicted in sub-figure (u) is mapped to:

$$O^{ch}_{(u)} = \{f_R, f_A, f_B\}$$

$$\mathcal{J}^{ch}_{(u)} = f_R \wedge f_A \wedge (f_A \Leftrightarrow f_R) \wedge (f_B \Rightarrow f_R)$$

$$\mathcal{P}^{ch}_{(u)} = \{(f_A, f_R)\}$$

$$\mathcal{D}^{ch}_{(u)} = \{(f_R, true)\}$$

The choice depicted in (v) is unique and strongly consistent:

$$c^{ch}_{(v)} = \{(f_R, true), (f_A, true), (f_B, true)\}$$

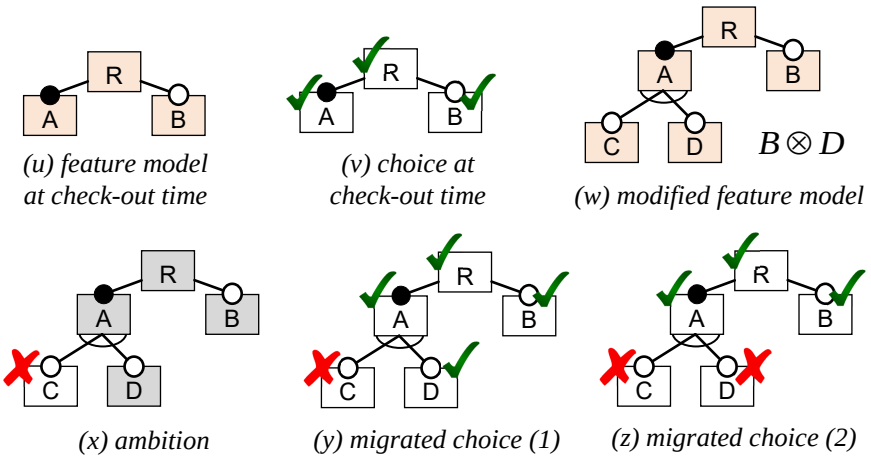


Figure 11.6: Example of inapplicability of the choice migration operation.

After the following changes, the feature model (w) remains satisfiable:

$$O^{mo}_{(w)} = \{f_R, f_A, f_B, f_C, f_D\}$$

$$\mathcal{J}^{mo}_{(w)} = \mathcal{J}^{ch}_{(u)} \wedge (f_C \Rightarrow f_A) \wedge (f_D \Rightarrow f_A) \wedge (f_C \vee f_D \Leftrightarrow f_A) \wedge \neg(f_C \wedge f_D) \wedge \neg(f_B \wedge f_D)$$

Preferences and defaults remain unaffected.

During commit, the following weakly consistent ambition (x), which is also represented by the choice, is defined by the user:

$$a^{cm}_{(x)} = \{(f_C, false)\}$$

Then, we apply the MIGRATE operation as specified in Algorithm 11.6. Bindings for the new options o_C and o_D need to be added. While binding $(o_C, false)$ can be inferred from the ambition, o_D remains to be user-defined as no preference or default are applicable either. Based upon the two possible selections available to the user, the migrated choices depicted in (y) and (z) can be created.

$$c^{mi}_{(y)} = \{(f_R, true), (f_A, true), (f_B, true), (f_C, false), (f_D, true)\}$$

$$c^{mi}_{(z)} = \{(f_R, true), (f_A, true), (f_B, true), (f_C, false), (f_D, false)\}$$

Both migrated choices are unique and include $a^{cm}_{(x)}$. Though, both $c^{mi}_{(y)}$ and $c^{mi}_{(z)}$ are strongly inconsistent with the invariants of the migrated rule base $\mathcal{J}^{mo}_{(w)}$. In the first case, the *excludes* relationship $\neg(o_B \wedge o_D)$ is violated, whereas in the second case, both mutually exclusive children C and D of the XOR group are deselected.

The example demonstrates that cases in which the migrate operation cannot be applied successfully do exist, however, they require a certain amount of "viciousness". But even when assuming that this problem will occur in a more or less significant ratio of commits, the consequences are not too severe. In such cases, a check-out is issued automatically in order to re-populate the workspace.

11.6 Generalized Editing Model

In the beginning of this chapter, the commonalities and differences between *static* and *dynamic filtered editing* have been discussed. After having formalized the consistency constraints implied by the dynamic editing model, we revisit this comparison. Figure 11.7 illustrates the different optional and mandatory phases of the respective iterations.

This section sketches – without providing formal definitions or proof – how the editing model assumed so far can be generalized in order to support static filtered editing as well as blended forms of the editing models.

11.6.1 Static Filtered Editing

When compared to the dynamic way assumed by the algorithms, static filtered editing requires only to check a subset of the constraints investigated here. This is due to the missing evolution of the feature model, as well as due to the lack of the operation MIGRATE; a check-out is required in advance to each iteration. Furthermore, the order of consistency checks is different because the ambition is selected during check-out already.

The algorithms presented in Section 11.3 can be adjusted for SFE as follows:

– Ambition selection – including the automated revision graph management – is moved from COMMIT to CHECKOUT, more precisely to after filtering the feature model. Constraint 4 (weak ambition consistency) is preponed accordingly.

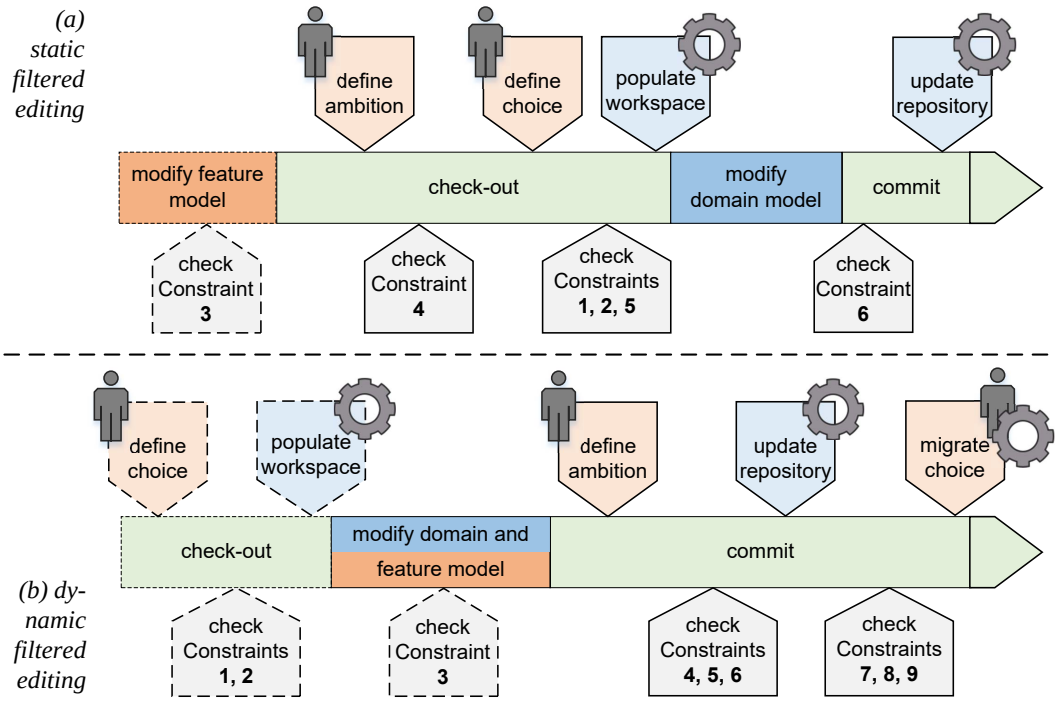


Figure 11.7: Static vs. dynamic filtered editing by phases and constraints.

– As the feature model is not made available for modification, it is not exported into the workspace during CHECKOUT.
– Constraints 1 and 2 are ensured before filtering the domain model as in DFE. Immediately afterwards, Constraint 5 (choice represents ambition) is checked. If this constraint fails, it is not the ambition but the choice that has to be altered.
– During MODIFY, only the domain model can be edited.
– The only constraint that remains to be checked at COMMIT is Constraint 6. If this constraint fails, however, the user cannot be asked for a new ambition. Rather, the *write set* representing the product-level changes must be revised in order to be *sufficiently general*. [11]
– MIGRATE (see Algorithm 11.6) is abandoned entirely.
– Feature model modification is enabled between workspace transactions in an unfiltered editing mode.

11.6.2 Restricted and Alternating Transactions

More restrictive forms of (static or dynamic) filtered editing can be approached by tailoring DFE towards only one product dimension, the feature model or the domain model.

Feature Model Transaction. In this form of restricted transaction, only the feature model is made available in the workspace. Since this is only versioned by the revision graph, whose consistency is managed automatically (see Section 11.4), constraint validation becomes entirely transparent to the user, who is accosted only with selections in the revision graph and, e.g., with commit messages.

Domain Model Transaction. It may also be desirable in many scenarios to remove the feature model from the workspace, or to make it unmodifiable. Then, the co-evolution problems motivating Constraints 3, 7, 8, and 9 become irrelevant.

A concrete workflow similar to the one implied by the SFE model can be realized by applying restricted feature model transactions and domain model transactions *alternatingly*. Furthermore, *nested* transactions are conceivable. A feature model transaction may aggregate several domain model transactions, all committed under the same historical scope.

11.6.3 Earlier Ambition Specification

Even when sticking to the DFE model, the action *specify ambition* can be preponed to any point in time after *specify choice*. This would allow the user to fix the scope of the intended change earlier than during commit; furthermore, the check of Constraints 4 (weak ambition consistency) and 5 (ambition represented by choice) can be applied earlier, which may prevent particular consistency violations. Once specified, the ambition would be memorized in a similar way as the choice is during CHECKOUT. Constraint 6 (sufficiently specific ambition) still needs to be checked during commit.

11 This would require a dedicated user interface for inspecting and altering product-level changes. As this is not available in SuperMod, a negative evaluation of this constraint only issues a warning in that case; the user may decide whether to ignore the warning or to revert the change entirely.

11.6.4 The Operation Amend

In Section 9.6, the *change space* optimization was presented, using which ambitions – otherwise occurring in multiple instances in element visibilities – are mapped to a transparent abstraction, namely *changes*. When using this optimization, changes are the singleton location where ambitions specified by the user are recorded and managed. Besides other advantages discussed previously, this introduces a convenient way to retrospectively correct erroneous ambitions used for recent commits even after having closed the corresponding transactions.

We semi-formally define an additional workspace operation AMEND, which behaves as follows:

- The user is asked for a selection in the revision graph. The chosen revision option is r_i.
- The feature model for r_i is reconstructed. It is internally represented by O_f and $\mathcal{R}_f$.
- The change space is searched for a change Δ_i whose mapped ambition corresponds to $r_i \wedge \hat{a}_f$, where a_f describes the feature ambition specified there.
- The user is asked to define a new feature ambition a'_f in O_f as a substitute for a_f.
- Constraint 4 is re-checked using the new ambition a'_f and $\mathcal{R}_f$. If the constraint is violated, the operation is aborted.
- The ambition associated with Δ_i is altered: $r_i \wedge \hat{a}'_f$.
- In $\mathcal{J}_\Delta$, $(\Delta_i \Leftrightarrow r_i \wedge \hat{a}_f)$ is replaced by $(\Delta_i \Leftrightarrow r_i \wedge \hat{a}'_f)$.
- In $\mathcal{P}_\Delta$, $(\Delta_i, r_i \wedge \hat{a}_f)$ is replaced by $(\Delta_i, r_i \wedge \hat{a}'_f)$.

Notice that this operation potentially behaves less consistently than the conventional way of ambition specification. In particular, Constraints 5 and 6, which ensure that the ambition is represented by the choice on the one hand, and sufficiently general for the product-level change on the other hand, are ignored. This is due to the fact that choices and write sets are temporary concepts being valid within one transaction only. As a consequence, elements connected to an amended ambition might not be fully reproducible in all versions included in it, and unexpected product well-formedness violations may occur.

11.7 Related Work

Before we conclude this section, approaches found in the literature that provide similar editing models or related consistency mechanisms are presented. Furthermore, we compare details of related validation techniques for the specific constraints presented in this chapter.

Fully, Partially, and Temporarily Filtered SPL Editing. Related approaches to filtered editing of software product lines may be categorized under *fully filtered editing*, *partially filtered editing*, and *temporarily filtered editing*; see Section 5.4.1.

Approaches to *fully filtered* multi-variant editing were influenced by early concepts of *multi-version editors* such as the *Multi-Version Personal Editor* (MVPE) [SBK88] or *P-Edit* [Kru84] (see also Section 1.4.3). They assume that a view – similar to the workspace in the

here considered framework – is created from a multi-variant document, which corresponds to the repository here. The version available in the view is defined by a *choice* that uniquely denotes a representative of the *ambition*. The here presented framework realizes fully filtered editing. It does, however, not assume or require a specific multi-version editor, but allows arbitrary tools to be used for editing the workspace. To this end, the operations that define and interpret the choice and ambition are provided by generalized version control metaphors, which additionally rely on SPLE abstractions.

Partially filtered editing [WO14; ZS97] aims at hiding variants to which the current change is immaterial, without requiring the choice to be *unique*. There is only a single filter serving as choice and ambition simultaneously. Variability information referring to non-resolved configuration options is presented in the view, e.g., in the form of annotations. As a consequence, specific tools or preprocessor languages are still required in order to cope with variability in the workspace.

A source code centric approach to *temporarily filtered editing* of software product lines is described in [Käs+09]. A partial feature configuration can be specified as *write filter*. Code fragments immaterial for the intended change are hidden. As approximation of a read filter, a *context* is derived as an extended view on the write filter. Similarly, the *FeatureMapper* [HKW08] approach, which is based on annotative variability, offers a temporary write filter in the multi-variant view. Having selected one or more features and invoked the *record* operation, all changes performed in the MVDM are associated with a feature expression derived from the provided feature selection.

View-Based vs. Transactional Filtered Editing. An orthogonal distinction can be made between the categories *view-based* and *transactional* filtered editing. For starting and closing transactions, different abstractions and metaphors are provided in the literature.

In the first case, the filter – which may or may not be further decomposed into read and write filter – can be dynamically changed by fading in and out specific configuration options. Altering the filter directly influences the visible workspace contents. Such an approach is followed, e.g., in [Käs+09; WO14; BPB17]. View-based filtered editing, however, requires specialized multi-version editors or at least a tight integration of the dynamic filter into existing editors. This makes this approach difficult to implement, particularly in model-driven development environments.

In contrast, the *transactional* approach assumes well-defined iterations during which the read filter remains equal. In the here contributed approach as well as in the precursor UVM [WMC01], transactions are opened and closed by generalized forms of the VCS metaphors *check-out* and *commit*. The approach presented in [Stă+16] defines similar operations, *get* and *put*, inspired by the *view-update problem* known from databases [BS81]. P-EDIT [SBK88] relies on the metaphors of conditional compilation, introducing a *write* operation that closes a transaction by a specific write filter. In FeatureMapper [HKW08], temporary transactions are opened and closed by starting and stopping change recording (see above); the view is inferred from a feature selection similar to feature ambitions.

Static vs. Dynamic Filtered Editing. Representatives of *static* filtered editing, e.g., UVM [WMC01] and EPOS [Mun93], require that the ambition be specified at check-

out time; since the rule base does not evolve, constraints dealing with its evolution are unnecessary. Similarly, in [WO14], having a single filter requires that the scope of a change be known beforehand, inhibiting the concurrent introduction of a feature and its realization.

[Stă+16] moves the specification of the write filter from check-out time to commit time, which slightly deviates from strict SFE as defined above. As the version space is not represented explicitly, no co-evolution problems may occur, and no dynamism is required for the editing model.

elIn the presented *dynamic* filtered editing model, the variability model may evolve during an iteration embraced by CHECKOUT and COMMIT. In particular, new configuration options and new configuration rules may be introduced this way. The flexibility implied by this approach is – to the best of the author's knowledge – unique in the literature. The implied consistency problems are described by Constraints 7, 8, and 9, and solved by the novel operation MIGRATE contributed in Algorithm 11.6.

Generality of the Write Set. In the list of dynamism-aware consistency constraints provided here, Constraint 6 plays a special role, considering not only the soundness of the version space (i.e., options, rule base, choice, and ambition) but also of the connection to the product space (by taking change sets and visibilities into consideration). Phrased in the vocabulary used in this thesis, this constraint ensures that "the ambition is specific enough to reproduce the change in all affected variants", or conversely speaking "the change is general enough to be reproduced in all variants included in the ambition".

The potential inconsistencies that may occur by having the user inadvertently change a larger set of variants than he/she intends to do – namely the variants that contain those elements visible in the view – have been recognized in the literature previously. Different solution strategies have been developed:

In [Kru84; SBK88], there is a distinction between *fixed* and *unfixed* fragments made available in the workspace. Fixed fragments are visible in all variants included in the ambition, whereas unfixed fragments are visible only in a part of the ambition that includes the choice. Unfixed fragments must be managed in a more or less restrictive way. For example, as explained in [SBK88], P-EDIT highlights unfixed fragments, such that the developer becomes aware of a potentially undesired modification of hidden artifacts.

The *edit isolation principle* described in [WO14, p. 31] states that "the only variants that change in the source are those that can be reached from the view", where "source" denotes the multi-version representation. This has been used as the central design constraint for the specification of an *update* (here: commit) operation, which does not only detect but also repair situations in which the principle is violated. When compared to the edit isolation principle, Constraint 6 is more restrictive since it disallows, e.g., modifications that destroy cross-links to invisible elements.

11.8 Summary

This chapter has addressed the combination of the version space (cf. Chapter 9) and the product space (10) within a consistency-preserving dynamic filtered editing model.

The following assumptions underlie the DFE model: First, the user wants to specify all information referring to the version space *as late as possible*; therefore, the definition of an ambition is deferred to the COMMIT phase, and modifications to the feature model need not be performed in advance to an iteration, but may be incorporated during the MODIFY phase. Second, the user wants to be accosted with version specification tasks *as seldom as possible*, which vindicates the decision to make the CHECKOUT operation optional and to introduce MIGRATE in favor of a reduced obtrusiveness. Third and last, the editing model should be *no more restrictive than necessary* in order to prevent the user from performing changes that cause product inconsistencies or that potentially cannot be reproduced.

Consistency is effectively checked by explicit constraints assigned to the different phases of an editing cycle. During CHECKOUT, it is ensured that the specified choice – a feature configuration – is *unique* and *consistent* with respect to the feature model. During MODIFY, changes that would make the feature model unsatisfiable are disallowed. The feature ambition defined by the user during COMMIT is checked for *weak consistency*, as well as for being *represented* by the choice. Furthermore, it must be *sufficiently specific* for the performed product-level change. The newly introduced operation MIGRATE automatically produces a choice for the next iteration based on the previous choice and the ambition, in order to obviate repeated check-outs that reproduce the current workspace view. By corresponding constraints, the migrated choice is checked for *uniqueness* and *strong consistency* with the evolved version of the feature model. Furthermore, it must *include* the ambition.

The constraints are formally defined based on the concepts introduced in Chapter 9, which have been in turn inherited from UVM [WMC01]. The correctness of the consistency-preserving operations is proved; furthermore, we provide evidence that the revision graph is managed not only automatically but also consistently.

The decision whether to apply *static* or *dynamic filtered editing* is related to the amount of *flexibility* (i.e., late ambition specification, co-evolution of feature model and domain model) and of *consistency guarantees* (i.e., by preventing certain co-evolution problems) required by a specific project. In the presented conceptual framework, it is assumed that DFE is the preferred style, but SFE can be adopted gradually in case a more conservative workflow is demanded.

Several ways of generalizing the presented dynamic editing model have been sketched. On the one hand, purely static filtered editing model may be realized by moving particular version selection interactions and constraint checks within the algorithms. Restricted transactions may ensure that only the feature model or only the domain model are edited, e.g., in an alternating fashion. Furthermore, the ambition may also be specified at an earlier point in time, which slightly increases the consistency at the expense of a more restricted editing model. Last, the AMEND operation even allows to retrospectively alter the ambition used for a previous commit, such that erroneous version specifications can be revised.

Taken together, this chapter contributes to the satisfaction of multiple requirements arrayed in Section 2.3. Filtered editing enables (automated) *management of variability annotations* (**R7**), *views on product variants* (**R8**), and *automated product derivation* (**R9**). Through the automated and consistent management of the revision dimension, **R2** (extensional revision selection), **R3** (immutability of revisions), and **R4** (transparent multi-revision storage) are provably fulfilled. Furthermore, the evolution of the feature model (**R13**) is now controlled in a consistent way, and the underlying uniform version mechanism (**R15**)

is specified in greater detail. The AMEND operation potentially reduces the gap between historical and logical versioning (**R14**) by allowing to retrospectively map an evolutionary increment to an optional feature.

Implementation details with respect to both the DFE model and its generalizations are provided in Section 14.2. Furthermore, in Sections 15.4.3 and 15.4.4, we evaluate the benefits of the DFE model over unfiltered SPL editing as well as over the SFE model.

And there's no such thing
as too much back-up.

STEPHEN BAXTER AND TERRY PRATCHETT: THE LONG WAR (2013 NOVEL)

Chapter 12

Collaborative and Distributed Versioning

Abstract

Collaborative software development represents one of the key functionalities to be supported by version control, and it is also relevant to SPLE. Hitherto, the conceptual framework is restricted to single-user operation. The basic design principle to overcome this restriction is the controlled orchestration of the evolution and synchronization of physically independent copies of a repository. Local transactions realized by check-out and commit are complemented by remote transactions, which are provided through the operations pull and push. The first extension concerns the revision graph metamodel: A distinction between public revisions, embraced by pull and push, and private revisions, representing local check-out/commit transactions, is made, introducing a two-level hierarchy to the revision graph. Secondly, the dedicated master copy of the repository is extended by transaction support. Third, as the optimistic synchronization strategy may involve conflicting modifications, three-way merging comes into play. To this end, a context-free variability-aware three-way merge strategy is contributed; merge conflict resolution is addressed by the successor chapter. At the end of this chapter, the operations pull and push are semi-formally specified.

Contents

12.1 Overview

In this chapter, we explain extensions to the conceptual framework providing the basis for *collaborative* filtered MDSPLE. The remainder of this section sketches the extended architecture as well as the most relevant design decisions. Furthermore, it gives a brief introduction to the added functionality. In Section 12.2, the revision graph metamodel and the corresponding mapping to the version space base layer, introduced in Section 9.3, is refined in order to meet the extended architecture and functionality. Section 12.3 is dedicated to the centralized management of public transactions. Specialized three-way merging strategies for different copies of the repository are introduced in Section 12.4. The building blocks presented in the preceding three sections are combined in a collaborative editing model in Section 12.5, where the synchronizing operations PULL and PUSH are semi-formally defined. Before the chapter is concluded, related work on distributed MDSPLE is presented in Section 12.6. [1]

12.1.1 Architectural Sketch

An architectural sketch illustrating the extended conceptual framework is depicted in Figure 12.1. The illustration complements Figure 9.4, whose contents are represented in a condensed form on the right hand side. As before, the user performs modifications in a local workspace and communicates with a – likewise *local* – repository using generalized forms of the VCS commands CHECKOUT and COMMIT.

Several local repositories communicate with the central remote repository using the synchronization operations PULL (for accepting changes incoming from remote copies) and PUSH (for delivering local changes to other remote copies), inspired by the *distributed version control system* (DVCS) *Git* [Cha09].

1 This chapter shares a significant amount of material with [SW16a].

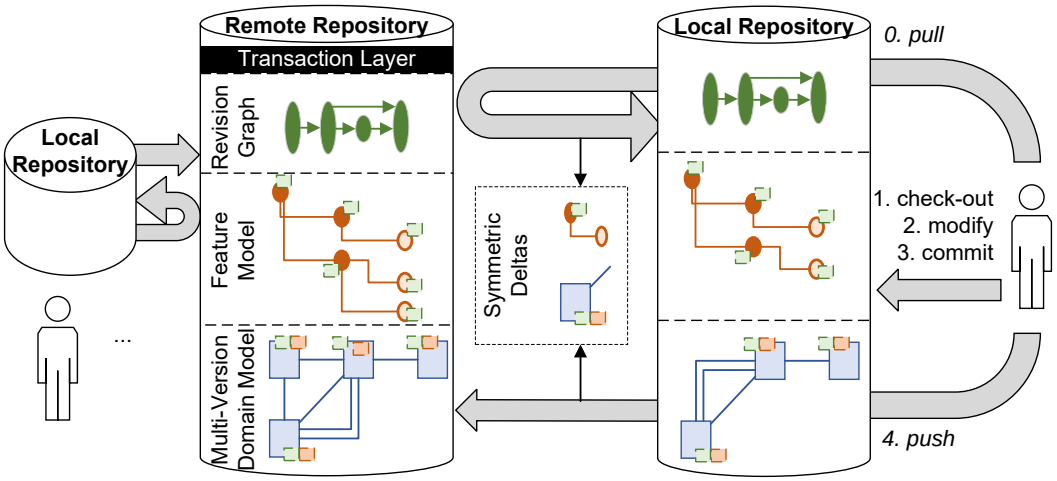


Figure 12.1: Architectural sketch: Collaborative extension of the conceptual framework. Based on [SW16a, Figure 4].

The extrinsic representation of the repository artifacts, see Chapters 9 and 10, is used within the remote repository in the same way. As shown in Figure 12.1, the remote repository contains in addition a *transaction layer* on top of the revision graph. This organizes synchronization operations invoked by different local repositories and detects, e.g., concurrent modifications, which must be reported to the connected users.

Last, the information interchanged along with PULL and PUSH needs to be defined. As the connection between the repositories is network-based, traffic should be reduced to a minimum. Therefore, *projections* of *symmetric deltas*, which only contain those elements that are required to describe the transferred repository updates, are employed.

12.1.2 Internal Design Decisions

The detailed explanations provided subsequently, which refine the described architecture, are justified by the following design decisions.

Distributed Replication by Singleton Master. According to Section 4.5, there are two general approaches based upon which *collaborative version control*, issued by design decision **D9** on page 137, may be realized. The *centralized* paradigm implies that every check-out and commit read and write the state of a singleton master repository. In contrast, the *distributed* approach allows for several copies of the repository, which are synchronized not upon each and every check-out and commit but at dedicated user-defined synchronization events. Different synchronization strategies include *peer-to-peer* and *singleton master*; see related work in Section 12.6.

The conceptual extension contributed here employs the *singleton master* strategy for synchronizing different copies of a repository. Accordingly, there is a *central remote repository*, which communicates with multiple *client repositories*.

Remote and Local Transactions. The extended framework distinguishes between remote transactions, which are started with a PULL and finished with a PUSH operation,

from local transactions, which realize an CHECKOUT/MODIFY/COMMIT iteration. This reduces synchronization overhead and enables off-line product line development. Moreover, it is not necessary to start a remote transaction explicitly; rather, all necessary bookkeeping steps are enforced after each PULL transparently.

Enforcement of a Linear Version History. Each revision committed to the repository is a successor of the latest revision available, the *head*. More precisely, *public* revisions, which organize remote transactions, form a *totally ordered directed acyclic graph* (TODAG), which contains transitive edges (created by *merging*) in addition to a sequential chain. Each public node is refined into a *sequence* of *private* revisions, which organize local transactions. By intention, the conceptual framework does not support permanent *branches*; as a replacement, configuration decisions that justify co-existing versions of the product should be modeled as features.

Projected Symmetric Deltas for Information Interchange. The data transferred between different copies of the repository during PULL and PUSH is represented as regular model instances complying to the extrinsic metamodels introduced in Chapter 10. To ensure that only the information necessary to reproduce the change is transferred, unmodified parts of the product space are projected away from the symmetric deltas.

Referential Integrity. The conceptual framework abstracts from specific product space types; however, it is assumed that *cross-links* between elements of the product space need to be managed. Besides, references between higher-level version models and low-level rule base elements (e.g., from features to their options) exist. Last, through *visibilities*, product space elements refer to options defined in the version space. It is therefore important to correctly handle links from elements part of a symmetric delta to elements not included there.

Context-Free and Non-Interactive Three-Way Merging. According to our chosen optimistic versioning strategy, concurrent modifications can occur. The here applied non-interactive three-way merging procedure is context-free and deterministic. The surrounding collaboration mechanism involves the user as late as possible, i.e., when he/she attempts to check-out a product that contains conflicts. To reduce the cognitive overhead for the user, only the contents available in the workspace are analyzed for well-formedness. This is a subject of the next chapter; see Section 13.4.

12.1.3 Added Functionality

Besides the aforementioned PULL and PUSH, operations for initializing, connecting to, and removing the connection to a central remote repository are needed. The overall added functionality is listed in terms of user-visible operations below; Figure 12.2 illustrates the collaborative editing model as seen from a single user's perspective. The state chart extends Figure 11.4 by new states and transitions, the latter of which correspond to synchronization operations.

Create. The user should be enabled to create a new central remote repository with empty contents, and empty version history. The workspace from which this operation was called is considered to be the only client registered in the beginning.

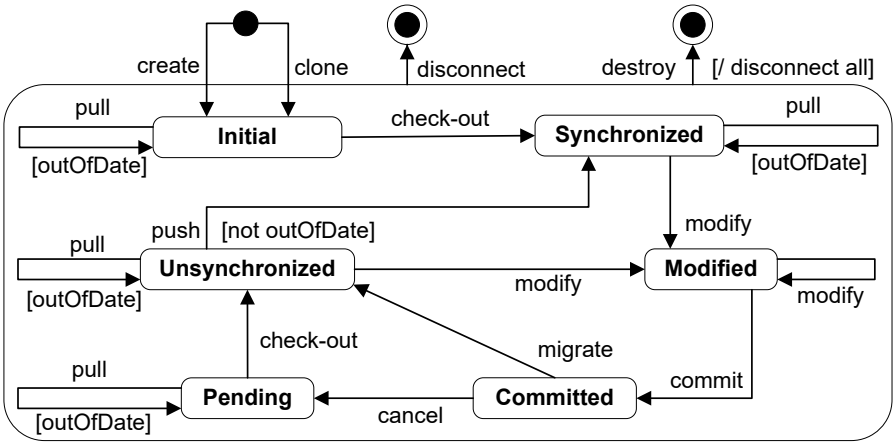


Figure 12.2: States and transitions of a client in the collaborative editing model.

Clone. The functionality provided by this operation comprises the initialization of a local client repository whose contents are copied from a given master repository. The client is registered with the master.

Pull. The contents of the client repository are updated in order to incorporate all changes to all versioned elements pushed to the master repository by other clients since the last synchronization. The operation is applicable only in case remote changes are pending (*out of date* situation) and in case no private transaction is active. Local changes committed in the meantime shall not get lost.

Push. Provides the inverse functionality of PULL. All changes locally committed since the last synchronization are to be transferred to the master repository. Applicable only when no remote changes can be pulled and when no private transaction is active.

Disconnect. The inverse of CLONE. Removes a client from the list of repositories connected to the master. This operation also removes the client copy of the repository, but never the workspace contents.

Destroy. Permanently deletes a master repository, forcing remaining clients to disconnect.

CREATE, CLONE, DISCONNECT, and DESTROY are mainly of technical interest and are therefore not detailed any further in the remainder of this chapter. Their implementation is briefly discussed in Section 14.5.4. In contrast, PULL and PUSH impose considerable conceptual challenges, which are examined in the following.

12.2 Collaborative Revision Graphs and their Mapping

Figure 12.3 and Table 12.1 illustrate an Ecore-compliant metamodel for collaborative revision graphs as well as a mapping to the low-level rule base for instances thereof. Both the figure and the table redefine the mapping for single-user revision graphs from Section 9.3.

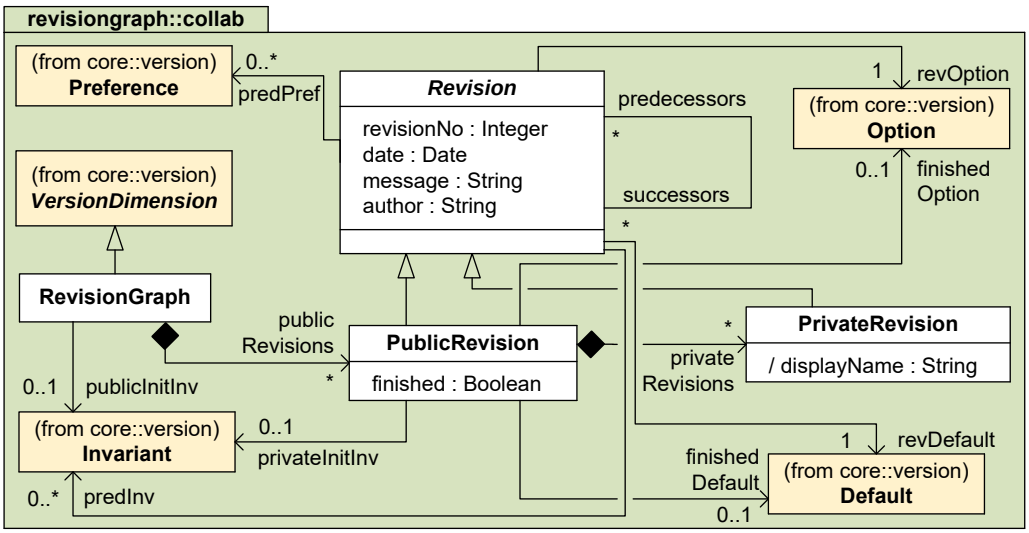


Figure 12.3: Metamodel for collaborative revision graphs. Based on [SW16a, Figure 2].

12.2.1 Structural Design

A *revision graph* is a container for *public revisions*, which represent remote transactions and in turn contain *private revisions* expressing local transactions. Both inherit from an abstract base class Revision that defines an attribute for the revision number – private revisions are externally displayed using the superordinate public revision as qualifier – and additional commit details (date, message, author). Secondly, a generic predecessors/successors relationship[2] is defined, which is instantiated in order to connect both public and private revisions according to the formal mapping defined below.

12.2.2 Formal Mapping

In addition to the structure of the revision graph, the metamodel shown in Figure 12.3 also defines references to low-level rule base elements (see Section 9.2.2), which are transparently derived using the transformation patterns defined in Table 12.1.

Private revisions are mapped to a revision option straightforwardly (cf. pattern 2), whereas public revisions are mapped to two options, starting (1) and finishing (3) the transaction, respectively. As in the single-user mapping (Table 9.1), either option is supplemented with a default that defines a negative selection state in case a revision is (and all of its successors are) deselected. Pattern (4) ensures that the initial public revision is always selected.

As in the single-user mapping described in Section 9.3, predecessor/successor relationships between two revisions x and y are mapped to invariants of the form $r_y \Rightarrow r_x$ and

2 There are two reasons why both references have been defined as multi-valued (unlike in single-user revision graphs; see Figure 9.8): First, for public revisions, the TODAG structure may contain transitive edges in addition to a chain that expresses a linear order. Second, some relationships between public and private revisions are defined redundantly; e.g., a public revision has a public in addition to a private predecessor. See example.

Table 12.1: Detailed mapping between collaborative revision graphs and low-level rule base concepts. Based on [SW16a, Table 1].

	Pattern	Transformation		Metamodel
1	public revision i	option	r_i	revOption
		default	$(r_i, false)$	revDefault
2	private revision $i.j$	option	$r_{i.j}$	revOption
	nested in public revision i	default	$(r_{i.j}, false)$	revDefault
3	finished public revision i	option	$r_{i.\infty}$	finishedOption
		default	$(r_{i.\infty}, false)$	finishedDefault
4	initial public revision 0	invariant	r_0	publicInitInv
5	public revision i as successor	invariant	$r_i \Rightarrow r_{f.\infty}$	predInv
	of finished public revision f	preference	$(r_{f.\infty}, r_i)$	predPref
6	initial private revision $i.0$	invariant	$r_{i.0} \Rightarrow r_i$	privateInitInv
	nested in public revision i			
7	private revision $i.[j+1]$ as	invariant	$r_{i.[j+1]} \Rightarrow r_{i.j}$	predInv
	successor of private revision $i.j$	preference	$(r_{i.j}, r_{i.[j+1]})$	predPref
8	finished public revision f	invariant	$r_{f.\infty} \Rightarrow r_{f.h}$	predInv
	with private head $f.h$	preference	$(r_{f.h}, r_{f.\infty})$	predPref
9	finished public f with remotely	invariant	$r_{f.\infty} \Rightarrow r_{c.\infty}$	predInv
	finished c causing *out of date*	preference	$(r_{c.\infty}, r_{f.\infty})$	predPref

defaults (r_x, r_y). There are several cases in which such relationships[3] are created: (5) A finished remote transaction is superseded by a succeeding remote transaction, which is started immediately after PUSH. (6) All private revisions follow the start revision of the parent public revision. Pattern (7) ensures successorship of private revisions, which are organized as a sequence. Pattern (8) is instantiated before finishing a remote transaction and connects the *private head* (the most recently committed private revision) to the *finished option* of the organizing public revision.

Last, pattern (9) comes into play as soon as concurrent modifications happen—it enforces a totally ordered version history by automatically superseding remotely finished transactions. To the user, such a situation is signaled as *out of date*: A change represented by a remotely finished transaction c must be pulled first, merging the changes locally before pushing f.

12.2.3 Example

To illustrate the dynamic behavior of a collaborative revision graph, represented as instance of the metamodel shown in Figure 12.3, the example in Figure 12.4 depicts a version history involving two fictional developers, Alice and Bob.

Alice creates the repository, which transparently introduces a revision 0 and a corresponding public revision option r_0, a default $(r_0, false)$, and an initial invariant r_0 (cf. patterns

3 This is the only occurrence of ambiguous preferences for options in the whole framework. Conflicting preferences, thus non-determinism, are avoided, one the one hand by the TODAG structure, and on the other hand by the fact that preferences cannot evaluate to $false$ (in contrast to defaults, which are applied secondly).

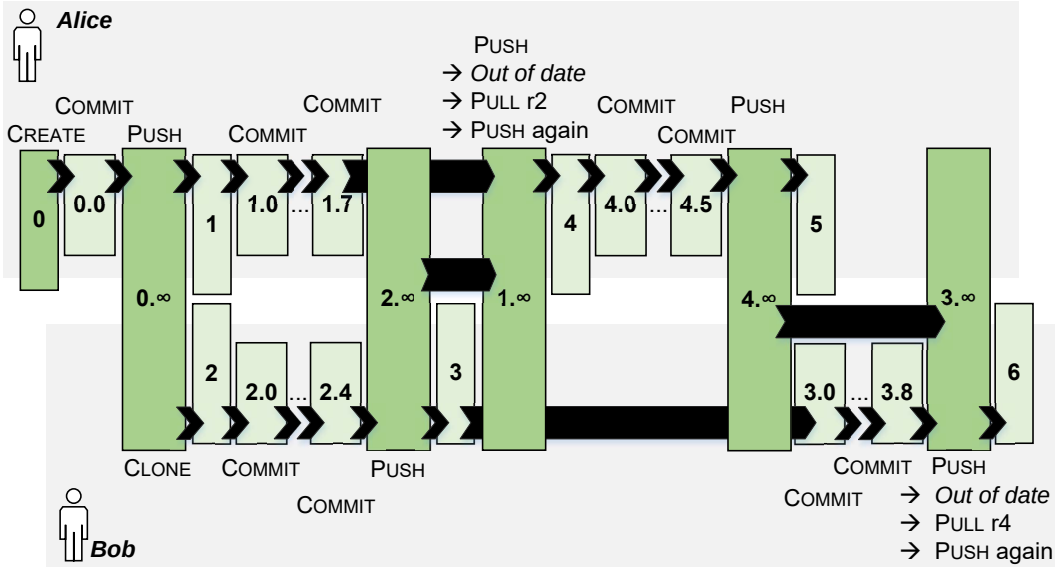


Figure 12.4: Example revision graph illustrating the interplay between public and private revisions. Boxes denote revisions with corresponding options. Black arrows starting at r_x and ending at r_y denote an instantiation of a predecessor invariant $r_y \Rightarrow r_x$ and of a predecessor preference (r_x, r_y). Based on [SW16a, Figure 3].

1 and 4 in Table 12.1). Next, she performs an initial commit, which introduces a nested private revision $r_{0.0}$ as successor of r_0 (pattern 6). By finishing the transaction through the PUSH operation, option $r_{0.\infty}$, default $(r_{0.\infty}, false)$ (3), and invariant $r_{0.\infty} \Rightarrow r_{0.0}$ (8) are introduced transparently. The next transaction is started instantaneously, introducing r_1, $(r_1, false)$ (1), as well as invariant $r_1 \Rightarrow r_{0.\infty}$ and preference $(r_{0.\infty}, r_1)$ (5).

In the meantime, Bob clones the repository, which also starts a write transaction (option r_2, preference $(r_2, false)$, invariant $r_2 \Rightarrow r_{0.\infty}$, and preference $(r_{0.\infty}, r_2)$). Concurrently, both developers commit private revisions to their local repositories – such that pattern (7) is applied repeatedly – and then finish their current remote transaction. Bob is the first to push, r_2 is closed, and r_3 is started immediately. Thereafter, Alice attempts to push, but receives an *out of date* error, enforcing a PULL based on the current head, such that the incoming $r_{2.\infty}$ is locally merged with $r_{1.\infty}$. When pushing again, pattern (9) is instantiated, adding the invariant $r_{1.\infty} \Rightarrow r_{2.\infty}$ and the preference $(r_{2.\infty}, r_{1.\infty})$ to the rule base transparently. Moreover, a new remote transaction is begun by introducing revision 4 and applying patterns (1) and (5) in a suitable way.

Alice finishes her pending remote transaction straightforwardly. After that, Bob starts his work forgetting to pull r_4. Thus, he receives an *out of date* error when attempting to push $r_{3.\infty}$. The incoming $r_{4.\infty}$ is automatically merged before finishing r_3.

Lessons Learned. When compared to single-user revision graph management, the underlying mechanisms defined above in this section and illustrated in this example are intrinsically complex. Nevertheless, the fictional users are only exposed to few additional revision control metaphors (*push*, PULL, *out of date*), such that the added functionality –

collaborative SPL versioning – is paid with a tolerable increase in cognitive complexity.

12.3 Centralized Management of Remote Transactions

After having described the extensions to the version space, let us move on to the product space related topics. As explained in the introductory section of this chapter, both the master and local repositories each comprise a full copy of the product space. Nevertheless, symmetric deltas transferred with PULL and PUSH should be kept small. To this end, it is necessary to keep track of the elements modified by specific remote transactions. This is provided by a *low-level transaction layer*; see Section 12.3.1. Proximately, Section 12.3.2 explains how symmetric deltas are eventually calculated.

12.3.1 Low-Level Transaction Layer

In order to keep track of the elements modified by the current write transaction, a primitive *transaction layer* has been added to the conceptual framework. It is realized by the following extensions to the server-side master and/or the client-side local repositories:

- Each local repository carries a *read transaction number*, which indicates the revision number of the most recently pulled public revision; moreover, a *write transaction number*, which equals the revision number of the public revision organizing the current remote transaction. Both are part of the *metadata* section of local repositories; see Section 13.2.1, p. 271.
- Each product space element carries a numerical *transaction identifier* that indicates the transaction associated with its most recent modification (i.e., insertion, deletion, or modification of a child element). In the product space base metamodel shown in Figure 10.2 on page 194, this has been conceptually prepared by the attribute transactionId.
- The *visibility update* operation (cf. Algorithm 11.5 in Section 11.3.3 on page 232) is modified such that it assigns the current write transaction number to inserted and deleted elements.
- The master repository manages a global *transaction log* to which transaction starting and finishing events are appended in historical order. Also from this log, new public revision numbers are generated by incrementing the number of the most recently opened transaction by one. Transactions are opened but not necessarily closed, in numerical order.

Example. The transaction log for the example from Figure 12.4 is

```
o0 c0 o1 o2 c2 o3 c1 o4 c4 o5 c3 o6
```

where `oX` denotes that a transaction X has been opened, and `cX` represents the corresponding closing event. From this log, we may infer that transactions 5 and 6 are still running, and that the subsequently started remote transaction will carry the number 7. Furthermore, *out of date* situations can be detected; for instance, if Alice pushed now, she would be forced to pull revision 3, pushed in the meantime, first. In contrast, Bob may push revision 6 without being interrupted, since no new transaction was closed after the event `o6`.

```
procedure ProjectDelta(P, tNo)
    Π0 ← {e0 ∈ P| the transaction number of e0 equals tNo}
    Π1 ← ∅
    for e0 ∈ Π0 do
        for {e1 ∈ P|e1 ∈ crossLinks(e0)} do
            if e1 ∉ Π0 then
                Π1 ← Π1 ∪ {e1}
        for o1 ∈ options referred to in v0 (visibility of e0) do
            ve1 ← high-level product space element (feature or revision) belonging to o1
            if ve1 ∉ (Π0 ∪ Π1) then
                Π1 ← Π1 ∪ {ve1}
    Πpar ← ∅
    for e01 ∈ (Π0 ∪ Π1) do
        for epar ∈ parent+(e01) do
            if epar ∉ (Π0 ∪ Π1 ∪ Πpar) then
                Πpar ← Πpar ∪ {epar}
    return Π0 ∪ Π1 ∪ Πpar
```

Algorithm 12.1: Symmetric delta projection. Based on [SW16a, Algorithm 1].

12.3.2 Symmetric Delta Projection

The increments transferred along with push and pull are *symmetric deltas*; they are internally represented as subsets of the product space, which in turn consists of the feature model and the domain model. As pointed out in Section 12.1.2, special emphasis is put on the integrity of *cross-links* between different types of elements.

Algorithm 12.1 describes a generic procedure for computing symmetric deltas as a projection of the product space based on a given transaction number. The delta is returned as the subset of the product space that is unionized by the following elements:

0-Context. All elements that carry the specified transaction number.

1-Context. Elements cross-referenced by elements in 0-context or by the visibilities of elements in 0-context.

Parent Context. The transitive closure over the containers (denoted as $parent^+$) of all elements in the union of 0-context and 1-context.

Symmetric delta projection is used in two instances in the extended editing model (see Section 12.5): on the client side for identifying the elements to PUSH based on the write transaction number, and on the remote side when calculating the element set to be transferred to the client who requested a PULL for a given read transaction number.

12.4 Context-Free Three-Way Merging

The counterpart to delta projection is *merging*, the key functionality to be offered by an optimistic synchronization strategy (see design decision **D9**). This operation applies an incoming delta to a product space P, which is an "append-only" structure; no element is ever permanently removed in order to enable the reconstruction of previous historical versions.

Three-way merging as explained here is context-free and non-interactive; it assumes a purely set-theoretic definition of the product space and does not take into consideration the well-formedness rules defined by the underlying metamodels. Such inconsistencies are detected in local workspaces in a product-based way; details are explained in Chapter 13.

12.4.1 Element Merging

Relying on the asymmetric two-way raw MERGE operation introduced in Algorithm 10.4 on page 213, a merged product space P' can be easily computed from a master version P and an incoming delta Π as follows:

$$P' = \text{MERGE}(\Pi, P, \mathit{diff}, match) \tag{12.1}$$

In the above equation, $match = \text{MATCH}(\Pi, P)$, and $\mathit{diff} = \text{DIFF}(match)$.

The merge is performed based on the criterion $same$ introduced in Section 10.2.2, thus:

$$\forall e_1, e_2 \in P' : same(e_1, e_2) \Rightarrow e_1 = e_2 \tag{12.2}$$

12.4.2 Raw Visibility Merging

Though, *visibility conflicts*[4] arise whenever two *same* elements $e_i^P \in P$ and $e_i^\Pi \in \Pi$, which are merged to one element in $e_i' \in P'$, carry different visibilities in the mappings provided by P and Π. Special attention must be paid since visibilities encode insertions and deletions of elements. We have to distinguish between *raw merging* (an incoming delta is supposed to be integrated into the repository that has so far been *up to date*) and *three-way merging* (a local repository contains outgoing changes which conflict with an incoming delta, i.e., the local repository is *out of date*).

In case no outgoing changes exist to redefine the visibility of an element $e_i^P \in P$, the (more recent) visibility v_i^Π is transferred from the delta version Π to the master version P.

Thus, visibilities are raw merged as follows:

$$v_i' = \begin{cases} v_i^\Pi & \text{if } e_i \in \Pi \text{ and } v_i^\Pi \text{is defined} \\ v_i^P & \text{otherwise} \end{cases} \tag{12.3}$$

As before, in case an element e_i does carry a visibility, $v_i = true$ is assumed implicitly.

12.4.3 Three-Way Visibility Merging

If the local repository is *out of date*, however, the visibilities of its elements in P may conflict with the visibilities defined in Π. In this case, the common base version B must be considered:

$$v_i' = \begin{cases} v_i^P & \text{if } e_i \notin \Pi \text{ or if } v_i^\Pi \text{ is undefined} \\ v_i^\Pi & \text{if } e_i \notin P \text{ or if } v_i^P \text{ is undefined} \\ \mu(v_i^B, v_i^P, v_i^\Pi) & \text{otherwise} \end{cases} \tag{12.4}$$

4 To avoid confusion between element identifiers and versions of a product space, the latter are here denoted using superscripts rather than subscripts, i.e., v_i^P denotes the visibility of element e_i in product space P.

Here, v_i^B is the visibility of e_i in the common *base* version B of the considered revisions in the revision graph. Product space version B can be identified by searching the revision graph for the latest common predecessor of the revisions underlying P and Π. [5]

Along with this, μ denotes the *three-way visibility merge function*[6]:

$$\mu(v_i^B, v_i^1, v_i^2) = (v_i^1 \wedge v_i^2) \vee (v_i^1 \wedge \neg v_i^B) \vee (v_i^2 \wedge \neg v_i^B) \tag{12.5}$$

In case there is no difference between v_i^B and v_i^1, this function degenerates to v_i^2 (and conversely for swapping v_i^1 and v_i^2). Otherwise, insertions and deletions made effective by the visibility update operation (Algorithm 11.5) in v_i^1 and v_i^2 in isolation are combined.

12.4.4 Visibility Forest Merging

Provided the *visibility forest* optimization presented in Section 9.7, where propositional logical operations are represented as nodes in a directed graph structure, the evaluation effort as well as the traceability of visibility expressions created by the three-way merging operator μ may be further improved: We have introduced a ternary node type Merge – technically, a subtype of BinaryExpr with an additional operand base – in the option expression metamodel in Figure 9.7. Altogether, the base and the other two operands refer to v_i^B, v_i^1, and v_i^2, which are internally represented by option expression references to existing nodes. The evaluation of a merge node, externally represented by the operator μ, is then realized as defined in (**12.5**).

12.4.5 Example

Let us illustrate the visibility forest merging strategy by the following example. An element e_i is assumed to have been introduced in revision 0 by user Bob with a global scope, such that $v_i^B = r_{0.0}$. Then, Bob commits and pushes. Next, Alice pulls revision 0 and deletes e_i under a logical ambition where feature A is selected. Therefore, $v_i^1 = r_{0.0} \wedge \neg(r_{1.0} \wedge f_A)$. She commits and pushes. In the meantime, Bob pulls and locally deletes the same element e_i,

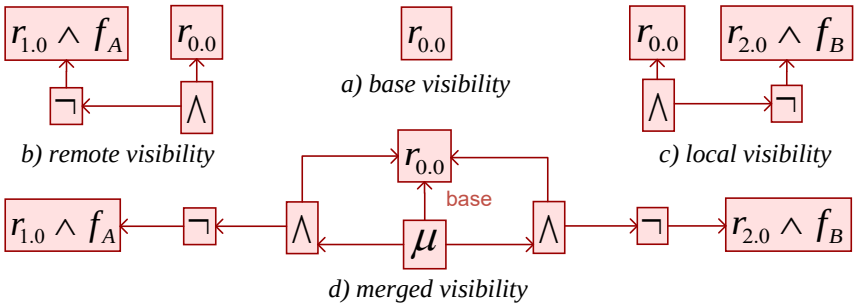


Figure 12.5: An instance of a visibility forest with merge node.

5 Technically, v_i^B is obtained by taking as a basis v_i^P and removing all terms from the conjunctively represented visibility which refer to a revision option that supersedes B.

6 This function is a propositional logical conversion of the three-way merge formula for sets; see **6.1** on page 109, which originates from [Wes14].

but under ambition B. Before pushing, the local visibility of e_i is $v_i^2 = r_{0.0} \wedge \neg(r_{2.0} \wedge f_B)$. When attempting to push, Bob gets an *out of date* warning and is forced to pull Alice's change first. The affected repository contents are merged in his local workspace. Since the visibility of e_i has been modified conflictingly, three-way visibility merging is applied as follows in order to construct the merged revision $r_{3.0}$:

$$\begin{aligned}\mu(v_i^B, v_i^1, v_i^2) = \big((r_{0.0} \wedge \neg(r_{1.0} \wedge f_A)) \wedge (r_{0.0} \wedge \neg(r_{2.0} \wedge f_B)) \vee \\ \big(r_{0.0} \wedge \neg(r_{1.0} \wedge f_A) \wedge \neg r_{0.0}\big) \vee \big(r_{0.0} \wedge \neg(r_{2.0} \wedge f_B) \wedge \neg r_{0.0}\big) \\ = r_{0.0} \wedge \neg(r_{1.0} \wedge f_A) \wedge \neg(r_{2.0} \wedge f_B)\end{aligned}$$

Figure 12.5 depicts the internal representation of the relevant cut-outs of the visibility forest in its state at v_i^B, v_i^1, v_i^2, and the merged visibility v_i'. A merge node, represented by μ, was introduced in the course of three-way visibility merging. The node labeled with $r_{0.0}$ was identified as base node, whereas the alternative visibility values, after merging the forest, are referenced as operands.

12.5 Semi-Formal Definition of a Collaborative Editing Model

Based upon the definitions from the preceding subsections, we semi-formally define the operations PULL and PUSH referred to in Figure 12.1. By intention, we refrain from specifying the operations at the level of detail of algorithms as in Section 11.3. After all, too many details depend on implementation decisions; see Sections 14.4.5 and 14.5.4. Furthermore, the existing operation COMMIT must be extended in order to comply with the rule base mapping redefined in Table 12.1, particularly with respect to the management of private revisions.

The descriptions given for both operations assume that the master repository has already been initialized using the CREATE operation introduced in Section 12.1.3. The collaborative revision graph is equipped with an empty public revision 0.0, and accordingly, patterns 1 and 4 defined in Table 12.1 have been instantiated. Furthermore, both operations are only allowed if there are not any pending local changes (cf. Figure 12.2); for preventing them from getting lost, these need to be committed in advance.

12.5.1 Pull

Operation PULL fetches incoming changes from the master and appends them to a dedicated client repository, from which the operation is called. It proceeds in the following way:

1. The current *read transaction number* r of the client repository is sent to the master as part of a pull request.
2. The *transaction log* of the master is analyzed in order to find all write transactions $w_1, \ldots, w_m$ closed after the closing event corresponding to the transmitted read transaction number r.
3. If there are not any closed write transactions, the client repository is *up to date*. Then, cancel the operation.

4. For each write transaction w_i, a *symmetric delta* Π_{w_i} is computed using Algorithm 12.1. As the base element set P, the master repository's product space is taken.
5. The deltas $\Pi_{w_1}, \ldots, \Pi_{w_m}$ are serialized and sent to the client.
6. On the client, the incoming deltas are successively raw-merged with the product space as shown in Section 12.4.2. Visibilities are processed as follows:
   a) In case there are no outgoing changes, *raw merging* is applied; cf. (**12.3**).
   b) In case there are outgoing changes (i.e., pending commits in an unfinished public revision), *three-way merging* (**12.4**) is applied.
7. The client's *read transaction number* is updated to the write transaction number w_m that belongs to the most recent symmetric delta pulled, Π_{w_m}.
8. A CHECKOUT is recommended to the client user in order to transfer the pulled changes from the local repository to the workspace. Here, product well-formedness violations caused by three-way merging are addressed (see Chapter 13).

12.5.2 Push

The inverse operation, PUSH, finishes a remote transaction – representing a sequence of local transactions – and transfers all therein committed changes to the server.

1. The client's read transaction number r is matched with the most recently closed transaction number available in the sever-side transaction log, w_m. If the numbers differ ($r \neq w_m$) the client is *out of date*. This is signaled to the user, who may decide either to PULL the incoming changes or to abort the operation.
2. If necessary and if the user agrees, a PULL is enforced. For incoming changes, corresponding public revisions $w_1, \ldots, w_m$ are memorized.
3. The current remote transaction w is finished, instantiating patterns 3 and 8 from Table 12.1. Revision details (author, date, commit message) are applied.
4. A *symmetric delta* Π_w is computed using Algorithm 12.1 and the current write transaction number w. The local repository's product space is taken as base set P.
5. The delta Π_w is sent to the master as part of a push request.
6. On the master repository, the incoming delta Π_w is raw-merged with the remote repository's product space P as shown in Section 12.4.1. For visibility merging, the *raw* strategy (cf. Section 12.4.2) is applied.
7. The *transaction log* is updated by closing the client's write transaction and by opening a new write transaction immediately.
8. The new *write transaction number* is transferred to the client.
9. In the client revision graph, a new public revision is introduced, instantiating patterns 1 and 5 from Table 12.1. In addition, for each of the incoming revisions $w_1, \ldots, w_m$ memorized in step 2, pattern 9 is enforced.

12.5.3 Collaboration-Aware Commit

The mapping of several patterns in the collaborative revision graph to low-level rule base elements also refers to private revisions. These are organized not by the operation PUSH, but by COMMIT, which finalizes a local transaction. Rather than formally redefining the operation here, we describe situations in which corresponding patterns are instantiated:

- If the current public revision i does not contain any private revision yet, create an initial private revision $i.0$ and apply patterns 2 and 6 listed in Table 12.1.
- Otherwise, take the current private head revision, $i.j$ as a basis. Create a new private revision $i.[j+1]$ as successor of $i.j$. Instantiate patterns 2 and 7 from Table 12.1.
- In both cases, execute the standard COMMIT operation defined in Section 11.3.3, omitting all revision management steps, which are replaced by the collaborative patterns redefined here. Use the new revision option ($r_{i.0}$ or $r_{i.[j+1]}$) for the historical part of the ambition (a_r^{cm}). In case the operation fails, undo the pattern applications described above.
- Extend the utility operation UPDATEVISIBILITIES (cf. Algorithm 11.5) in such a way that the current write transaction number of the enclosing remote transaction is assigned to all inserted or deleted elements to which visibility updates are applied.

12.5.4 Example

We refer back to the example introduced in Section 12.2.3 and move the focus from the instantiation of rule base mapping patterns to delta calculation, three-way merging, and the peculiarities of the collaborative editing model.

To facilitate understanding the example, we make some simplifications: (1) One change is performed per commit. (2) For each remote transaction, only the first commit is shown. (3) The feature model consists of a root feature with option f_R and of two optional features having options f_A and f_B and not imposing any further constraints. (4) The product space is represented as a simple *bubbles and arcs* model. Both bubbles and arcs carry a *label*, which also serves as equality criterion. [7]

The final state of the remote repository's multi-version domain model, including visibilities, is depicted in Figure 12.6. Below, the performed changes (each followed by PUSH and COMMIT) are ordered by push date (but not necessarily by public revision number).

Revision 0. Alice initializes the repository, creates the entire feature model, and an initial product consisting of bubbles v, w, x and arc q, which are all contained by the diagram canvas k. She does not associate her change with a specific feature, resulting in the logical ambition $true$ for the lone private revision 0.0. The symmetric delta projection transferred is $\Pi_0 = \{R, A, B, v, w, x, q, k\}$. Here and in subsequent steps, k is included since it is the parent of all affected bubbles and arcs (therefore, an element of the *parent context*).

Revision 2. Ensuing from revision 0, Bob removes q and x, and inserts arc s and bubble z, all under ambition $\{(f_B, true)\}$. The delta is $\Pi_2 = \{q, x, s, z, B, k\}$. In this case, B

7 Bubbles and arcs essentially correspond to labeled directed graphs, but have been renamed here in order to avoid confusion with internals of the conceptual framework.

is included because its low-level option f_B is referenced in the visibilities of newly inserted domain model elements (1-context).

Revision 1. Alice concurrently removes arc q and bubble x, and adds arc p, under ambition $\{(f_A, true)\}$. The delta is $\Pi_1 = \{q, x, p, w, A, k\}$. Here, w is included as an element cross-referenced by the inserted arc p (1-context).

Merging Revisions 1 and 2. Bob was the first to finish, so Alice receives an *out of date* error when attempting to push. Upon pulling revision 2, the incoming delta is merged. The visibility of arc q and bubble x has been concurrently modified, such that $v^P_{q,x} = r_{0.0} \wedge \neg(r_{1.0} \wedge f_A)$ and $v^{\Pi}_{q,x} = r_{0.0} \wedge \neg(r_{2.0} \wedge f_B)$. Using the base visibility $v^b_{q,x} = r_{0.0}$, the three-way visibility merge function (**12.5**) evaluates to $v'_{q,x} = \mu(v^b_{q,x}, v^P_{q,x}, v^{\Pi}_{q,x}) = r_{0.0} \wedge \neg(r_{1.0} \wedge f_A) \wedge \neg(r_{2.0} \wedge f_B)$. Moreover, the visibility of z is raw-merged in order to obtain Bob's insertion under feature ambition $\{(f_B, true)\}$.

Revision 4. Alice adds bubble y, arcs t and u under feature ambition $\{(f_A, true)\}$. Therefore, $\Pi_4 = \{y, t, u, w, x, A, k\}$. Here, w, x, and A are 1-context elements.

Revision 3. Ensuing from revision 1, Bob concurrently deletes bubble w and arc s under feature ambition $\{(f_B, true)\}$. The transferred delta is calculated as $\Pi_3 = \{w, s, z, B, k\}$. Here, bubble z is a 1-context element cross-referenced by s.

Merging Revisions 3 and 4. Bob receives an *out of date* error when attempting to push revision 3. As a consequence, the incoming delta belonging to revision 4 is merged. In the master repository, the visibilities of bubble y, arcs t and u added by Alice are raw-merged: $v'_{y,t,u} = r_{4.0} \wedge f_A$.

Lessons Learned. In this example, we have focused on the internals of low-level transaction management, delta calculation, visibility update, and visibility merging. It is important to notice again that the Alice and Bob do not get accosted with these details, since they operate in their local workspaces using the VCS abstractions COMMIT, PUSH, and PULL, which hide complexity from them.

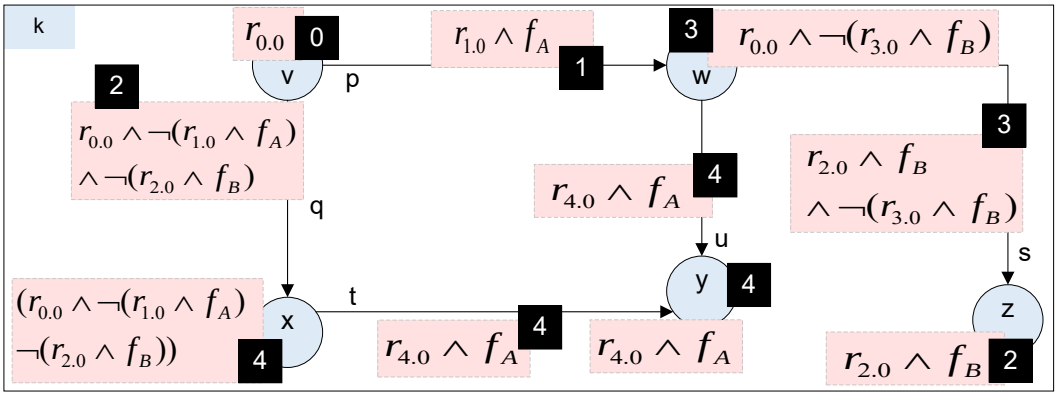


Figure 12.6: Example product space including visibilities (red boxes) and transaction numbers (filled black boxes, white font). Based on [SW16a, Figure 5].

The example also shows that visibility merging produces the intuitive result; the concurrent deletions of q and t under different ambitions as well as Alice's insertions of u and y have been combined, while the logical scopes – the ambitions specified by the users – have been maintained. During delta calculation, the smallest subset necessary to describe the changes in a self-contained way is yielded, reducing synchronization traffic to a minimum.

12.6 Related Work

The here presented extensions, which add multi-user support to the conceptual framework presented in the previous chapters, are related to other approaches described in the literature. On the one hand, the extensions are inspired by *distributed version control systems*. On the other hand, conceptually different solutions for *collaborative SPLE* have been proposed. Moreover, related approaches to *change-oriented versioning* offer collaboration support on the basis of different *transaction* management mechanisms and/or different (pessimistic or optimistic) *synchronization* strategies.

Distributed VCS. The distinction between local and remote transactions used in the presented extension was borrowed from distributed version control systems (DVCS), which extend centralized VCS such as Subversion [CFP04], where every transaction is a remote transaction, by allowing for multiple, distributed copies of a repository.

Many DVCS, especially those employed in the open source community, follow an *open* replication strategy that does not define a global master repository per se, but technically relies on a *peer-to-peer* synchronization principle, where each repository may push and pull from a dedicated origin. Examples include *Git*[8] and *Mercurial*[9]. In this way, arbitrary hierarchies of repositories can be built, such that at the same time, higher-level management of repositories, e.g., through hosting platforms, becomes relevant. In contrast, the here considered framework extension follows a *closed* strategy that allows for only one level of replication (*singleton master* repository).

The concepts contributed here advance the state of the art in DVCS in two ways. On the one hand, support for logical variants has been added by integrating SPLE metaphors; by providing the possibility of combining independently developed features (i.e., *intensional versioning*), the approach at hand goes considerably beyond *branches* and *forks* offered by centralized and distributed VCS. On the other hand, *models* are versioned as structured artifacts rather than taking their text-based serialization as a basis for line-oriented version control.

Collaborative SPLE. It seems natural to build support for collaborative SPLE on top of existing general purpose version control systems. Tools following the *clone-and-own* approach (see Section 5.4.1) natively support multi-user operation. Approaches that explicitly address collaborative SPLE include [RCC13; Pfo+16; LELH16]. They have, however, in common that they organize product-level variation *extensionally* by explicitly persisting branches rather than relying on a fine-grained decomposition of the product.

8 https://git-scm.com/
9 https://www.mercurial-scm.org

The component-based approach to SPL versioning described in [Tha12a] relies on change propagation at configuration level, realizing unfiltered collaborative SPLE. A solution for semi-automatic backward propagation of product-specific changes to the product line is presented. However, instead of connecting visibilities of updated elements to an *ambition* as provided here, manual visibility updates are required.

Revision Graph Management. Open DVCS that follow a peer-to-peer replication strategy, such as *Git* [Cha09], allow for an arbitrarily deep multi-level organization of nested revisions.

The collaborative revision graph metamodel introduced in Section 12.2.1 represents a special case of a *two-level* organization (cf. Section 4.2.2). The first level, formed by public revisions, is a totally ordered directed acyclic graph (TODAG), whereas the private revisions contained on the second level correspond to a sequence. A similar two-level organization is applied, among others, in *CVS* [Ves06], where the restriction of a totally ordered first level is suspended in favor of parallel branches. By intention, branches are not supported by the here considered conceptual framework as the variant dimension is covered by an orthogonal feature model.

A problem related to the hierarchical organization of revisions is their *labeling*. The conceptual framework introduced above uses a two-level scheme of the form `public.private`, where both components are created by counters. In contrast to related approaches, public revisions are incremented not by order of finishing but by starting events of remote transactions. As a consequence, finished revisions are not necessarily ordered by commit date, which potentially causes confusion. In the literature, two other methods can be found to circumvent this problem: First, ordering by finishing events of remote transactions, as applied in the centralized VCS Subversion [CFP04]; then, however, the prefix of pending private revisions must be represented by a placeholder until the definite public revision number becomes available upon push. Second, replacing integer-valued revision numbers by randomly generated identifiers or *hashes*, as performed in Git [Cha09], does not suggest any linear order among revisions to the user.

Transactions in Change-Oriented Version Control. The *Uniform Version Model* (UVM) (cf. Section 8.4 and [WMC01]) is a descendant of *change-oriented versioning* (CoV), which was implemented in the version control system EPOS [Mun93]. The design of a collaborative component for EPOS is presented in [CM91]. EPOS and the framework at hand have in common a low-level transaction layer assigning transaction identifiers to versioned elements. *UVM*'s *layered architecture* [WMC01] extends the low-level transaction layer of the database versioning system EPOS. Transactions are manifested in visibilities by means of *transaction options*, which coarsely correspond to *public revision options*. These are, though, not shown to the user as an explicit VCS artifact, e.g., in the form of a revision graph.

As another difference, in EPOS, transactions may be nested in a tree rather than being restricted to a two-level layout; user modifications are allowed only in leaf transactions. Synchronization is orchestrated by a *propagation* mechanism between different workspaces. In contrast, the new conceptual framework separates filtered editing (check-out/commit) from synchronization (pull/push) and transfers symmetric deltas only on demand.

Pessimistic Synchronization. The aforementioned EPOS system [Mun93] offers (but is not restricted to) *pessimistic synchronization* by inhibiting multiple transactions that are intended to be run under mutually overlapping logical ambitions. Such strategy is not applicable to the here considered dynamic editing model, where the ambition is specified at commit time. Therefore and in order to enable non-restrictive collaboration, the presented framework extensions rely on optimistic synchronization. Conversely, following the pessimistic paradigm, EPOS does not require three-way merging.

In [Kel17], a distributed model versioning strategy based on the lock-modify-unlock paradigm is presented. The approach is feasible with, but does not explicitly address, model-driven product lines.

12.7 Summary

The key contribution of this chapter consists in the multi-user extension of a conceptual framework that enables (single-user) MDSPLE on the basis of a filtered editing model. To this end, the existing repertoire of operations CHECKOUT and COMMIT has been enhanced by PULL and PUSH, which synchronize different copies of local repositories with the help of a dedicated master repository. The operations are defined based upon a *collaborative revision graph*, which organizes remote and local transactions. The central remote repository organizes the transaction history in a log. The artifacts transferred along with the operations pull and push are *symmetric deltas*, which are projected from the entire multi-version domain model based upon transaction identifiers.

When referring back to the requirements stated at the beginning of this thesis in Section 2.3, the presented contributions help satisfy **R17** (*multi-user version control*) and **R18** (*collaborative SPLE*). By adopting Git's pull/push model, the framework adheres to the *distributed* paradigm. Furthermore, *optimistic* synchronization is applied to the domain model, to the feature model, as well as to the (transparent) mapping in between.

As soon as several users apply concurrent modifications to the same versioned element(s), conflicts at product level may arise. To resolve these, a generic *three-way merge* procedure has been introduced, which is, firstly, non-interactive by not allowing the user to intervene in merge decisions, and secondly, context-free in a sense that it does not guarantee the well-formedness of workspace contents. This type of consistency control is supposed to be orthogonal to the here considered problem of collaborative editing. *Product well-formedness analysis* is therefore addressed by the subsequent chapter.

As proof of concept, a client/server application that implements the collaborative editing model and its underlying extensions to the framework is presented in Chapter 14.

Before software can be reusable
it first has to be usable.

RALPH JOHNSON

Chapter 13

Metadata Management and Well-Formedness Analysis

Abstract

The description of the conceptual framework is concluded with two closely related topics, metadata management and well-formedness analysis of products presented in the workspace. Workspace metadata subsumes information that must be organized in addition to the user-visible workspace contents. This information in turn depends on the concrete product dimensions employed as well as on the (single or multi-user) execution mode. We formalize by Ecore-compliant metamodels the metadata for the versioned file system and for the feature model. By the same dimensions, we organize the description of well-formedness conditions by means of which conflicts are detected in the product to be checked out at the beginning of an editing model iteration. The conflict handling employed in the framework follows a product-based a-posteriori paradigm using default resolution strategies, whose properties considerably differ from related analysis techniques described in the literature.

Contents

13.1 The Rough Edges of the Conceptual Framework

The preceding chapters have dealt with specific parts of the conceptual framework in isolation and have left some design decisions, which refer to products being developed by the user in the *workspace*, open. The workspace contains the artifacts the users of a tool following the approach at hand are supposed to read and manipulate during the greater part of their working time. So far, we have assumed that the workspace is physically isolated from the repository. Nevertheless, questions like the following demand for a more precise definition of this concept:

– How is the cut-out of the local file system, which corresponds to the workspace in the domain model dimension, managed? Typically, VCS allow that files are dynamically *added* to and *removed* from version control. Furthermore, there is also the possibility to *ignore* files. To support such operations, it is necessary to keep track of which part of the local file system is currently mapped to the repository.

– Where is the workspace feature model, which is made available for modification within an editing model iteration, located? In which format is it persisted, and how does it differ from the repository-internal (i.e., extrinsic) representation?

– In the algorithmic descriptions in Section 11.3, statements of the form "memorize the current choice for a subsequent commit" have been used. Furthermore, the generalized editing model presented in Section 11.6 assumes that the ambition may also be defined before commit. How are the *memorized choice and ambition*, temporarily valid within one local transaction, represented?

– The COMMIT operation has been defined in such a way that for unaffected product dimensions, no version membership needs to be specified—for instance, in case the domain model remains unmodified, the user need not be asked for a feature ambition. This raises the question how to *detect modifications*. (In graphical VCS back-ends, it is also common to display to the user the files having been modified within the current transaction.)

– In Chapter 12, collaborative versioning has been explained; it is based on a low-level transaction layer that extends both the master and client repositories. How are the local read/write transaction numbers managed? How is the transaction log belonging to the master repository represented? How is the non-permanent connection from client to master organized in order to be established on demand?

– Two properties of the conceptual framework may give rise to *product well-formedness conflicts*, which impede the filtered repository contents from being completely and/or consistently exported into the workspace: On the one hand, when combining multiple *optional features*, which do not exclude each other according to the feature model rules, variability conflicts such as static feature interaction (cf. Section 5.5.1) may occur. On the other hand, *collaboration* may cause similar types of conflicts, provided that the three-way merge strategy presented in Section 12.4 does not consider context-sensitive correctness. The internal representation as well as the precise conditions for the detection of different types of conflicts remain to be introduced formally.

From this list of items, we deduce two important roles for the workspace which need to be formally defined in a more detailed way. On the one hand, *metadata management* comes into play. On the other hand, by supporting optional features as well as through collaboration, *product well-formedness conflicts*, which prevent the selected version from being consistently represented in the workspace, may occur. These need to be presented to the user in a meaningful way, in order to help him/her resolve them.

The remainder of this chapter is structured in the following way. The two aforementioned roles, metadata management and conflict detection, are discussed in Sections 13.2 and 13.3; the sections are structured along the product space dimensions (file hierarchy consisting of text and EMF files, as well as feature model). In Section 13.4, we motivate and explain the product-based a-posteriori analysis and conflict resolution strategy employed in this conceptual framework. Related work, with a focus on well-formedness analysis, is presented in Section 13.5, and Section 13.6 concludes this chapter and thereby Part IV of the thesis. [1]

13.2 Metadata Management

Under *workspace metadata*, we subsume all information that must be organized in addition to the user-visible workspace contents in order to be able to provide for a meaningful and consistent workflow. In the here presented conceptual framework, the *metadata section* is a regular part of the transparently organized repository. Without loss of generality, we assume a one-to-one relationship between local repository and workspace.

In Section 9.2.1, Figure 9.5 on page 165, this has been conceptually prepared by an abstract class Metadata to be contained by each instance of Repository. Figure 13.1 refines the corresponding package core::meta, whose contents are explained throughout the subsequent subsections and thereafter.

13.2.1 Metadata for Distributed Versioning

The framework extensions necessary to support collaborative versioning have been presented in Section 12.3.1, relying on a primitive transaction management layer. The corresponding properties have been incorporated while making an explicit distinction between *client* and *master metadata* (cf. metamodel in Figure 13.1).

1 An excerpt of the text of this chapter has been published in a considerably more concise form in [Schwä+16]. It is, moreover, planned to submit the here presented a-posteriori product-based analysis approach in a more general form for publication [SW18].

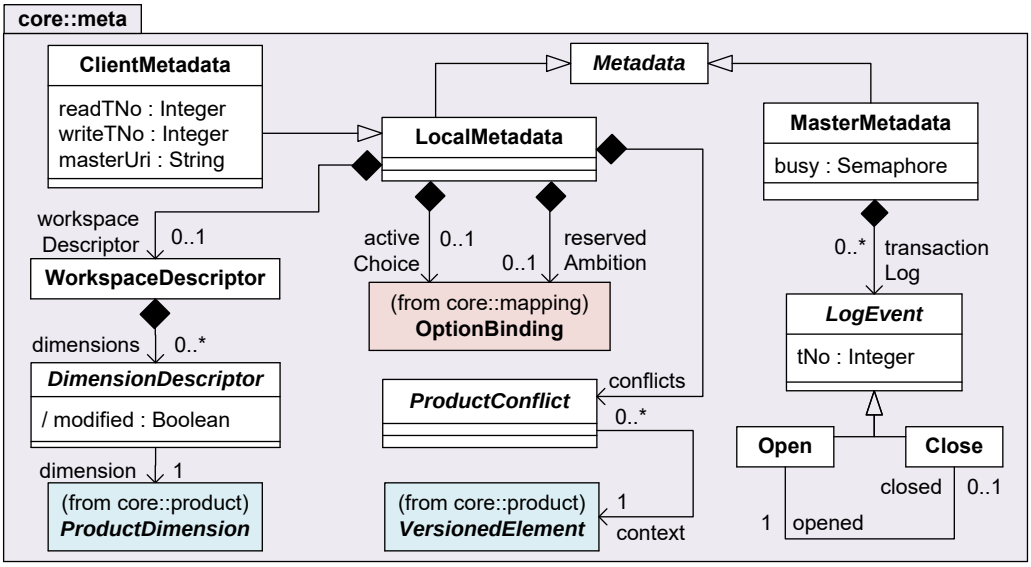


Figure 13.1: General workspace management metamodel. Partly based on [Schwä+16, Figure 9].

Class ClientMetadata contains as attributes a *read* (readTNo) and a *write transaction number* (writeTNo). Furthermore, the aforementioned non-permanent link to the master repository has been conceptually modeled by means of a *uniform resource identifier* (masterUri). It strongly depends on the technical implementation how this information is interpreted; a concrete example is given in Section 14.4.5.

Complementarily, MasterMetadata contains a *transaction log* which is organized as an ordered collection of Open and Close events, both carrying the corresponding transaction number (tNo). Furthermore, it is possible to navigate from a closing event to a corresponding opening event, and, if applicable, vice versa. The attribute busy ensures by a *semaphore* mechanism [HDH02] that no write transaction is executed concurrently with any other transaction on the same master repository instance. Concurrent read transactions, however, are allowed.

13.2.2 Local Workspace Management Metadata

The aforementioned class ClientMetadata specializes LocalMetadata, which in turn represents the metadata section of the lone repository copy in single-user scenarios. Local metadata augment the information represented by the workspace itself by data necessary for organizing local transactions. As the metamodel shown in Figure 13.1 suggests, they consist of four components.

First of all, the activeChoice persists the effective choice derived from user-based version selections made for the current local transaction. It is represented as an instance of OptionBinding, whose bound options refer to the version space of the local repository. On the one hand, the active choice is needed during COMMIT in order to reconstruct the checked-out workspace. On the other hand, it is purposeful to present the active choice to the user, making him aware of the selection of features that describe the current read filter.

Optionally, the reference reservedAmbition is used for *static filtered editing* as well as for *earlier ambition specification*; see Section 11.6. Like the choice, it is serialized in non-completed form, i.e., in its state before applying preferences and defaults to it.

The *workspace descriptor* augments the visible parts of the workspace with management metadata that differ among specific product dimensions; concrete subclasses of DimensionDescriptor are provided in the two following subsections. The derived attribute modified must be defined in order to state whether any artifact of the corresponding dimension has been edited during the current local transaction. This is necessary to detect situations in which COMMIT is inapplicable (i.e., if no dimension is modified), and in which ambition specification can be omitted for a certain version dimension (i.e., in case the domain model is unmodified, no feature ambition is required; cf. Algorithm 11.4 on page 231).

Last, the local metadata section contains a list of *product conflicts*, concrete instances of which are listed in Section 13.3. Each conflict refers to a dedicated *context*—the element that is supposed to be reported as *conflicting* to the user. After each commit, this list is purged.

13.2.3 Dimension Descriptor for Versioned File System

When confining to the versioned file hierarchy dimension, the workspace corresponds to a cut-out of the user's local file system that can be uniquely defined by a *root folder*. In instances of the *file hierarchy descriptor*, whose metamodel is depicted in Figure 13.2 as Ecore-compliant class diagram, this is covered by a corresponding attribute workspaceRootUri. This determines where in the local file system to apply the corresponding transformations IMPORT and EXPORT in order to convert between the intrinsic and the extrinsic product representation. [2]

It is also a responsibility of the file hierarchy descriptor to keep track of which files and folders are currently under version control, which of them are explicitly ignored, and whether or not the contents of a specific versioned file have been modified in the course of the running local transaction. These responsibilities are covered by corresponding attributes, where it is assumed that the internal state of a file is processed by a *hash function*. The value of the attribute hash corresponds to the result of applying this function to the corresponding

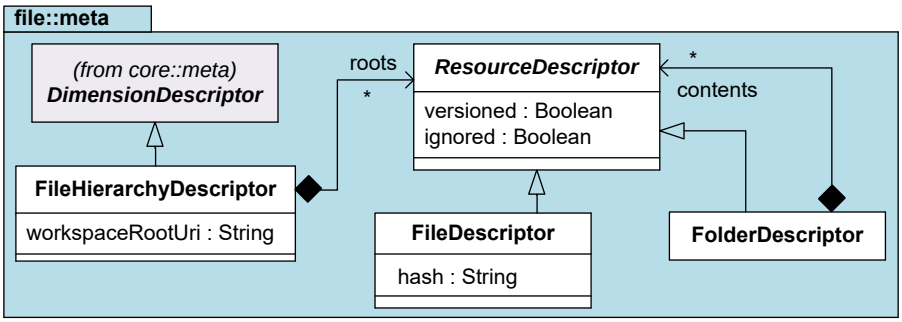


Figure 13.2: Local metadata for file hierarchy available in workspace.

2 Conversely, we assume *holistic versioning* of the file hierarchy; i.e., selections of sub-directories in the versioned file hierarchy are not supported at check-out.

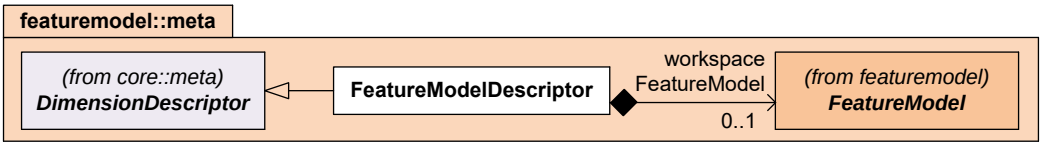


Figure 13.3: Local metadata for feature model in workspace.

file right after check-out. In case hashing the file contents in their current state delivers a different result, the file, hence the superordinate file hierarchy, are considered as modified.

13.2.4 Dimension Descriptor for Feature Model

In contrast to the domain model – which is represented as a file hierarchy as shown above – the feature model is not supposed to be accessible by third-party tools. Most notably, it must be protected from external modification in order to disallow inconsistencies. Therefore, in the conceptual framework, it is assumed that the feature model is only editable by means of a specialized editor, which particularly implements the auxiliary operations SAVEFEATUREMODEL and DELETEFEATURE as defined in Section 11.3.2 on page 228. Furthermore, the editor must ensure that the feature model is always consistent with its low-level mapping; cf. Table 9.2 on page 174.

For these reasons, the workspace version of the feature model is hidden in the local metadata section, being directly contained by the *feature model descriptor*; see Figure 13.3. As a consequence, local feature model metadata are self-describing.

The metamodel clarifies another internal design decision: For the feature model, the intrinsic and extrinsic representation are equal[3] in the sense that there is no specific single-version feature metamodel defined. As a consequence, the transformations IMPORT and EXPORT may be easily implemented by the identity function, with one exception: during export, all visibilities are removed from the workspace feature model. Also after export, well-formedness conditions ensure that the feature model represents a single consistent version, despite being represented as an instance of a multi-version metamodel. Conditions describing well-formedness violations are listed in Section 13.3.3.

13.3 Product Well-Formedness Conflicts and Conditions

Also part of the local metadata sections are *product conflicts*. Rules for the detection of these are here called *well-formedness conflict conditions*, since they cannot guarantee that the checked artifacts are semantically meaningful, but only syntactically valid inasmuch as they can be exported into the local workspace in a way that allows external tools to open the models, and to report potential semantical errors to the user(s) in the usual way.

Below, specific types of product conflicts are explained together with their well-formedness conflict conditions. In case a violation is detected on an arbitrary instance of the context class, an instance of the corresponding subclass of ProductConflict is created and appended

3 Though, the instances, of course differ. In particular, workspaceFeatureModel represents the checked-out state of the corresponding version dimension, filtered by the active workspace choice.

to conflicts. There are three product dimensions (or building blocks thereof) relevant for product conflicts: *ordered collections*, which occur in both EMF and text files, *extrinsic EMF model instances* as parts of the versioned file hierarchy, and finally, *feature models*.

The formal description of each conflict is divided up into the following four parts:

Context. The class for whose instances the conflict may occur.

Condition. A formal description of the condition that must be fulfilled for the conflict to be present. If adequate, the condition is phrased as an OCL invariant that holds unless the conflict has been resolved.

Message. The message that is displayed to the user based on a given unresolved conflict instance. OCL-compliant navigating expressions in curly braces are replaced by the actual values of the corresponding properties of the conflict instance.

Resolution. Available conflict resolution alternatives, including a description of how conflict resolution is expressed in concrete conflict instances.

13.3.1 Generic Conflict Condition for Sequences

According to Section 10.3.2, the EXPORT transformation for multi-version sequences – internally represented as directed graphs – consists of two steps, filtering the graph by the specified choice and linearizing the graph according to Algorithm 8.1 (see page 145). This algorithm contains an interactive step, which is here modeled as a product conflict. Figure 13.4 provides a structural view.

Conflict 1 (*Order*)— The linearization of a sequence is ambiguous.

Context: core::product::oc::OrderedCollection

Condition: Algorithm 8.1 is interrupted by an input prompt that requires to choose one among multiple successorCandidates after a given previousVertex.

Message: "The element order in {contextCollection} is ambiguous. Multiple candidates may immediately succeed {previousVertex.occurringElement}: {successorCandidates.occurringElement}."

Resolution: One of the elements available in successorCandidates is chosen and the corresponding vertex is set as selectedSuccessor. A new edge is inserted between previousVertex and selectedSuccessor, and edges in the opposite direction, if present, are removed. Then, the algorithm is resumed.

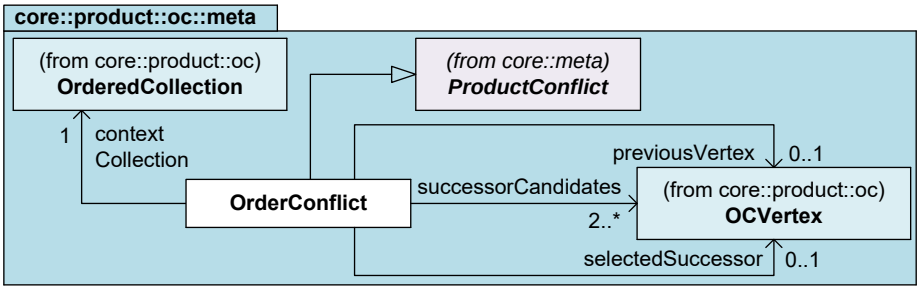


Figure 13.4: Extended metadata for product conflicts in ordered collections.

In case the conflict appears for the first element of the collection, the corresponding remarks referring to previousVertex are omitted from the definitions given above.

In the assumed product model, sequences represented by OrderedCollection exist in two concrete places: For defining the line order in text files (cf. Section 10.5.1) and for ordered multi-valued collections in extrinsically represented EMF model instances (see Section 10.6.1). Accordingly, Conflict 1 is checked in these places.

13.3.2 Conflict Conditions for Extrinsic EMF Model Instances

The semantical validity of EMF model instances present in the workspace is supposed to be ensured by external mechanisms, such as metamodel-specific OCL constraints or tool-internal validation routines. In order to guarantee that EMF models can be represented in their abstract syntax, allowing to serialize models, e.g., in XMI, the metamodel-agnostic well-formedness conflict conditions listed in the following come into play. The structural perspective onto these conflicts is provided in Figure 13.5.

Conflict 2 (*Object Classification*)— The class of an EMF object is ambiguous.
Context: file::emf::Object
Condition: `contextObject.classes->size() > 1`
Message: "The class of object {contextObject} is ambiguous. Multiple candidates are available: {contextObject.classes}."

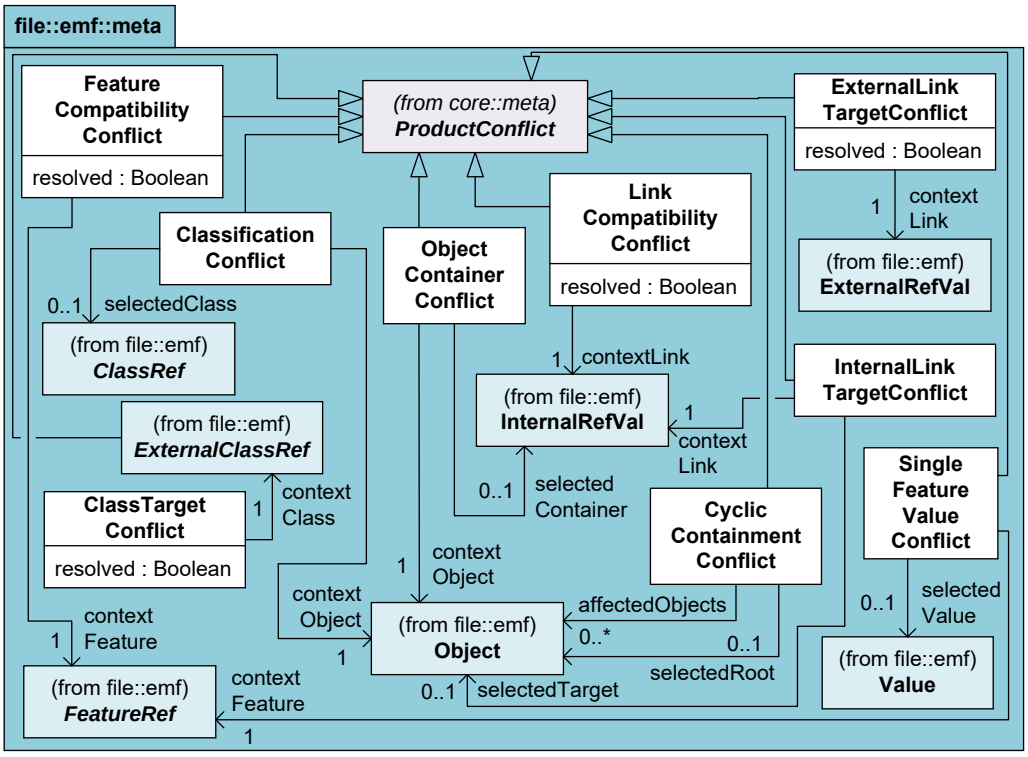


Figure 13.5: Extended metadata for product conflicts in EMF model instances.

Resolution: One of the classes available in contextObject.classes is interactively chosen and set as selectedClass. All other candidate classes are removed.

Conflict 3 (*Class Target*)— The externally defined class of an EMF object is not available in the local modeling environment.

Context: file::emf::ExternalClassRef

Condition: In the modeling environment, no package with URI contextClass.packageUri is defined[4], or the package does not contain a class named contextClass.className.

Message: "The class {contextClass.packageUri}::{contextClass.className} is not available in the modeling environment."

Resolution: A legal class available in the modeling environment is selected interactively and the packageUri and className are updated accordingly. After having resolved the conflict, property resolved is set to true.

Conflict 4 (*Structural Feature Compatibility*)— An EMF object contains a value for a structural feature that is not defined for the object's class in the local modeling environment.

Context: file::emf::FeatureRef

Condition: In the modeling environment, no feature named contextFeature.featureName is defined for class contextFeature.object.classes->at(0) or one of its superclasses.

Message: "The feature {contextFeature.featureName} is not defined for class {contextFeature.object.classes->at(0).className}."

Resolution: A legal feature of the object class is selected and the featureName are updated accordingly. After having successfully resolved the conflict, property resolved is set to true.

Conflict 5 (*Object Container*)— The container of an EMF object is ambiguous.

Context: file::emf::Object

Condition: `contextObject.incomingRefVals->filter(oclIsKindOf(ContainmentRefVal))->size() > 1`

Message: "The container of object {contextObject} is ambiguous. Multiple candidates are available: {contextObject.incomingRefVals->filter(oclIsKindOf(ContainmentRefVal))}."

Resolution: One of the suggested candidates is chosen and set as selectedContainer. The incoming containment links from all other candidate objects are removed.

Conflict 6 (*Cyclic Containment*)— The containment hierarchy formed by several EMF objects contains a cycle.

Context: file::emf::Object

Condition: `affectedObjects->forAll(o | o->closure(features.values->filter(oclIsKindOf(ContainmentRefVal)).target->includes(o))`

Message: "The following objects form a containment cycle: {affectedElements}."

Resolution: From the list of affected objects, one is chosen to represent the top of the containment hierarchy and set as selectedRoot. Incoming containment links ensuing from any other affected object are removed.

4 It depends on the employed modeling framework how references to the metamodel are resolved. For instance, in the Eclipse Modeling Framework, the global *package registry* for metamodels can be queried.

Conflict 7 (*Single-Valued Structural Feature Value*)— Multiple values exist for a single-valued structural feature. [5]

Context: file::emf::FeatureRef

Condition: Provided that the context feature reference represents a valid structural feature with an upper multiplicity bound of 1, the following precondition is checked:
`contextFeature.values->size() > 1`

Message: "Object {contextFeature.object} defines multiple values for the single-valued structural feature {contextFeature}: {contextFeature.values}."

Resolution: From the list of available values (contextFeature.values), one is chosen and set as selectedValue. All other candidate values are removed.

Conflict 8 (*External Link Target*)— The target of an external reference value ensuing from an EMF object is not defined in the local modeling environment.

Context: file::emf::ExternalRefVal

Condition: In the local modeling environment, no object with URI contextLink.targetUri is defined.

Message: "Object {contextLink.featureRef.object} contains a reference to an external EMF object with URI {contextLink.targetUri}. Such object cannot be found in the modeling environment."

Resolution: A legal link target is chosen in the local file system; the value of the link contextLink.targetUri is updated accordingly. After having resolved the conflict, property resolved is set to true.

Conflict 9 (*Internal Link Target*)— The target of an internal reference value ensuing from an EMF object is not defined in the filtered product space.

Context: file::emf::InternalRefVal

Condition: `contextLink.target->oclIsUndefined()`

Message: "Object {contextLink.featureRef.object} contains a reference to an undefined internal object."

Resolution: A legal link target is chosen in the product space and set as selectedTarget. The value of contextLink.target is updated accordingly.

Conflict 10 (*Link Compatibility*)— An EMF object contains a value for a reference whose target is not compatible with the class of its structural feature.

Context: file::emf::InvernalRefVal

Condition: The class contextLink.target.classes.at(0) does neither match the class defined as reference type in contextLink.featureRef nor one of its superclasses.

Message: "Object {contextLink.featureRef.object} contains a link to {contextLink.target}, which is incompatible with the type of the reference {contextLink.featureRef}."

Resolution: A legal link target is chosen and the value of contextLink.target is updated accordingly. After resolution, property resolved is set to true.

5 A more restrictive form of this condition would check whether the precise upper and lower bounds of the corresponding structural feature are violated. This relaxed form is oriented towards EMF, which technically merely distinguishes between single-valued (upper bound of 1) and multi-valued properties.

Together with Conflict 1 (*order*), the list of EMF-specific conflicts completes the description of conflict conditions in the versioned file system. On files and folders, no conflicts may occur since resources are unambiguously identified based upon their parent and name. For text files, exclusively (line) order conflicts are relevant.

13.3.3 Conflict Conditions for Feature Models

The second product dimension consists in the *feature model*, which is versioned by the revision graph when assuming the default three-layered architecture introduced in Chapter 9. At that point, *version rules* for the semantical correctness of single-version feature models were defined. One has to distinguish these from syntactical *well-formedness* rules, whose violation would disallow the representation of a single-version feature model in the workspace (see Section 13.2.4). The conflicts listed below can be caused by concurrent modifications to the same feature model instance. A structural view, provided in Figure 13.6, complements the textual description.

Conflict 11 (*Root Feature*)— The root feature of the feature model is ambiguous.
Context: featuremodel::FeatureModel
Condition: `contextModel.roots->size() > 1`
Message: "The root of the feature model is ambiguous. Multiple candidates are available: {contextModel.roots.root}."
Resolution: One of the suggested candidates is chosen and the corresponding instance of RootRelationship is set as selectedRoot. All remaining candidate instances of the same class are removed.

Conflict 12 (*Parent Feature*)— The parent of a feature is ambiguous.

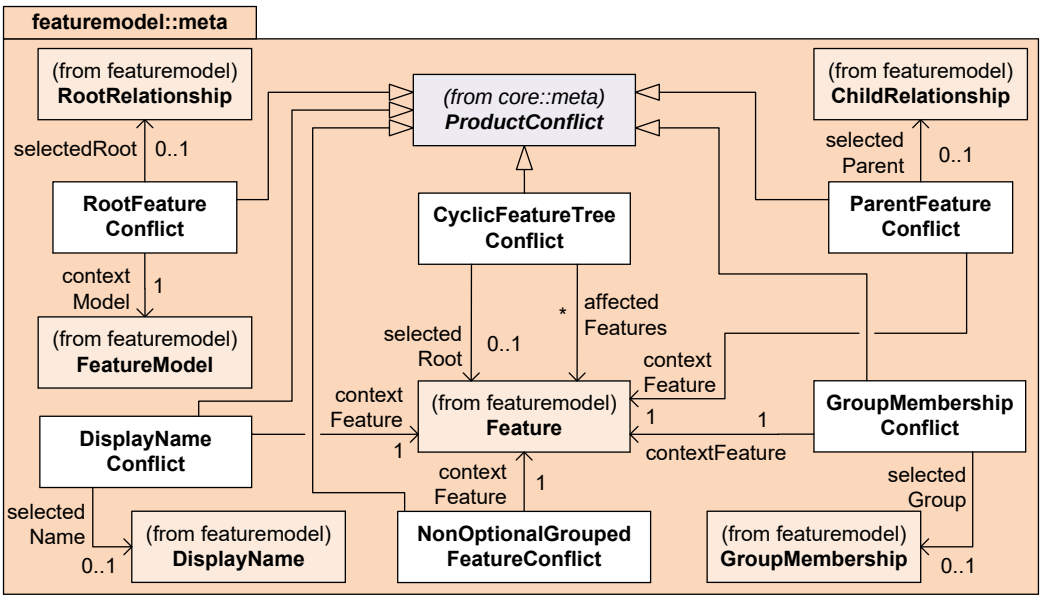


Figure 13.6: Extended metadata for (syntactic) product conflicts in feature models.

Context: featuremodel::Feature
Condition: `contextFeature.parents->size() > 1`
Message: "The parent of feature {contextFeature} is ambiguous. Multiple candidates are available: {contextFeature.parents.parent}."
Resolution: One of the suggested candidates is chosen and the corresponding instance of ChildRelationship is set as selectedParent. Remaining candidate parents are removed.

Conflict 13 (*Cyclic Feature Tree*)— The containment hierarchy formed by several features contains a cycle.
Context: featuremodel::Feature
Condition: `affectedFeatures->forAll(f | f->closure(children.childFeature)->includes(f))`
Message: "The following features form a cycle: {affectedFeatures}."
Resolution: From the list of affected features, one is chosen to represent the root of the tree and set as selectedRoot. Incoming parent relationship links from any other affected feature are removed.

Conflict 14 (*Group Membership*)— A feature is part of multiple groups.
Context: featuremodel::Feature
Condition: `contextFeature.groupedBy->size() > 1`
Message: "{contextFeature} is in multiple groups: {contextFeature.groupedBy.group}."
Resolution: One of the suggested group candidates is chosen and the corresponding instance of GroupMembership is set as selectedGroup. All remaining candidate parents are removed from the reference contextFeature.groupedBy.

Conflict 15 (*Non-Optional Grouped Feature*)— A feature part of any group is not optional.
Context: featuremodel::Feature
Condition: `not (contextFeature.groupedBy->size() > 0 implies contextFeature.mandatory->oclIsUndefined())`
Message: "Feature {contextFeature} is part of a feature group but not optional"
Resolution: The feature is made optional or it is removed from the group(s) containing it.

Conflict 16 (*Display Name*)— Multiple display names are defined for a feature.
Context: featuremodel::Feature
Condition: `contextFeature.names->size() > 1`
Message: "For the same feature, multiple names are defined: {contextFeature.names}."
Resolution: One name is chosen and set as selectedName. All remaining candidate instances of DisplayName are removed.

13.4 A-Posteriori Product-Based Well-Formedness Analysis

After having explained the conditions, the structure, as well as resolution alternatives for specific types of conflicts that can occur in the versioned file system and/or in the feature

model organized in the workspace, the question *when* and *how* the analysis and the repair are applied in the conceptual framework remains to be answered. The here presented approach performs an *a-posteriori* (i.e., after filtering) *product-based* (i.e., only the checked-out version is considered) analysis and relies on *default resolution strategies* that can be manually revised later; see design decision **D12** on page 137. In this section, we delineate the chosen solution from other candidate analysis strategies and justify why it integrates better with the underlying filtered editing model. After the corresponding explanations, an example of a concrete conflict handling workflow is presented.

13.4.1 Three Candidate Analysis Strategies

A general and intuitive requirement for all analysis techniques is that the analysis results be presented to the user in a meaningful and comprehensible way. As explained in Section 5.5.2, well-formedness analysis is preferably applied during *domain engineering* in a *family-based* way rather than on concrete product instances. In the conceptual framework assumed here, however, domain engineering is essentially realized by filtered application engineering—a design decision that also affects analysis.

The product version to be analyzed is specified along with the operation CHECKOUT. Therefore, the activities *well-formedness analysis*, i.e., the detection, and *well-formedness repair*, i.e., the resolution of conflicts must be arranged with the existing activities FILTER and EXPORT, which, taken together, realize the conversion of a multi-version product space available in the repository into a single-version workspace. According to Figure 13.7, there are at least three candidate orders in which the activities might be arranged.

A-Priori Family-Based Analysis. When adopting this strategy, conflicts are detected and resolved based on the extrinsically represented multi-variant domain model, which is exposed to the user for this purpose. The operations FILTER and EXPORT may correctly assume that their input is syntactically well-formed. Essentially, well-formedness management is not entangled with the operations part of check-out, such that it may be applied between subsequent editing model iterations, but not as a part thereof.

A-Priori Product-Based Analysis. In many scenarios, the well-formedness not of the whole product line but only of the variant to be derived is of interest. Then, analysis and repair may be moved after *filter*, such that they are applied in a single-variant context. The representation upon which the conflict conditions are evaluated, however, is still the internal (i.e., extrinsic) repository format, which differs from the (intrinsic) workspace representation, which is generated as output of the concluding EXPORT transformation.

A-Posteriori Product-Based Analysis. When compared to the a-priori approach, check-out and well-formedness management are entangled to a greater extent. Analysis is still applied in the extrinsic representation, but for offering the user the possibility of conflict resolution, a *preliminary* workspace representation is created. All conflicts are supposed to be resolved until the subsequent COMMIT. Special attention must be paid in order to guarantee that the EXPORT transformation is applicable; this may not be the case if conflicts have been detected.

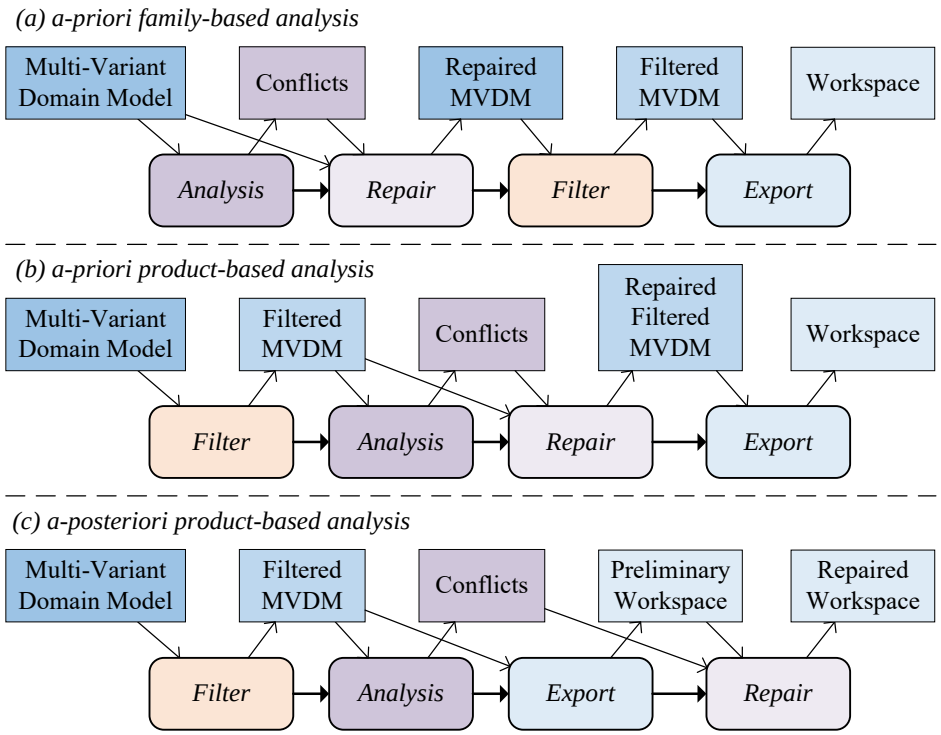


Figure 13.7: Different conceivable product analysis workflows in terms of the conceptual framework.

From the list of viable analysis strategies, exclusively the *a-posteriori product-based* approach visualized in Figure 13.7(c) has been chosen for further consideration in the conceptual framework (and also for implementation in *SuperMod*, see Section 14.6). This decision needs to be justified.

In both a-priori approaches, well-formedness conditions are validated based upon the extrinsic representation *before* being exported into the local workspace. The designated user, however, prefers to get notified about well-formedness violations based upon the familiar intrinsic representation, which is available only *after* having exported the filtered product. Therefore, *a-posteriori* approaches are here considered as more user-friendly.

The question whether to employ product-based or family-based analysis must be answered in a twofold way. On the one hand, when browsing the available literature, there is an agreement that family-based analysis techniques are preferable because they are more precise than repeated product-based analysis, but also because they may more reliably ensure that the whole product line is in a well-formed state.

On the other hand, when considering the specific requirements and circumstances under which the conceptual framework is intended to be employed, there are also two arguments against. First, family-based analysis is irreconcilable with the a-posteriori approach since the multi-variant domain model cannot be represented in a form that is meaningful and comprehensible to the user. Second, the filtered editing model represents *product-based product line creation*—a paradigm switch to *family-based product line analysis* would seem contradictory and confusing to the user.

Seen from the requirements perspective, the ideal solution would have been *a-posteriori family-based analysis*, which is, however, not feasible due to the distinction between intrinsic and extrinsic product representation, whose existence in turn is justified by several arguments (namely *unconstrained variability* and *tool independence*; see Section 2.8).

13.4.2 A Check-Out Operation that Enforces Product Well-Formedness

Due to the aforementioned strong entanglement of the operations FILTER and EXPORT with *analysis* and *repair*, well-formedness management is defined as another extension to the operation CHECKOUT, which has been formally defined in Section 11.3.1.

Since a more exact description would require too many assumptions about the implementation, we here explain the extended CHECKOUT operation only informally at a comparably low level of detail. Implementation remarks, especially concerning the presentation of conflicts to the user in the development environment, are given in Section 14.6.

1. Request a *choice* from the user and apply all necessary version consistency checks.
2. FILTER the multi-version product space by the choice; the outcome is here denoted as *filtered product space*.
3. Identify *conflicts* in the filtered product space by matching the corresponding conflict conditions described in Section 13.3 with all applicable context elements.
4. Apply a *default resolution strategy* (see Section 13.4.3 below) to each detected conflict. Extend the generated conflict description by a remark that explains how the conflict has been automatically resolved. The results of this step include a *preliminarily repaired filtered product space* and a list of *enhanced conflict descriptions*.
5. EXPORT the preliminarily repaired filtered product space into the local workspace.
6. For each element of the enhanced conflict set, find the workspace element that corresponds to the contextual instance of the conflict. Attach the enhanced description of the corresponding conflict (including the default resolution remark) to this element.

During the subsequent MODIFY phase, the user may inspect the automatically applied conflict resolutions and, if necessary, correct them using his/her preferred single-version editing tool in the workspace. All manual revisions are connected to a common feature ambition at COMMIT.

13.4.3 Default Resolution Strategies

The semi-formal definition presented above includes one peculiarity that has not been further explained so far, namely *default resolution*. The reason why this mechanism is applied is buried in the following dichotomy: On the one hand, users deal with conflicts in the workspace, which is available *after* exporting. On the other hand, conflicts must be resolved *before* applying the export transformation, since the output would potentially be illegal otherwise. Therefore, the strategy is to apply conflict resolution *both before and after* export. The first run is non-interactive and has the goal to make the product well-formed,

whereas the second and interactive run is controlled by the user, who may revise the effects of non-interactive default resolution for the sake of context-free and semantical correctness.

Subsequently, six viable default conflict resolution strategies, which can be employed in the first and non-interactive run, are listed. They are based upon different assumptions referring to the version space (i.e., commit time, commit author, or feature ambition used).

The Most Recent Change Wins. This strategy is applicable to all product elements versioned by the revision graph. When selecting a suitable element to resolve the conflict (see *resolution* paragraphs in conflict descriptions), the element whose visibility refers to the most recently introduced revision obtains the highest priority.

The Least Recent Change Wins. The inverse strategy, prioritizing "older" revisions.

My Change Wins. In case the source of a conflict is concurrent modifications in collaborative mode, this resolution strategy gives elements that have been inserted or modified locally a higher selection priority than elements touched in a remote transaction.

Their Change Wins. The inverse strategy, prioritizing remotely modified elements.

The More Specific Change Wins. By comparing the mandatory flags of features referenced in the visibilities of candidate elements, "more optional" elements get a higher selection priority assigned. (Intuition: specific changes override general changes.)

The Less Specific Change Wins. The inverse strategy, prioritizing elements whose visibility refers primarily to mandatory features. (Intuition: the change that is present in a greater number of product variants is preferred.)

Random Resolution. Selects a random element. The randomization strategy may also be trivial, e.g., by selecting always the first element from the list of candidates.

The sources of conflict can be manifold, and so can the corresponding default resolution strategies be combined. For instance, a conflict involving both collaboration and the selection of multiple optional interacting features might be resolved by a combination of "their change wins" and "the more specific change wins". It also depends on the user's preferences whether the described strategies are applied *globally* or in a *conflict-specific* way that resolves, for instance, conflicts in the feature model by "most recent change wins" and conflicts in the versioned file system by "more specific change wins", if applicable.

In any case, it remains to be mentioned that default resolution is a *heuristic* strategy that has its limitations. In particular:

- It depends on the organization of the repository whether some of the presented strategies are applicable. For instance, feature-related strategies are not applicable to elements of the feature model, whose visibilities do not refer to feature options. Similarly, the terms "my" and "their change" are accurate in collaborative mode only.
- The strategies may be non-deterministic in some cases. For instance, there may exist several optional features in the visibility of a candidate element. Such situations are resolved by random selection.
- For specific kinds of conflicts, the semantics of default resolution is not clear. For instance, Conflict 4 (EMF *feature compatibility*) and 9 (EMF *link compatibility*) require selections

of domain model elements. These cannot be assessed by the criteria underlying default resolution, e.g., "mine" or "less specific". In such cases, the context element is deleted, and the default resolution message communicates to the user that he/she must manually create a new value or a new link ensuing from the context object during manual revision.

– Potential interaction between different conflicts is ignored. For instance, the default resolution of one conflict may cause another conflict or obviate an existing conflict. Such situations must be analyzed and resolved manually.

13.4.4 Example

In the following, an example based on the running *Graph* scenario is presented. Three sources of conflicts occur. First, a common base version of a feature model is extended with different features by two developers without raising a well-formedness violation; another feature is renamed conflictingly. Second, the developers realize their own optional features in a fashion that causes an order conflict as soon as both features are combined. Third, the two developers conflictingly change the target of a link using different feature ambitions. After synchronization, the conflicts are detected and default-resolved in Bob's workspace. Last, the revised feature and domain model – containing manual corrections to the automatic repairs – are committed and pushed. The explanations given below are visually supported by Figure 13.8.

Base Version. Sub-figure 13.8(a) unveils the transparent contents of the master repository based on which Alice and Bob initiate their work. All contents have been created during remote transaction 1, and different feature ambitions have been used to incorporate modifications specific to the features Graph, Vertices, Edges, and the optional Couleured – incorrectly spelled –, within nested local transactions 1.1 until 1.4. The product space contains a (semantically incorrect) derived unidirectional association elements, which connects class Graph to Vertex.

Alice's Changes. In (b), a single-version view (with all features activated) on Alice's local repository, after the modifications she performed during remote transaction 2, is presented. In the lone local transaction 2.1, the feature model is extended by a feature Weighted, and moreover, Couleured is renamed to Kolored—not the correct spelling either. As modification to the product space, which is performed under ambition $\{(o_{Weighted}, true)\}$, she introduces an attribute weight and changes the target of the association end elements to Edge. The second change is questionable for two reasons. First, vertices should also be considered as elements. Last, the change should be global rather than being specific to feature Weighted.

Bob's Changes. Sub-figure 13.8(c) presents Bob's local workspace contents (with all features activated) after his concurrent remote transaction 3, which contains three nested local transactions. Revision 3.1 is committed in a global scope (i.e., using the empty set as ambition). In the domain model, an abstract class GraphElement with incoming generalizations from Vertex and Edge is introduced. This class is used as the new target of association elements. Additionally, Couleured is renamed to Coloured—the British spelling. In revision 3.2, Bob introduces the feature Labeled

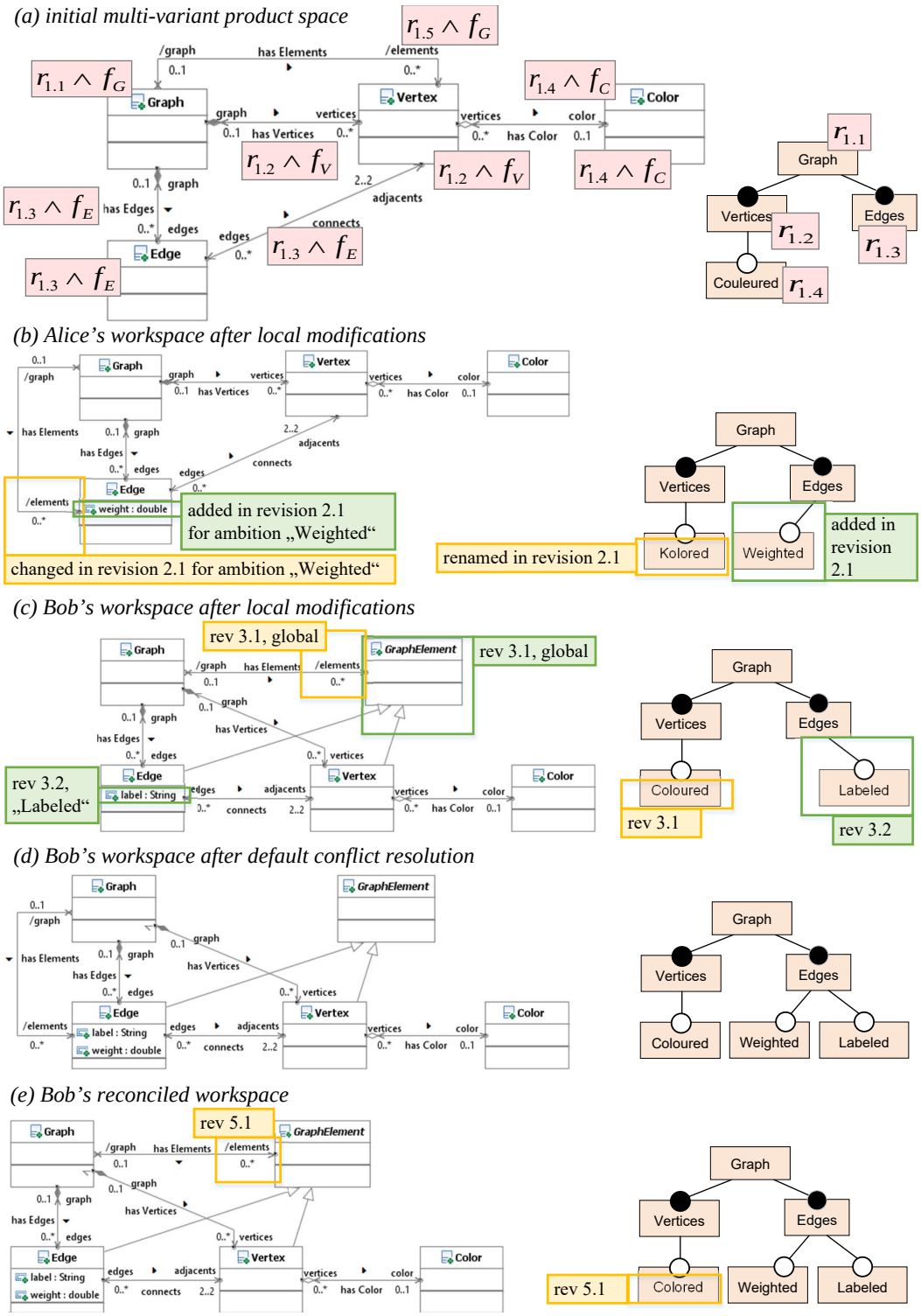


Figure 13.8: Example: Concurrent modifications and product conflict resolution.

and uses this for the ambition $\{(o_{Labeled}, true)\}$, which describes the insertion of an attribute label.

Three-Way Merging. Public revisions 2 and 3 have been created by Alice and Bob as successors of revision 1, such that a concurrent modification is detected, which involves three-way merging (see Section 12.4). Since Alice pushes first, merging takes place in Bob's local workspace as soon as he attempts to push. After merging, the product space is pushed automatically, remote transaction 3 is finished, and the successor transaction 5 is started immediately. [6]

Conflict Detection. During the enforced CHECKOUT operation, Bob selects a choice with all available features, including Alice's feature Weighted, positively selected. In the filtered product space, three conflicts with the following descriptions are detected:

- *Order Conflict*: "The element order in the properties list of class Edge is ambiguous. Multiple candidates may immediately succeed adjacents: {label, weight}." [7]
- *Single-Valued Feature Value Conflict*: "Object elements defines multiple values for the single-valued structural feature type: {GraphElement, Edge}."
- *Display Name Conflict*: "For the same feature, multiple names have been defined: Coloured, Kolored".

Default Conflict Resolution. Let us assume that the following default conflict resolution has been set up: If applicable, "the more specific change wins"; otherwise, "my [i.e., Bob's] change wins". Then, before exporting, the following repair actions would be applied transparently:

- Both changes refer to optional features, so the secondary strategy is applied, such that Bob's insertion obtains a higher priority. The order conflict is resolved by inserting an edge (label, weight) into the underlying collection graph. The linearized order is [..., adjacents, label, weight].
- Being more specific, Alice's change – accidentally connected to the optional feature Weighted – is preferred here: The conflict is resolved by setting Edge as target of the unidirectional association elements. The alternative value for the type specified by Bob, namely GraphElement, is removed although semantically more meaningful.
- The feature name Coloured defined by Bob is chosen, and Kolored is abandoned.

Enhanced Conflict Descriptions. After the default resolutions have been applied, the preliminary filtered domain model is exported into Bob's workspace; the result is shown in Figure 13.8(d). Furthermore, the conflict descriptions, which are made available to Bob, too, are enhanced:

- *Order Conflict*: "[...] The conflict was default-resolved by selecting label."
- *Single-Valued Feature Value Conflict*: "[...] The conflict was default-resolved by selecting Edge."
- *Display Name Conflict*: "[...] The conflict was default-resolved selecting Coloured."

6 Revision number 4 was given to Alice's new public revision, which is, however, not considered any further.
7 In the UML metamodel, attributes and association ends are shared in a meta-reference properties.

Manual Reconciliation. Bob now utilizes this conflict list in order to revise the applied default repair actions. He agrees with the first correction, provided that the order of attributes is not semantically decisive anyway. For the second conflict resolution, however, Bob disagrees with Alice's change, which has been preferred by the strategy. Accordingly, he manually reverts the association target from Edge to GraphElement. Concerning the display name conflict, Bob recognizes that both spellings are incorrect (with respect to American English), and manually defines a fourth value: Colored. The final revised workspace is visualized in a single-version view in Figure 11.5(e). Bob commits this as revision 5.1 under feature ambition[8] $\{(o_{Labeled}, true), (o_{Weighted}, true), (o_{Colored}, true)\}$. He finishes remote transaction 5 with a PUSH.

13.4.5 Retrospective Discussion

Let us recall the presented example while considering different aspects: the amount and complexity of *user interaction*, the degree of *correctness* that is guaranteed, and the *flexibility* of the illustrated a-posteriori product-based well-formedness checking strategy.

User Effort. Considering the first aspect, the amount of *user interaction*, Alice and Bob must intervene only in case default resolution produces an unexpected result. Furthermore, conflicts not relevant for the currently selected product variant are faded out, which reduces complexity. For the reconciliation of conflict resolution, familiar workspace editing tools may be employed without accosting the user(s) with additional tools[9]. The user is, moreover, guided by meaningful conflict descriptions.

Conversely, as the example has demonstrated, the user who revises the preliminary workspace is forced to define a rather specific feature ambition; it may therefore happen that the same or similar reconciliations have to be applied repeatedly for related product variants. As aforementioned, less restrictive well-formedness checking is accepted, however, in favor of systematically following a dynamic product-based product line creation (and, therefore, validation) paradigm.

Correctness. As far as *correctness* is concerned, it must be mentioned that this is only ensured *locally* in the workspace, which is determined by the choice. Although the applied conflict resolutions may affect other variants, potentially all versions described by the ambition, the only variant for which one can really guarantee correctness after applying default conflict resolution is the one available in the workspace.

---

8 This appears to be a rather specific ambition, however, the corrections are not applicable in a more general scope as they depend on product elements connected to all three features bound. Omitting some of the bindings would violate Constraint 6 (see page 226). This problem could have been avoided by organizing the resolution decisions into multiple commits; however, in its current state, the framework requires that all conflicts be resolved in one single iteration.

9 Nevertheless, user experience can still be improved by a better integration of the conflict descriptions, which are in this example assumed as a flat list of textual renderings. In the tool *SuperMod*, conflict descriptions are attached to the context element in question as *conflict resolution markers*, which ease the identification of conflicting model elements. See Section 14.6.3.

Moreover, the reported conflict set depends on the choice. When deselecting, e.g., the feature Weighted, only the display conflict would be reported as both other conflicts are conditional to that feature. When being compared to family-based, *global* analysis strategies, this involves the danger of "overlooked" conflicts hidden in other variants, particularly those caused by unforeseen static feature interaction.

Flexibility. The amount of *flexibility* provided by the here considered conflict resolution approach is high. In contrast to many three-way merging tools, the definition of a fourth alternative value is allowed, e.g., the conflicting renaming of Couleured to Coloured and Kolored could be resolved by freely defining the alternative name Colored. In some cases, the added flexibility is paid, however, with a higher effort of conflict resolution. For example, the target of the association elements had to be manually redefined.

13.5 Related Work

Being a quite technical topic, workspace metadata management is not extensively covered by related work. Product well-formedness analysis, in contrast, has been the subject of a multitude of publications; an introduction was given in Section 5.5.2. Further below, techniques related to the here considered a-posteriori product-based well-formedness checking approach are outlined. They have their origins, among others, in line-oriented VCS, in three-way model merging, in MDSPLE, and in family-based SPL analysis.

Metadata Management in Model VCS. Every version control system needs to keep track of the exported workspace contents. Here, we are particularly interested in how other model VCS manage this task.

In *AMOR* [Alt+08], the models part of the workspace are virtually extended by *profiles* that contain two types of information: general versioning data and conflicts. Furthermore, a specialized editor (restricted to UML diagrams), which graphically displays version and conflict information, is provided.

EMFStore [KH10], an operation-based model VCS that follows a classical client/server architecture, organizes the multitude of metadata centrally on the master repository—there is no distinction between local repository and local workspace. The metadata stored for clients is confined to the data required to establish the network connection to the server. There, the entire version history, including three-way merge conflicts, is managed in terms of *change packages*.

Batch-Oriented Three-Way Merging of Text Files. The strategy underlying a-posteriori product-based analysis, namely exporting a preliminarily merged file into the workspace and having the user incorporate his/her conflict resolution decisions in a post-processing step, is similar to the behavior of many contemporary text-oriented version control systems when it comes to resolving three-way merge conflicts.

To mention only two representatives, in Git [Cha09] and in Subversion [CFP04], conflictingly inserted blocks of text are appended to the workspace text files in both versions and surrounded by corresponding merge tags that indicate the block's origin. The user

may resolve the conflict either by deleting one of the candidate blocks or by combining the blocks' contents manually. To this end, support is provided, among others, by graphical tools relying on *diff3* [KKP07] as back-end.

The difference to the here presented strategy is that all candidate versions of conflicting elements are presented in the workspace. In general, this is not feasible for intrinsically represented model instances since these need to comply to single-version metamodel rules. Therefore, one of the candidate versions is selected by the default resolution strategy as a "best guess", and the other candidates are only presented indirectly in the form of enhanced conflict descriptions (or conflict resolution markers in SuperMod).

Consistent Three-Way Merging of EMF Models. The state-based three-way EMF model merging approach described in [Wes14] supports the detection of an exhaustive catalog of context-free and context-sensitive merge conflicts, a subset of which have been also introduced in a generalized form in Section 13.3.2. In contrast to the approach presented here, where well-formedness violations can be due to conflicting modifications, but also to the combination of optional features, the description in [Wes14] explicitly assumes optimistic collaboration as reason for conflicts. On the one hand, this restricts the scope of conflict detection to three-way merging. On the other hand, both conflict detection and resolution can be defined in a more accurate way; for instance, a *delete-reference conflict* is caused by the deletion of an object in one alternative revision and a concurrently inserted new reference to the same object in the opposite revision. Using the approach presented here, such a situation would be recognized as a (less specific) *internal link target* conflict, and a dedicated opportunity to restore the deleted object would not become available during conflict resolution.

The three-way merge algorithm has been implemented by the tool *BTMerge* [SUW13b], which has technically influenced the tool *SuperMod* to be presented in Chapter 14. BTMerge presents the extensional multi-version domain model (here: merge model) in a three-column tree editor, and conflicts must be resolved *interactively* before the merge result is exported. Thus, the tool follows an *a-priori* approach, which is connected to the disadvantage of restricting the user to a specialized editor he/she may not be familiar with. Furthermore, by only allowing for predefined conflict resolution actions, the tool is less flexible than the strategy presented here.

Well-Formedness Repair by Default Strategies. The MDSPLE tool *FAMILE* [BS12b] employs a similar *product-based* strategy for the analysis and preliminary resolution of context-free – and, to a limited extent, context-sensitive – conflicts that may occur in specific product variants due to contradictory presence conditions attached to interdependent domain model elements. FAMILE uses an ordinary (intrinsic) model instance as multi-variant domain model, but allows to virtually extend the model by alternative model elements.

As described by [BS16a], two generic forms of conflict can be detected: *dependency conflicts*, which occur whenever an element included in the selected variant structurally depends (i.e., by a context-free cross-link or a domain-specific context-sensitive dependency) on an element not included there; and *mutex conflicts*, which appear in case several (regular or alternative) model elements compete for the value of a single-valued structural feature.

For the preliminary resolution, so called *propagation strategies* – default resolution instructions for dependency conflicts – and *selection strategies* – for mutex conflicts – can be defined globally for a product line. These strategies retrospectively change the selection state of affected elements, such that the conflict is (locally) resolved. If desired, the effect of conflict resolution can be reverted in a product preview between selecting the feature configuration and exporting the product variant. When regarding the classification introduced in Section 13.4.1, FAMILE relies on an *a-priori product-based* analysis strategy. Albeit, manual repair actions are made effective in the variant presented in the preview exclusively.

Family-Based Well-Formedness Analysis for Models. It was repeatedly mentioned that the generally preferred SPL analysis approach is *family-based* well-formedness checking as it allows to guarantee that each consistent feature configuration will produce a syntactically well-formed product variant. In Section 5.5.2, it has also been explained that family-based strategies are computationally complex, such that approximations like *sample-based* or in particular *feature-based* analysis come into question. The survey by Thüm et al. [Thü+14a] presents a multitude of approaches to integrate variability-aware analysis into model-driven software product line support.

The question whether a model conforms to a metamodel can be formulated as a type-checking problem. Correspondingly, variability-aware analysis of models may be performed using *family-based type checking*, which is realized, among others, by [Ap+10] in a compositional SPL approach.

[Hei09] describes the family-based analysis approach realized in the tool *FeatureMapper*. The performed generic multi-variant checks consider multiplicity and typing of model elements. Albeit, the constraint class of semantical validity is not further refined since it intrinsically depends on the modeling language in question. This property is shared with the approach presented in the text at hand.

Rules for the semantical correctness of models are often defined by means of OCL constraints; see Section 3.4.2. The variability-aware well-formedness checking approach by [CP06] evaluates OCL constraints in a multi-variant context, such that they are transparently ensured for all valid product variants. The product line to be validated is expected in the custom form of a a *feature-based model template*, which realizes *transformational variability*. Reported conflicts are, however, difficult to interpret by the modeler, who is forced to operate in a multi-variant context.

13.6 Summary and Conclusion

The description of the conceptual framework has been concluded by two related topics, *workspace metadata management* and *product well-formedness analysis*.

As workspace metadata, we understand all information that must be organized in addition to the user-visible workspace contents themselves in order to guarantee a consistent and accountable editing model workflow. Internally, metadata are represented as an extension to the model that represents the repository extrinsically. The information is differently shaped for master and for local copies of the repository, such that distributed versioning is

enabled. *Local metadata*, which is also relevant in single-user mode, comprise the active choice and optionally the reserved ambition, a *workspace descriptor* that is divided up into several *dimension descriptors*, which keep track of modification of different parts of the workspace, and a list of *product conflicts* that have been detected in the beginning of the current iteration.

For the detection and resolution of individual conflicts, suitable conditions have been formalized for the diverse product dimensions (the versioned file system, particularly EMF models, and the feature model). These conflict conditions are supposed to be evaluated based upon the internal, i.e., extrinsic product space representation that exists in the repository. As we argue that the designated users prefer to get notified about conflicts in their familiar intrinsic product representation that is used in the workspace, a compromise has been made with respect to the employed well-formedness management approach: *a-posteriori product-based analysis*. Using this technique, conflicts are detected in the filtered repository view that is internally created upon check-out; then, non-interactive *default resolution* is applied before the preliminary workspace is exported; and finally, the user may revise the automatically applied resolution decisions in the workspace with his/her preferred modeling tool, while being supported by meaningful conflict (resolution) descriptions.

The concepts explained in this chapter have made contributions towards the satisfaction of requirement **R12** (*product well-formedness control*; see page 21) and complemented the description of *collaborative SPLE* (**R18**) begun in the previous chapter.

All in all, the contributed approach to a-posteriori product-based well-formedness checking requires a moderate amount of user interaction, guarantees local correctness, and provides for a high level of flexibility for conflict resolution. When being evaluated in isolation, this technique is not necessarily the best choice for SPL analysis; however, it integrates well with the filtered editing model – which provides essentially product-based product line creation – underlying the conceptual framework. It must be kept in mind that product-based conflict resolution decisions are not confined to the product available in the workspace, but they also affect related members of the product family—a property that is unique when comparing the here presented strategy to other product-based approaches from the literature.

When compared to the four preceding chapters, this chapter was prone to many more assumptions about the technical realization of the framework. Not at least because of this, we now transition from the specification to the implementation – see Chapter 14 – and evaluation—see Chapter 15.

Part V

Proof of Concept

Source code is just a view
on the model of the system.

COLIN ATKINSON
(MODELSWARD2014 KEYNOTE)

Chapter 14

Implementation

Abstract

The presented conceptual framework for the integration of MDSE, VC, and SPLE has been fully implemented in *SuperMod*, a model-driven tool that is embedded into the integrated development environment Eclipse. This chapter first describes the user-visible functionalities offered by the tool, before implementation details are considered. SuperMod follows a client/server architecture; for the implementation of both components, the Eclipse Modeling Framework has been utilized. Relying on a dependency injection framework for the management of behavior, the application is highly modular. Moreover, a SAT solver supports the implementation of feature model constraint checking. The communication between client and server has been implemented as a REST-based web service. Product well-formedness analysis deeply integrates into Eclipse by enhancing the user-visible workspace contents with conflict resolution markers.

Contents

14.1 Overview

The purpose of this chapter is twofold. On the one hand, the description of the conceptual framework provided in the preceding Part IV left a couple of realization decisions open. They are resolved in this chapter, where the underlying technology is introduced. On the other hand, the evaluation provided in the next chapter utilizes the implementation described here in order to draw conclusions about the properties of the formal conceptual framework. This indirection is necessary because many observations require a practical application. The latter is here provided as the tool *SuperMod*[1], a complete Java-based model-driven implementation of the conceptual framework. Still, the tool is an academic prototype; restrictions that must be eliminated before making the tool applicable to industrial-scale projects are listed in the conclusion of this thesis.

The remainder is organized as follows: First, the tool's functionality is explained from the user's perspective in Section 14.2. Subsequently, supported repository architectures are distinguished. In Section 14.4, the utilized implementation techniques are presented. Section 14.5 deals with optimizing implementation details concerning both the client and the server side of the application. Remarks regarding the realization of the a-posteriori product-based well-formedness analysis strategy are provided in Section 14.6. An overview of related implementation topics found in the literature is given in Section 14.7. The chapter is concluded with tool availability remarks.

14.2 User Interface and Functionalities

This section[2] provides a functionally organized description of SuperMod. In the respective subsections, the provided user interface (UI) commands are described one after another. As

1 The acronym is for "*Super*imposition of *Mod*els", reflecting the central design decision **D2**.
2 The current Section 14.2 shares material with the tool demonstration paper [SW16b].

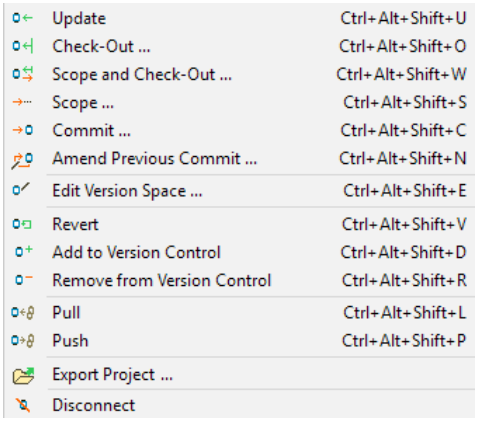


Figure 14.1: SuperMod's team context menu. From [SW16b, Figure 2].

shown in Figure 14.1, the commands have been integrated into Eclipse by means of a *team context menu* that is enabled for all projects managed by SuperMod.

14.2.1 Workspace Management

The workspace, represented by the contents of an Eclipse project, presents the currently selected product variant described by the choice. According to the extrinsic product model (see Chapter 10), the workspace may consist of arbitrary EMF model resources and of plain text files. Following a purely *state-based* approach, no tool extensions are required for resources to be supported by SuperMod.

A non-versioned Eclipse project may be put under version control using the UI command Share (not shown in Figure 14.1). As in ordinary version control systems, individual folders and files may then be added to or removed from version control—see UI commands Add to and Remove from Version Control in Figure 14.1. In case the user has not decided on whether a new file is to be versioned, a dialog, where the user may select relevant files and folders, is shown in the beginning of Commit.

For a better overview, the *active choice* is presented in a textual form in the Eclipse project explorer next to the project name. In this condensed representation, only positively selected leaf features are listed. Furthermore, in case an ambition is *reserved* (see below), its non-neutral bindings are also shown here. The displayed information originates from the metadata attributes activeChoice and reservedAmbition (cf. Figure 13.1 on page 272.)

Figure 14.2 shows an example where the choice includes a positive selection of the optional features weighted and undirected, and the ambition is defined by a positive selection

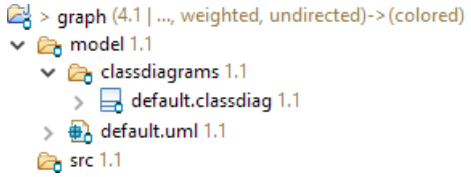


Figure 14.2: The Eclipse project explorer with an active SuperMod project. From [SW16b, Figure 3].

of a newly introduced feature colored.

Alongside of the workspace files and folders, the *visibilities* of the corresponding versioned elements in the repository are displayed in a human-readable form. Within file contents, however, no visibilities are displayed due to technical restrictions resulting from requirement **R10** (*reuse of existing tools*).

Finally, the UI command Disconnect removes all local version control metadata and retains only the current variant of the project in the local workspace.

14.2.2 Feature Model Editor

Conceptually, the feature model is part of the workspace as it incorporates an additional product dimension.

For its editing, a dedicated UI command Edit Version Space is provided (see Figure 14.1). Once selected, the feature model editor (a customized EMF tree editor; cf. screenshot in Figure 14.3) is opened, presenting the chosen revision of the feature model. Here, features can be added or deleted, toggled mandatory (filled circle) or optional (empty circle), or renamed. Furthermore, the tree structure can be modified arbitrarily, and *feature groups* and *requires/excludes constraints* can be specified.

In the background, the feature model is kept consistent with its mapping to the low-level rule base; see Section 9.4.2. In addition, dedicated EMF Validation [Ste+09], in connection with satisfiability checking (see Section 14.4.3), ensures that the feature model remains satisfiable according to Constraint 3 (see Section 11.2.2 on page 225).

14.2.3 Version Selection

Every editing model iteration is embraced by the commands CHECKOUT and COMMIT formally defined in Section 11.3. As shown in Figure 14.1, UI commands are offered for both actions. Furthermore, the operation MIGRATE is transparently invoked after each commit. This subsection considers the "straightforward" dynamic workflow, where the choice is specified during check-out and the ambition during commit. The user may deviate from this by using one of the specialized commands presented subsequently in Section 14.2.5.

Check-Out. SuperMod follows the two-level version selection strategy provided by the conceptual framework. Having executed the UI command Check-Out, a *revision* must be

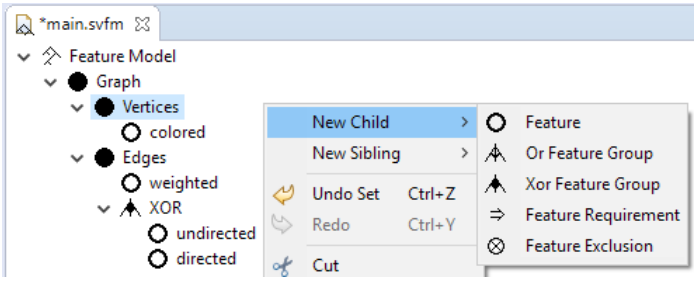


Figure 14.3: The feature model editor. From [SW16b, Figure 4].

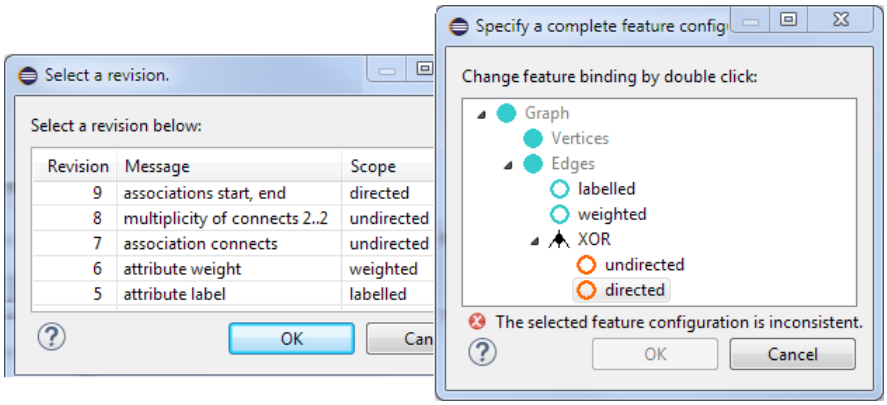


Figure 14.4: Dialogs for revision and feature configuration selection shown during check-out. Based on [SW16b, Figure 5].

selected first (cf. left hand part of Figure 14.4; column Scope displays the ambition having been used for the respective commit). Then, the selected revision of the feature model is presented to the user, who must then specify a *feature configuration* binding all visible features either to *selected* (cyan) or to *deselected* (orange symbols in the right hand part of Figure 14.4). The configuration is created in a top-down way, where mandatory children are automatically bound positively.

The feature configuration specification dialog also ensures that the selected configuration is *consistent* according to Constraints 1 and 2; see page 224. The inconsistency reported in Figure 14.4, for instance, is caused by none of the features undirected and directed located in the same XOR group being selected.

Update. Technically, the UI command Update specializes Check-Out in a non-interactive way. In the revision graph, the latest revision is selected; in the feature model, the current choice is retained. If the selected revision of the feature model contains a feature that is not bound currently, the user is requested to complete the configuration in an exceptional dialog. The effects equal those of the operation Check-Out: The workspace contents are replaced according to the choice, and product well-formedness conflicts are reported if present.

Both check-out and update are only applicable as long as the workspace contents have not been modified; otherwise, an explicit Revert (see below) is necessary in advance. Also notice that an explicit Pull (see below) is required in order to synchronize with remote modifications; update does not enforce pull.

Commit. The UI command Commit terminates an iteration and prompts the user for an *ambition* that delineates the set of versions for which the change, representatively applied in the version described by the *choice*, is intended. A new revision is introduced transparently; the author of the commit is merely prompted for a commit message (see left hand part of Figure 14.5).

Thereafter, the feature ambition specification dialog appears (right hand part). In addition to *selected* and *deselected*, the value *neutral* (yellow) is permitted here, signaling that the change is immaterial to the respective feature. Following Constraints 4 and 5 (cf. page 225),

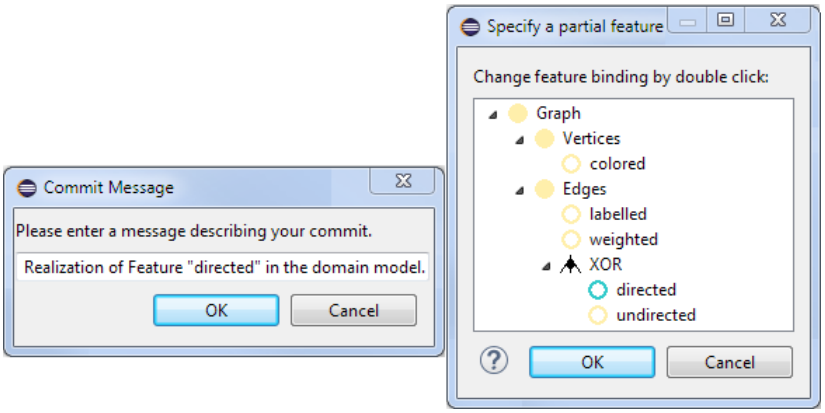


Figure 14.5: Dialogs for commit message and feature ambition selection shown during commit. From [SW16b, Figure 6].

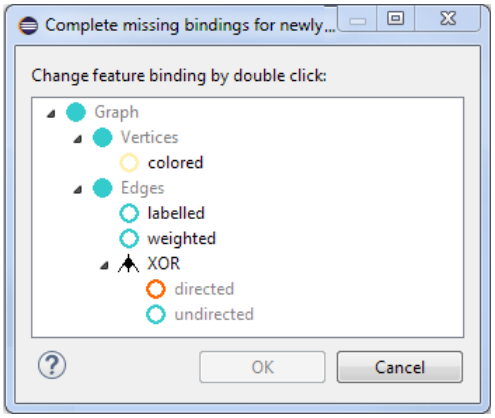


Figure 14.6: Dialog supporting choice migration.

the dialog ensures that the ambition is *weakly consistent* and *represented by* the previously defined choice.

Furthermore, it is checked whether the selected feature ambition is *sufficiently specific* to the performed product-level modification (cf. Constraint 6). If this is not the case, the user has two options: Either, the current iteration is reverted (see below), or the feature configuration is refined by additional non-neutral bindings in order to satisfy the constraint [3]. The user is assisted by a pre-calculated sufficiently specific ambition; see Section 14.5.2.

Migrate. In the dynamic filtered editing model realized by SuperMod, repeated check-outs become optional—it is assumed that the user wants to stay in the current variant after having committed. For this sake, the choice is *migrated* into the next iteration; see Section 11.3.4.

The non-deterministic selections included in Algorithm 11.6 are realized by an additional choice migration dialog, an example of which is depicted in Figure 14.6. Without

3 This constraint can be globally disabled in SuperMod in order to follow a less restrictive yet more error-prone workflow.

predefining a concrete selection order, those features whose binding state cannot be inferred automatically from the ambition or from preferences or from defaults are made available for selection or deselection. The dialog actively enforces Constraints 7, 8, and 9, and always offers the possibility to *cancel*, which triggers a new check-out immediately.

14.2.4 Additional Workspace Operations

Revert. This operation cancels the current iteration, undoing all modifications made after the last check-out or migration. The original workspace contents are reproduced by applying the active choice to the product space and by exporting the results.

Export Project. To finally deploy products to customers, the development loop may be left by the UI command Export Project. A revision and a feature configuration must be selected by the user; the product represented by the specified version is exported into a new Eclipse project clear of version control metadata. [4]

14.2.5 Generalized Editing Model

As explained in Section 11.6, the editing model part of the conceptual framework can be generalized such that it weakens the assumption that the choice is always specified during check-out and the ambition during commit. In SuperMod, parts of the generalized editing model have been realized.

A Lightweight Form of Static Filtered Editing. First of all, the tool may be used in a *lightweight static* way, supported by the UI command Scope and Check-Out. Here, both the ambition and the choice are specified already during check-out in order to fix the scope of the change as early as possible. After selecting a revision, the ambition selection dialog is shown first; thereafter, the user selects a representative choice—the validation of according consistency constraints is preponed.

In contrast to fully static filtered editing, feature model editing is, however, not inhibited; the validation of Constraint 5 is repeated during commit if necessary, and the *sufficiently specific ambition* check (Constraint 6) is applied as usual. In case it fails, the user may correct the ambition dynamically. Also, the MIGRATE operation is applied at the end of a static iteration.

Reserving Ambitions. Second, when selecting only a choice during check-out (or when omitting the check-out), it is still possible to *reserve* an ambition at arbitrary points in time during MODIFY. Upon having selected the UI operation Scope, the ambition selection dialog appears. Reserved ambitions – also those enabled by Scope and Check-Out – are displayed in the project explorer (cf. colored in Figure 14.2). During commit, the ambition selection dialog is omitted in case the ambition still satisfies all necessary constraints.

4 The inverse operation, Import Project, has been neither conceptually investigated nor implemented to date.

Amending Ambitions. Last, following the theoretical explanations provided in Section 11.6.4, an erroneously specified ambition can be retrospectively altered by using the UI command Amend Previous Commit. When selected, the ambition specification dialog shows up, where the ambition used for the preceding iteration may be redefined.

14.2.6 Collaboration

Being a fully-fledged version control system, SuperMod natively supports collaborative development. This has been realized using the *singleton master* replication strategy explained in Chapter 12. Each user interacts with a local repository using the commands Check-Out and Commit (see above). In multi-user mode, the revision selection dialog shown upon Check-Out presents additional details including author, commit date, the logical scope (i.e., the feature ambition defined for the respective commit), and the two-level revision naming scheme (see Figure 14.7).

In order to orchestrate multiple copies of the repository, the UI commands Pull and Push are provided, which communicate with a central remote repository according to the multi-user editing model semi-formally defined in Section 12.5.

Pull. In order to avoid conflicts between pending local changes and incoming remote modifications, the command Pull is enabled only if the local workspace is in an unmodified state. Furthermore, a network connection to the server must have been established.

After pull, the repository contents are up to date with the server-side master repository. In order to make this visible in the workspace, an Update is recommended to the user.

Push. This operation makes the same prerequisites: no unfinished private transactions and an active network connection to the server.

Before pushing, the central master repository is scanned for incoming changes. If present, an *out of date* situation is signaled to the user and a Pull is enforced in advance. In this case, however, the update recommended by the pull operation is delayed.

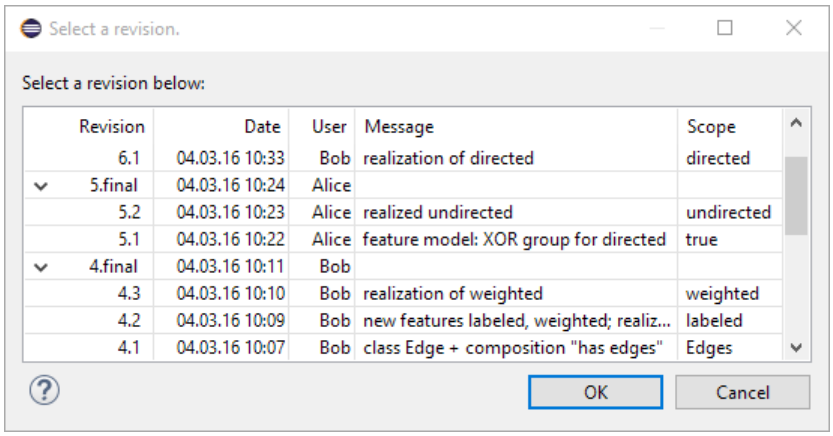


Figure 14.7: Revision selection dialog in collaborative versioning mode. From [SW16b, Figure 8].

In order to describe the intent behind the series of commits underlying the push, the user may phrase a *push message* in an additional dialog (not depicted here).

Then, the push is executed and all bookkeeping steps are applied as described in Section 12.5.2. Also here, non-interactive element raw merging and three-way visibility merging are applied (cf. Section 12.4).

Now, in the *out of date* case, an *update* is applied in order to transfer those remote changes that affect the selected product variant to the workspace. On this occasion, *product well-formedness violations* may arise due to conflicting remote and local changes. These are default-resolved and then reported in the local workspace according to the a-posteriori product-based analysis strategy; see Section 14.6.

14.3 Supported Repository Architectures

In most of the explanations and examples provided in previous chapters, we implicitly assumed that the contents of the repository conform to the hybrid architecture sketched in, e.g., Figure 9.4 on page 164. Furthermore, there was an explicit distinction between single-user mode and multi-user (collaborative) versioning, which has been introduced in Chapter 12 as a conceptual extension to the revision graph metamodel and its mapping.

In advance to the realization of the conceptual framework, it has turned out that differently shaped repositories, which reuse the same version dimensions, provide potentially useful applications, too. For instance, historical variability may already be covered by an external VCS, or logical variability may not be required at all, in specific projects.

In addition to the product and version dimensions explained in Part IV – revision graph, feature model, versioned file system, and collaborative revision graph – a *low-level logical dimension* has been implemented as an additional version dimension mainly for experimental purposes. This dimension straightly maps the rule base defined in Section 9.2.1 to a textual

```
1  logical dimension Graph {
2     option Graph
3     invariant graphMandatory { Graph }
4     default graphTrue for Graph { true }
5     option labeled
6     option directed
7     option undirected
8     invariant direction { directed xor undirected }
9     preference undirectedPref for undirected { not directed }
10    preference directedPref for directed { not undirected }
11    option transpose
12    invariant transposeDirected { transpose implies directed }
13 }
```

Listing 14.1: Example of a low-level textual version space definition. Options, invariants, preferences, and defaults have unique identifiers. The contents of lines 2, 4, 8, and 9 are mapped to the option o_{Graph}, the default $(o_{Graph}, true)$, the invariant $\neg(directed \wedge undirected)$, and the preference $(o_{directed}, \neg o_{undirected})$, respectively.

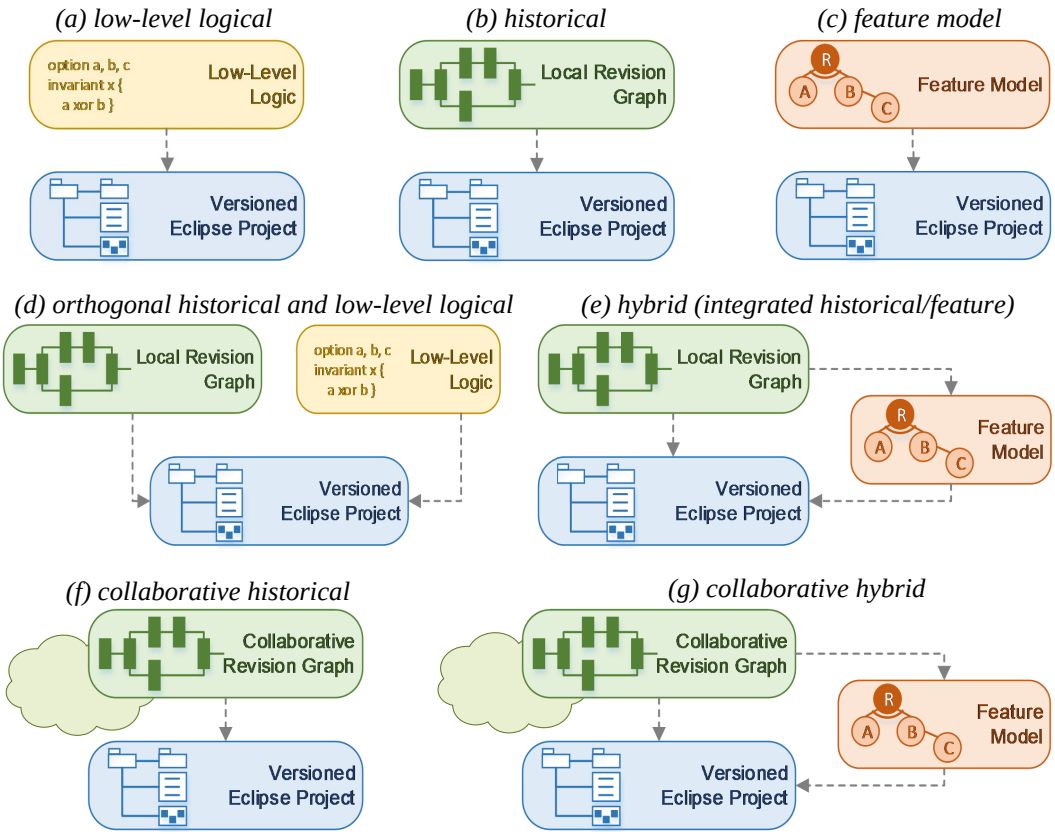


Figure 14.8: Seven distinct repository architectures available in SuperMod.

syntax; the corresponding file is made available for editing upon Edit Version Space. A textual definition referring to the Graph example is provided in Listing 14.1.

Technically, the co-existence of different configurations is realized with the help of *dependency injection*; details are provided in Section 14.4.2. The decision which architecture to use for a specific project versioned by SuperMod is made by the user during the Share action. Rather than being allowed for arbitrary composition of the five dimensions, the user may choose among the following pre-defined repository architectures (see Figure 14.8):

(a) Low-Level Logical Versioning. No historical versioning is applied. The version space is defined by low-level propositional logic in textual syntax. The level of abstraction is relatively low. Moreover, the logical rule base is not versioned at all.

(b) Historical Versioning. Using this configuration, the primary product space is exclusively historically versioned by a (single-user) revision graph. There is no possibility to define logical variability in addition.

(c) Versioning by a Feature Model. When compared to (a), this architecture provides a higher-level abstraction for purely logical versioning. Apart from this, the same restrictions with respect to historical versioning hold.

(d) Orthogonal Historical and Low-Level Logical Versioning. Historical and logical ver-

sioning are applied in an orthogonal way, i.e., a revision graph and a low-level logical rule base are available, but they are not connected in any way. In particular, this may lead to lost updates, e.g., when a logical option is deleted. Behind the scenes, an invisible *change space* (cf. Section 9.6) is used to abstract from both the revision graph and the logical rule base.

(e) Hybrid Versioning. The three-layered hybrid architecture introduced in Chapter 9. The feature model plays a dual role, being both a product and a version dimension. The *change space* is used as optimization in the background.

(f) Collaborative Historical Versioning. Like (b), but uses a collaborative revision graph and the distributed replication strategy presented in Chapter 12.

(g) Collaborative Hybrid Versioning. The most powerful but potentially the most complex alternative. Collaborative versioning is applied in an integrated architecture; see (e). This enables fully collaborative (MD)SPLE.

All of the architectures assume that the main product dimension to be versioned is an Eclipse project represented as a versioned file system extrinsically. We have not found any meaningful advantage of omitting this dimension except for collaborative feature model versioning. This, however, is enabled without any further restrictions by (g). Furthermore, the *visibility forest* optimization introduced in Section 9.7 is applied in all cases.

14.4 Internal Architecture and Implementation Technologies

The package diagram depicted in Figure 14.9 offers a coarse-grained view on the internal architecture of the tool. It consists of four tiers (displayed top-down), which are further refined by the available version and product dimensions. The metamodel tier defines the contents of the repository; see Section 14.4.1. Package supermod::services contains interface declarations and corresponding implementations for low-level operations such as FILTER, EXPORT, or MATCH. On the third tier, represented by supermod::commands, user actions such as CHECKOUT or PUSH (see Section 14.2) are provided; their implementation depends on the selected repository architecture (see previous section), all of which are mapped to individual sub-packages. Last, the client and server applications provide these commands to the user by a graphical interface or by web services, respectively.

For modularity, each of the overall 41 packages has been realized as an individual Eclipse project deployable on its own. For each of the repository architectures listed in the preceding chapter, a corresponding Eclipse plug-in packages the necessary deployables—see installation remarks in Section 14.9.

The remainder of this section explains how existing technology has been used for the implementation of particular packages resembling components of the tool.

14.4.1 Eclipse Modeling Framework

It was mentioned that SuperMod has been implemented in a model-driven way, relying on the Eclipse Modeling Framework (EMF). More precisely, source code for the structure of the repository contents (see package supermod::metamodels), enabling the extrinsic

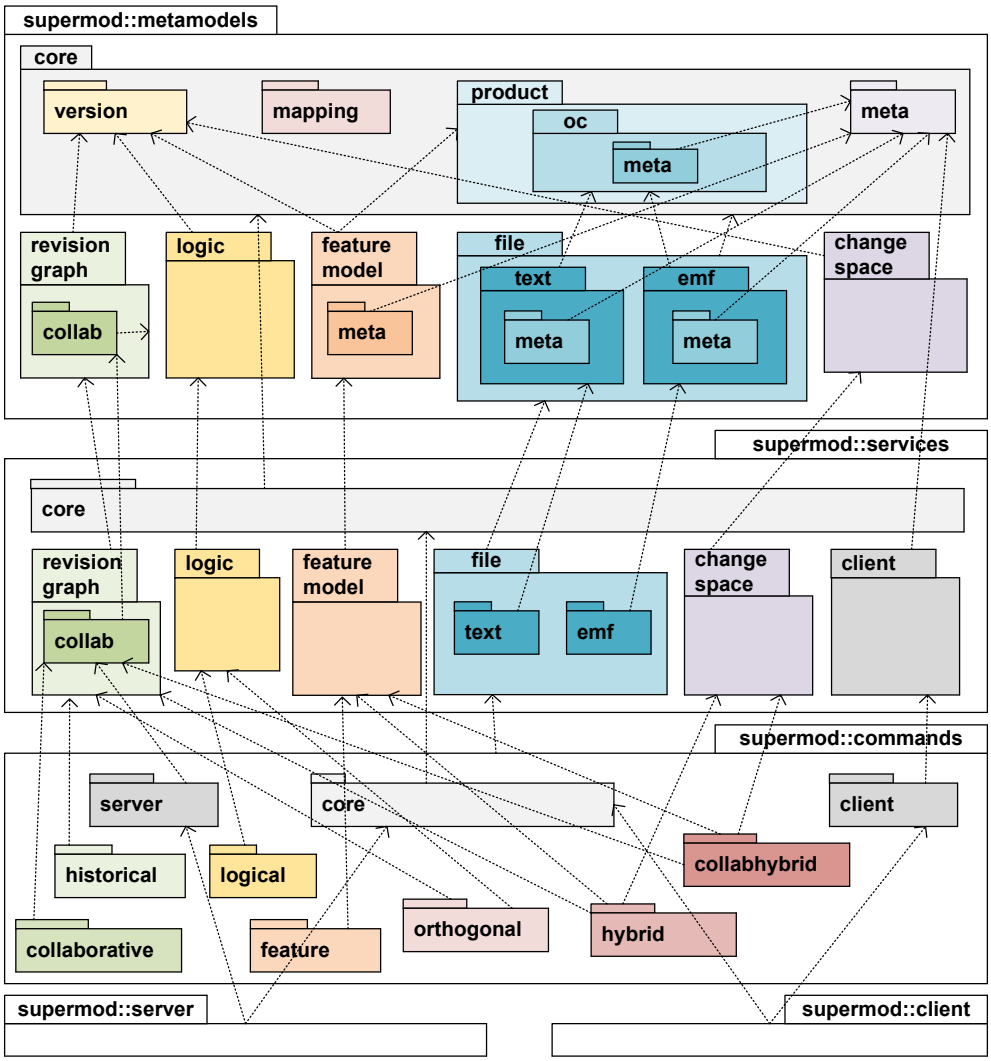


Figure 14.9: Package diagram describing SuperMod's repository structure. All dashed errors denote public imports (i.e., they are transitive and available in all inner packages).

representation, has been obtained through EMF code generation (see Section 3.5), taking as input extended versions of the metamodels presented in Chapters 9, 10, 12, and 13, as well as an additional (trivial) metamodel for the low-level version dimension logic.

The initial version of the feature model editor was generated by EMF, too. Since the workspace version is represented in an extrinsic multi-version format (cf. Section 13.2.4), the *Edit* and *Editor* code had to be significantly customized in order to simulate a single-version feature model to the end user. The customized editor ensures that all feature model well-formedness conditions listed in Section 13.3.3 remain satisfied at all times. The customized feature model editor also ensures the dynamic consistency constraints introduced in Section 11.3.2. For instance, in the case of deletion, an instance of Deleted is created under the respective feature; cf. Section 10.7.1 and Algorithm 11.3 on page 229.

14.4.2 Dependency Injection with Guice

The orchestration of the structural and behavioral components part of the tiers supermod::services and supermod::commands is achieved with the help of the *dependency injection* framework *Guice* [Van08].

Following the design principle of *inversion of control*, the Guice framework interprets Java classes as *services*, whose design contract is declared by an interface. The decision which concrete implementation to use for a service is made at run-time and can be controlled by *modules*, which bind specific interfaces to their preferred implementation. Services may depend on other, existing services, which are referenced by their interface. This way, service implementations can be exchanged, specialized, or extended on demand, without the need of changing the client code.

A concrete example of how this mechanism is exploited in SuperMod is provided in Listing 14.2. It presents, in simplified form, the implementation of the command CHECKOUT specific to the hybrid repository architecture (cf. Figure 14.8(e)). The corresponding service interface `ICheckout` is defined in the core package of the supermod::commands tier; the binding to `HybridCheckout` is defined externally in a Guice module part of package supermod::commands::hybrid.

```
1  package supermod.commands.hybrid;
2
3  import com.google.inject.Inject;
4  import supermod.metamodels.core.OptionBinding;
5  import supermod.metamodels.core.Repository;
6  import supermod.metamodels.revisiongraph.RevisionGraph;
7  import supermod.metamodels.featuremodel.FeatureModel;
8  import supermod.metamodels.file.VersionedFileSystem;
9  import supermod.services.*;
10
11 public class HybridCheckout implements supermod.commands.core.ICheckout {
12
13    @Inject @supermod.services.revisiongraph.Revisiongraph
14    private supermod.services.IChoiceDefinitionService rgChoiceServce;
15
16    @Inject @supermod.services.featuremodel.Feature
17    private supermod.services.IChoiceDefinitionService fmChoiceServce;
18
19    @Inject
20    private supermod.services.IFilterService coreFilterService;
21
22    @Inject @supermod.services.featuremodel.Feature
23    private supermod.services.IExportService fmExportService;
24
25    @Inject @supermod.services.file.File
26    private supermod.services.IExportService fileExportService;
27
28    @Override public void checkout(Repository repo) {
29       RevisionGraph rg = repo.getVersionDimensions().get(0);
30       FeatureModel fm = repo.getVersionDimensions().get(1);
```

```
31        VersionedFileSystem vf = repo.getProductDimensions().get(1);
32        OptionBinding rgChoice = rgChoiceService.choose(rg);
33        FeatureModel filteredFm = coreFilterService.filter(fm, rgChoice);
34        OptionBinding fmChoice = fmChoiceService.choose(filteredFm);
35        OptionBinding choice = rgChoice.union(fmChoice);
36        VersionedFileSystem filteredVf = coreFilterService.Filter(vf, choice);
37        fmExportService.export(filteredFm);
38        fileExportService.export(filteredVf);
39    }
40 }
```

Listing 14.2: Use of Guice dependency injection in the hybrid check-out command.

The interface defines a method `checkout`. As explained in Section 11.3.1, CHECKOUT is essentially a combination of FILTER (by a user-defined choice) and EXPORT. Due to the hybrid role of the feature model, a fixed selection order must be maintained.

Class `HybridCheckout` makes use of elsewhere defined services, whose concrete implementation is injected at run-time, such that instances of the corresponding classes are assigned to the object variables annotated with `Inject`. In the case of choice definition and exporting, dimension-specific services are invoked—the corresponding dimensions are defined by a constraining annotation, e.g., `@Feature`. In addition, the operation FILTER has been qualified in a generic way by the interface `IFilterService`, such that it can be applied to both the feature model and the versioned file system.

The fact that this service class implements `ICheckoutService` for the specific *hybrid* architecture is expressed by a *module binding* externally; see Listing 14.3.

```
1  package supermod.commands.hybrid;
2
3     public class HybridModule extends com.google.inject.AbstractModule {
4
5     @Override public void configure() {
6        bind(ICheckoutService.class).annotatedWith(Hybrid)
7                 .to(HybridCheckoutService.class);
8     }
9  }
```

Listing 14.3: Guice module binding for hybrid check-out command.

14.4.3 Satisfiability Checks with Sat4j

SuperMod fully implements the consistency-preserving dynamic filtered editing model presented in Chapter 11. Constraints 3 and 4 require to check the (constrained) satisfiability of the invariant set $\mathcal{J}$ part of the low-level rule base. Furthermore, in Section 8.3.2, it was mentioned that satisfiability checks can be approximated using three-valued logic.

This approximation has been implemented in SuperMod, too. If not applicable, the satisfiability check is performed with the help of the SAT solver *Sat4j* [BP10]. Internally, Sat4j maps satisfiability checks to systems of numerical equations and provides optimized strategies for solving those. As a programming interface, a so called *gate translator* is

offered by the library. With its help, the system of equations can be deduced from logical clauses k_i of the following form:

$$k_i \ : \ v_i \Leftrightarrow v_{i_1} \ op_i \ \ldots \ op_i \ v_{i_n} \tag{14.1}$$

Where v_i denotes either a positive or negated reference to a free variable or a boolean constant ($true$ or $false$). Moreover, v_{i_1} until v_{i_n} correspond to positive or negative occurrences of free variables different from v_i and others in the same clause, and op_i denotes one of the logical operators $\wedge$, $\vee$, $\otimes$, or $\Leftrightarrow$.

Invariants, in SuperMod internally represented as instances of `OptionExpr` (see Section 9.2.1), may be, however, arbitrarily complex shaped. Therefore, before applying a satisfiability check to them, they need to be translated into Sat4j-compliant logical clauses in a pre-processing step. The corresponding implementation located in the package supermod::services::core proceeds as follows: Complex expressions are reduced by introducing substitute variables, and by introducing new clauses in which the contained expressions are refined, using the substitute variable as left hand side. Operators not supported by Sat4j, in particular, $\Rightarrow$, are rearranged correspondingly. The substitution step is repeated recursively until all remaining gate clauses are valid with regard to the gate translator.

Taking into consideration feature models as a metaphor for invariants of the variant dimension, the satisfiability check involves a total of four levels of abstraction (see example depicted in Figure 14.10): feature models, propositional logical invariants, Sat4j-compliant clauses, and finally, systems of numerical equations. The last level is hidden from SuperMod's implementation and has therefore been omitted in the example.

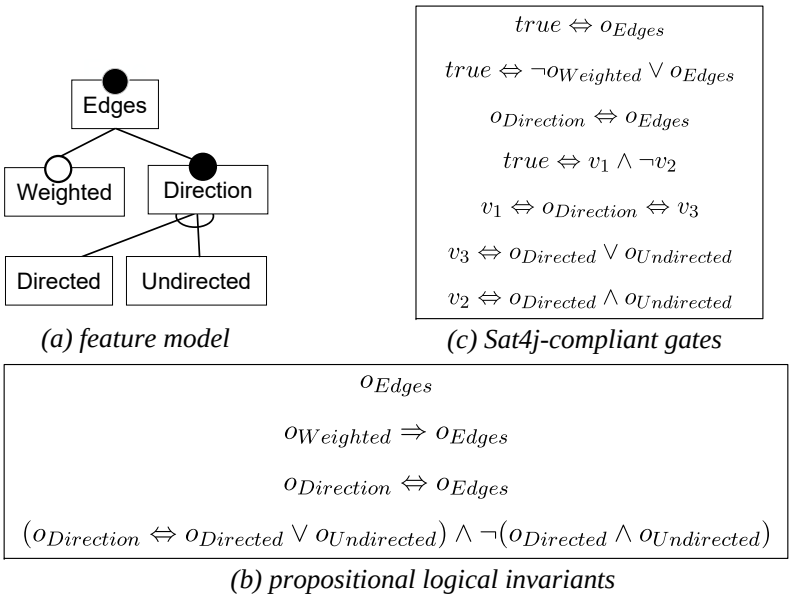


Figure 14.10: Connection between feature models, invariants, and Sat4j clauses.

14.4.4 Eclipse Team Provider

The client side of the application has been implemented as an Eclipse *team provider*[5] extension. For modularity, all dependencies to the programming interface of Eclipse – except for dependencies inherited from EMF – have been encapsulated in the package supermod::client (see Figure 14.9).

The team provider interface is defined by an *extension point* (see Section 5.4.3, *component frameworks*), whose mandatory operations – the Share command and the team menu – have been implemented by delegation to corresponding classes in the supermod::commands tier.

In Figure 14.2, another capability provided by the team provider extension has been presented: decorating versioned contents with a label. For SuperMod's implementation, this has been exploited for presenting the workspace choice and ambition (if present) at the workspace root, as well as the visibilities of versioned files and folders.

14.4.5 Representational State Transfer

The server component[6] of SuperMod has been realized as a *REST*-based (Representational State Transfer) web service designed to be hosted on an Apache Tomcat 7 servlet container[7] locally or on an external machine. The remote repository instance utilizes the server-side metadata structure defined in Section 13.2.1. For the implementation of the server-side REST interface (cf. package supermod::server in Figure 14.9), the framework *Jax-RS* [Bur09] has been utilized.

The architectural style REST is based on the *Hypertext Transfer Protocol* (HTTP). Remotely available data is universally addressed by means of *resources* encoded in hierarchical *URL*s. The content transferred with and obtained from a HTTP request – the *entity* – may have binary or text format. Moreover, different *methods* are distinguished: *GET* serves for reading, *POST* for creation, and *PUT* for modification of resources. Requests may return a result entity and a return code that distinguishes successfully processed requests from different types of failure situations, each being mapped to an individual error code.

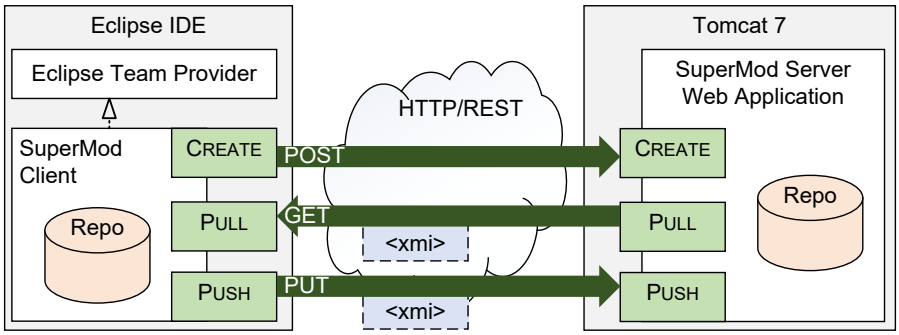


Figure 14.11: Implementation of client/server communication. Based on [SW16a, Figure 6].

5 https://projects.eclipse.org/free-tags/team-provider
6 This subsection is partly based on [SW16a, Section 5].
7 http://tomcat.apache.org/

Below, the mapping of the distributed VCS commands CREATE, PULL, and PUSH, to HTTP methods is explained as illustrated in Figure 14.11, assuming that the servlet is running at `http://root.url/supermod/`.

Create. The initialization of a new remote repository is mapped to a POST request of the form `http://root.url/supermod/repoPath/create?user=X`
where the entity contains the XMI serialization of the entire initial repository. Variable `repoPath` distinguishes several independent repositories. Parameter `user` is for authentication[8] and logging purposes. The initial write transaction number 1 is returned as result entity.

Pull. Server-side changes are requested using
GET `http://root.url/supermod/repoPath/pull?user=X&readTNo=Y`
where the query entity is kept empty. An XMI-serialized symmetric delta – see Section 12.3.2 – is returned that includes all changes referring to transactions closed after `Y`, which denotes the requesting client's latest read transaction number. (In the case of `Y`$= -1$, the entire repository contents are returned. This is exploited by the operation CLONE.)

Push. Transferring client-side changes to the remote repository is provided as PUT method:
`http://root.url/supermod/repoPath/push?user=X&readTNo=Y&writeTNo=Z`
where `writeTNo` denotes the number of the write transaction to be closed on the client side. In case `Y` does not match the most recently closed server-side transaction, the repository is *out of date*. Then, an error response code is returned that signals to the client that a PULL must be performed first. Otherwise, the local repository is merged with the XMI-serialized symmetric delta transferred in the entity—see Section 12.4. A new write transaction is started, whose number is returned in the result entity.

These methods are invoked by the SuperMod client (cf. package supermod::client in Figure 14.9) when the user selects the corresponding command. This has been implemented with the help of the network communication classes provided by the Java standard library.

14.5 Detailed Implementation Remarks

After having sketched the coarse architecture of SuperMod and having presented existing technologies used for its realization, we revisit particular pieces of implementation at a higher level of detail. Special emphasis is put on optimization strategies, which require to make assumptions about the underlying technical framework and were thus not introduced as part of the theoretical framework in Part IV.

14.5.1 Physical Organization of Local Repository and Metadata

The contents of the local repository and the metadata sections were conceptually presented in Part IV in terms of instances of different metamodels. Here, we supplement information how

8 To date, secure authentication is not supported by SuperMod.

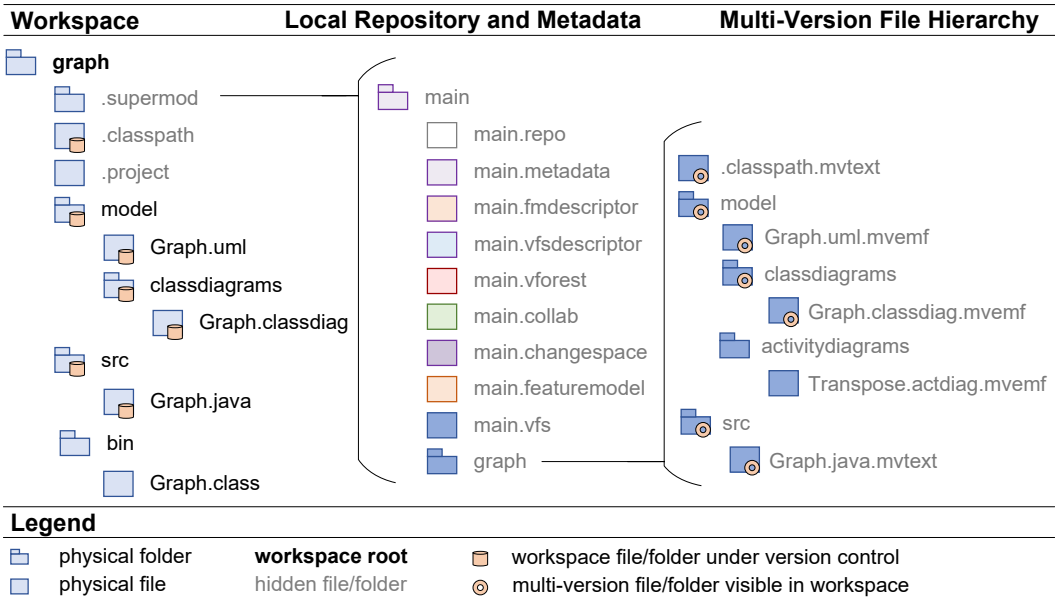


Figure 14.12: Mapping of workspace, repository, and version control metadata to the physical file system of the client.

these model instances are physically distributed over multiple files and folders. Figure 14.12 illustrates the explanations.

A SuperMod project is distinguished from a standard Eclipse project by a hidden folder `.supermod` below the workspace root. This folder contains both the local repository and the metadata section. The entry point, file `main/main.repo`, contains an instance of the core metamodel presented in Figure 9.5. In contrast to the conceptual description, however, the containment references visibilityForest, metadata, and both instances of dimensions have been replaced by cross-resource references, such that the corresponding model instances are contained in individual files (with suffixes `.metadata`, `.vforest`, `.featuremodel`, etc.). Also the different *dimension descriptors* (see Figure 13.1) are organized in this way (cf. `.fmdescriptor` and `.vfsdescriptor`).

The workspace version of the feature model is contained in the feature model descriptor as described in Section 13.2.4. For the reason of scalability, the product dimension *versioned file hierarchy* is decomposed into individual resources. The file hierarchy presented in the workspace is reflected, and the files part of the multi-version file hierarchy are represented as instances of subclasses of VersionedFile (see Figure 10.5) in individual physical files, whose extension depends on the multi-version file type (e.g. `.mvemf` for EMF files; see Figure 10.8).

To keep track of modifications made to workspace artifacts, file hashes are utilized as suggested in Section 13.2.3. In SuperMod, these are computed by the *Secure Hash Algorithm* SHA-1 [EJ01].

14.5.2 Optimizing Local Repository Operations

The description of the underlying conceptual framework already includes several optimizing strategies, which have been implemented in SuperMod accordingly. For instance, the

change space (see Section 9.6) and the visibility forest (see Section 9.7) avoid duplication of visibility expressions. Moreover, the constrained satisfiability approximation presented in Section 8.3.2 may accelerate the validation of consistency constraints presented in Chapter 11.

We here add implementation-level optimizations that have been realized in the client.

Visibility Evaluation Cache. The visibility forest is a global data structure for visibilities designed to reduce memory consumption. In addition, run-time can also be saved by *caching* evaluation results. To this end, in SuperMod, every node of the visibility forest, represented by instances of subclasses of OptionExpr obtains two additional attributes when compared to Figure 9.7:

– a cachedValue of type Tristate, and
– a boolean cacheValid with default value false.

In advance to each evaluation of the visibility v_i of an element e_i (see operation FILTER defined by (**10.6**)), it is checked whether cacheValid is true; if so, the cachedValue is returned for $v_i(c)$; otherwise, the visibility is evaluated normally, the result to be returned is saved as cachedValue, and cacheValid is set to true.

The validity of the cached value depends on the current workspace choice c. In the dynamic filtered editing model implemented, a Commit operation (followed by a transparent Migrate; cf. Section 11.3) will not change[9] any existing bindings in c, and therefore not invalidate the cached visibilities. Visibility updates are always represented by new (and non-cached) visibility nodes. Therefore, there are only two situations remaining in which cacheValid needs to be set to false:

– After an explicit CHECKOUT (or UPDATE), where a new choice is defined, whose binding may disagree with the cached visibilities.
– After a PULL in case three-way merging was involved.

Values of both attributes are never transferred along with push/pull operations since they are valid within one workspace (carrying an individual choice) only.

Revision Choice Completion. When confining to the revision graph dimension, the operation COMPLETE (see Algorithm 9.3 on page 169) behaves inefficiently. After the user has selected a revision n, it takes $n - 1$ iterations until relevant preferences of the form (r_i, r_{i+1}) have been applied, and one additional iteration for defaults $(r_i, false)$. In each iteration, all preference expressions are evaluated, exposing quadratic complexity in total.

Optimized revision choice completion exploits that the graph structure is encoded in the pair of references predecessor/successor. Redefining Algorithm 9.3, the strategy proceeds as follows based on the selected revision j (thus, $c_r = \{(r_j, true)\}$):

1. A depth-first search is started from revision j, following the successor reference[10]. Options of visited revisions are stored in a set O_{succ}.

9 For newly introduced options, new bindings are added. These do, however, not occur in cached visibilities.
10 This assumes that more recent revisions are accessed more likely than revisions located at the beginning of the revision graph.

2. For all $r_i \in O_{succ}$, a binding $(r_i, false)$ is added to c_r.
3. For all $r_k \in (O_r \setminus O_{succ})$, $(r_k, true)$ is added to c_r.

Altogether, linear run-time is achieved.

An analogous optimization exists for collaborative revision graphs, whose mapping has been defined by Table 12.1 on page 255. The fact that a revision may have multiple successors does not negatively affect the performance.

Exceptions to the Sufficiently Specific Ambition Check. Practical application of SuperMod has shown that requiring a *sufficiently specific ambition* as defined by Constraint 6 unnecessarily hampers the workflow in certain situations. To overcome this, SuperMod offers an extension point for the definition of *exceptions*, which exclude specific types of elements from this check.

Currently, two exceptions have been implemented—the list can be extended in future:

– After changing the order of elements in the workspace projection of a *versioned collection*, it may happen that specific edges transparently created or deleted may refer to hidden vertices, whose visibility is not included in the full ambition. Therefore, elements of *versioned collections*, i.e., instances of OCVertex and OCEdge are excepted under the assumption that non-representative changes to the mutual order of changes cannot cause errors more severe than order conflicts.

– When modifying the graphical representation of models in the workspace, layout changes are seldom meant to be connected to the ambition. In order to avoid pseudo inconsistencies of this type, an exception for EMF objects whose classes are defined in the GMF (*graphical modeling framework*, see Section 3.6.3) notation package has been introduced.

Recommendation of a Sufficiently Specific Feature Ambition. As another user-visible optimization based on Constraint 6, in case a correcting ambition is required after the constraint was violated, the feature ambition specification dialog (cf. Figure 14.5) is initialized with a *recommended ambition* that satisfies the constraint.

To this end, the following procedure has been implemented:

1. Identify those feature options $f_! \in O^!_f \subseteq O_f$ that are not bound to true or false in the completed feature ambition ${}^{\mathcal{PD}}a^{cm}_f$, but that appear in the visibility v'_i of any depending element e'_i that causes failure of Constraint 6 (i.e., $v'_i({}^{\mathcal{PD}}a^{cm}) \neq true$).
2. For all $f_! \in O^!_f$, transfer the selection state from the check-out time choice c^{ch}_f, if bound there, to a^{cm}_f. (This cannot guarantee in general that e'_i are visible.)

In the feature ambition specification dialog, the user may bind additional features, but never delete any existing bindings from the recommended ambition a^{cm}_f. In case the second dialog is canceled, however, the commit is definitely aborted and the user must revise his/her local workspace modifications.

14.5.3 Handling Multi-Version EMF Models

In Section 10.6, particular details with respect to the connection between intrinsic and extrinsic representation of EMF models have been left unexplained. The most relevant implementation remarks are summarized below.

Two-Phase Importing and Exporting of EMF Models. Both intrinsically and extrinsically (see Figure 10.8 on page 205), EMF models are highly connected graph structures, such that the question arises in which order the elements are processed during the operations IMPORT and EXPORT. Both transformations have been implemented as two-phase procedures, of which we here describe IMPORT representatively:

– Traverse the spanning containment tree of each EMF model part of the workspace in a top-down way. At this occasion, instances of Object, (subclasses of) ClassRef, (subclasses of) FeatureRef, EMFAttributeValue, and EMFContainmentRefVal are created. In a *trace*[11], the created instances of Object are associated with their intrinsic source EObject.
– Iterate over all instances of Object stored in the map (touching every object in the workspace for a second time). Resolve cross-references with the help of the map, and create corresponding instances of InternalRefVal, or ExternalRefVal in case no corresponding target Object exists. Last, for all references to multi-valued structural features, create a corresponding instance of OrderedCollection and set it as valueOrder.

To correctly resolve internal references to meta-concepts (i.e., InternalClassRef and InternalFeatureRef), the whole two-stage procedure is separately applied for two meta-levels. First, *metamodels* (carrying the extension `.ecore` in the workspace), and thereafter, *model instances* (having different file extensions) are processed.

Proxy-Based Comparison and Merging. With a similar motivation, extrinsically represented multi-version EMF model instances are temporarily converted into trees clear of cross-references before the operations MATCH and MERGE are applied to them. To this end, an internal proxy strategy (orthogonal to EMF's built-in proxy mechanism) has been implemented.

In particular, all instances of InternalRefVal, InternalClassRef, and InternalFeatureRef are extended such that their references to Object are replaced by a proxy attribute that gets the UUID of the target object assigned.

Both the implementation complexity and the run-time of MATCH and MERGE are positively affected by this optimization. After applying the corresponding operation, proxies are replaced by object links again. To this end, a strategy similar to step 2 of the Import operation described above comes into play.

Intrinsic vs. Extrinsic Object UUIDs. Depending on the concrete EMF-based tool(s) employed, intrinsically represented model resources may or may not attach string-valued UUIDs to individual objects. In case UUIDs are available, they can be straightforwardly

11 Traces creating during EXPORT are reused for a-posteriori product-based validation; see below. Corresponding *export traces* are created not only for EMF models but for all product dimensions and parts thereof.

transferred to the corresponding extrinsic object part of the repository during IMPORT, and in the opposite direction during EXPORT. In order to guarantee the uniqueness also among different resources and to avoid ambiguity problems with external modeling tools, the UUID is *reversed* at each conversion between intrinsic and extrinsic model representation.

Fragment Paths as Pseudo UUIDs. In case UUIDs are not used intrinsically, however, a corresponding extrinsic UUID needs to be created nonetheless, in order to provide for correct and meaningful behavior of the operations MATCH and MERGE, which rely on the UUIDs as sameness criteria (see Sections 10.6.2 and 10.8). Our technical solution here uses the *fragment paths* encoded in the XMI file as *pseudo UUIDs*. This solution is, however, not stable with respect to rename or move operations, as fragment paths may contain element names or absolute positions in a collection belonging to the container object. To pseudo UUIDs, the reversing strategy described above is not applied.

14.5.4 The Server Side Application

The central remote repository is deployed as a Tomcat servlet in a self-contained *web archive* (WAR). Below, the architectural remarks given in Section 14.4.5 are detailed.

Dependencies. Being embedded into the Eclipse platform, the client side of the application has an inevitably long list of dependencies. It was an important goal to reduce the number of external libraries to be exported to the server side application.

The external dependencies of the server side include the REST API *Jax-RS*, the dependency injection framework *Guice*, the core of the *Eclipse Modeling Framework* (excluding *Edit* and *Editor* plug-ins), as well as the tiers metamodels (excluding all meta packages), services (excluding package client), and commands (restricted to packages core, server, collaborative, and collabhybrid). All UI-related dependencies are decoupled.

Physical Organization of Repositories. As mentioned before, a SuperMod server application is capable of managing multiple independent repositories, each of which is identified by an individual *repository path* (see variable `repoPath` used in the queries in Section 14.4.5).

As illustrated by Figure 14.13, different repository paths are mapped to physical folders in the working directory of the servlet application. Here, the same repository data as on the client side are managed, with one exception: the local workspace descriptors, which are part of the client-specific metadata section only.

Low-Level Synchronization Strategies. In Section 13.2.1, it has been explained that the conceptual framework provides a dedicated metamodel for collaborative metadata, a subset of which are meant to be managed by the server-side repository. Unless suggested in Figure 13.1, however, SuperMod represents *master metadata* and its contents not as a part of a model instance, but uses two low-level text files for this purpose. [12]

12 The reason for this is rather technical; the EMF resource framework cannot readily avoid that a model file is opened in multiple parallel sessions because queries are executed in different static contexts.

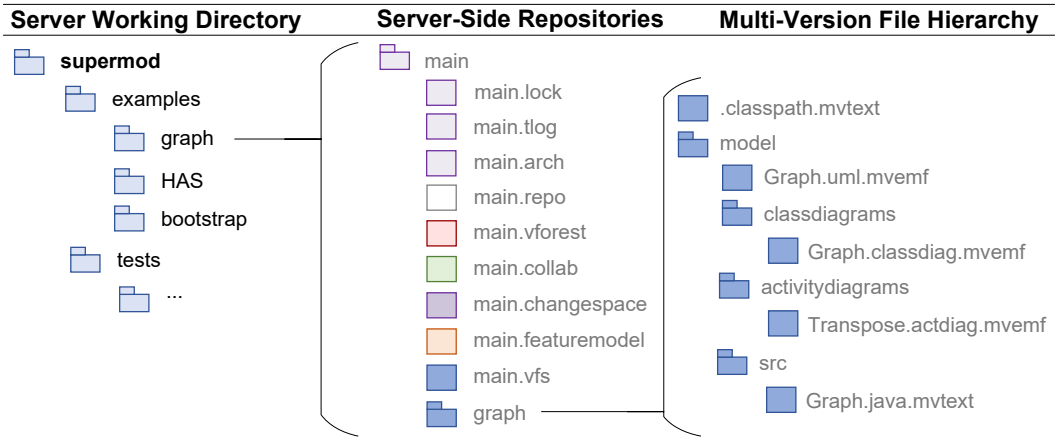


Figure 14.13: Physical organization of repositories on the server side.

More precisely, the *semaphore lock* (see main.lock) and the *transaction log* (see main.tlog in the example in Figure 14.13) are represented in an individual textual format. This way, the *isolation* of write transactions is already ensured at file system level. In the case of concurrent write access, a suitable response code signals to the client that the request should be re-attempted later.

An additional text file is dedicated to the distinction between different *repository architectures* supported (see Section 14.3). In order to ensure that all clients and the server agree with the architecture assumed for a specific repository, each repository folder contains a plain text file (see main.arch in the example), where the individual *repository architecture code* is recorded. This remains stable throughout the life-cycle of a server-side repository (i.e., until DESTROY).

Peripheral Web Service Requests. In addition to the aforementioned CREATE, PULL (including the special case CLONE), and PUSH, the web service offers additional low-level requests, which are summarized as *peripheral requests* here.

Ping. A simple request that checks whether a server-side repository, qualified by a specific repository path, is capable of receiving further requests. It is transparently invoked during the processing of all client commands prior to the actual requests.

Browse. Returns a list of valid repository paths available below a specified root URI. The query is executed by the user interface of the commands SHARE (in order to avoid collisions with existing repository paths) and CLONE (for suggesting paths of repositories to connect to).

Repository Architecture Code. Returns the *repository architecture code* assigned to a specific repository qualified by a given path. Used during the CLONE command in order to inject the correct dependencies that match the cloned repository's architecture.

Transactions. Offers various types of requests for the latest *read* and *write transaction numbers*, and for generating new public revision numbers. Transparently invoked by

the client in advance to the operations PULL and PUSH, and after CLONE (starting a public transaction immediately).

Destroy. This request is issued by the client when the corresponding command is selected by the user. Rather than physically deleting the repository on the server, a *destroyed* flag – represented by an additional file main.destroyed – is set. Upon any future request, it is signaled to clients by a corresponding response code that the server-side repository is no longer available; the clients may decide whether to step back to single-user version control or whether to apply the local DISCONNECT command, which also destroys the client-side copy of the repository.

Currently, SuperMod does not support an authentication mechanism. In future, this might be offered as an additional peripheral request to extend the above list.

14.6 Product Well-Formedness Analysis

This section completes the description of the tool *SuperMod* by providing implementation remarks concerning the *a-posteriori product-based well-formedness analysis* strategy introduced in Chapter 13. The strategy can be summarized as follows: After filtering out a single-version product, *detect conflicts* based on the extrinsic representation (see implementation remarks in Section 14.6.1). Then, apply *default resolution* actions (14.6.2). Last, export the repaired product to the workspace and attach *conflict markers* (14.6.3), which enable the user to accept or to retrospectively adjust the performed resolution actions. By the subsequent commit, the corrections are made persistent for those variants that are included in the specified feature ambition.

The explanations refer exclusively to the *client side* application; recall that synchronization problems caused by concurrent modifications are addressed by the context-free three-way merging strategy introduced in Section 12.4.

14.6.1 Conflict Detection

For the detection of conflicts, a low-level service interface VALIDATE has been defined; it is supposed to be implemented by specific product dimension dimensions. The operation takes as input an extrinsically represented product dimension and returns a set of conflicts. For the versioned file system dimension, the operation is further decomposed according to the respective file types supported.

Without any exception, the product constraints listed in Sections 13.3 have been implemented. In general, the conflict condition is checked for each instance of the context class. If the condition holds, an instance of the corresponding conflict class is created and attached to the metadata section of the local repository (see Section 13.2.2).

The list of recently detected (and currently unresolved) conflicts is temporarily made available to the user in a dedicated *conflicts dialog*. An example is provided in Figure 14.14. The three conflicts displayed emerge from a conflicting realization of the features labeled (renaming of class Edge into LabeledEdge and insertion of an attribute label) and weighted (renaming into WeightedEdge; insertion of weight). The value order conflicts refer to the abstract syntax and to the order in the attributes compartment of the diagram, respectively.

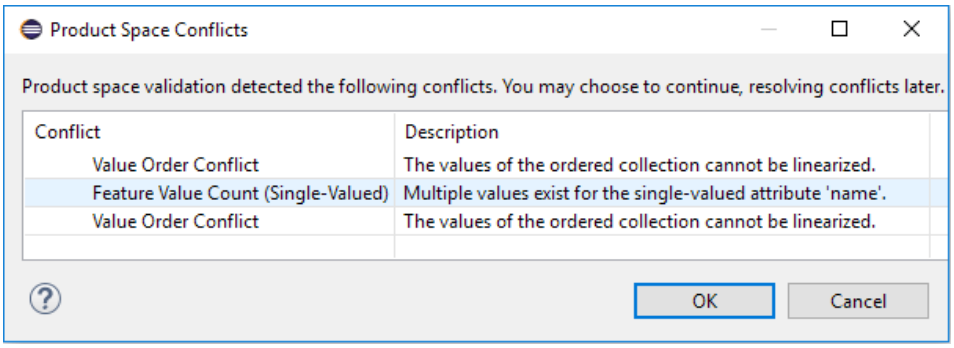


Figure 14.14: Example of SuperMod's conflicts dialog.

The user may accept the conflicts (OK), which triggers default resolution and the creation of conflict markers (see below). In case he/she selects Cancel, the surrounding check-out operation is aborted.

14.6.2 Default Resolution

Default conflict resolution has been implemented by another low-level service, DEFAULTRESOLVE, which takes as input the filtered product and the conflicts. Furthermore, the chosen *default resolution strategy* is available from a corresponding Eclipse preference page—the available options have been shown in Section 13.4.3.

In order to select a *preferred element* based on the selected default resolution strategy, the visibility of the context element is taken into consideration. For instance, given that the strategy "my change wins" has been selected and the visibility of an element refers to a revision option part of the locally running remote transaction, this element is preferred.

As sketched in Figure 13.7(b), the result of default resolution is a *preliminarily repaired filtered multi-variant domain model*. To this extrinsically represented product, the operation EXPORT can now be applied both deterministically and (syntactically) consistently.

14.6.3 Conflict Markers

After having exported the repaired MVDM into the workspace, the temporary *export trace* (see Section 14.5.3) is analyzed in order to identify those workspace elements that correspond to the context elements of specific products. Depending on the intrinsic representation, a workspace element can be an EMF object, a line of a text file, or a feature model element, for instance. This information is exploited for the creation of *conflict markers*, which basically implement *enhanced conflict descriptions* in the theoretical description of the redefined check-out operation provided in Section 13.4.2.

Conflict markers rely on an extension of the Eclipse-internal marker concept, which is also used, e.g., for showing compilation errors in source code files or for semantic model errors detected by the *EMF Validation Framework* [Ste+09].

Figure 14.15 depicts a screenshot of the markers generated based on the conflict set

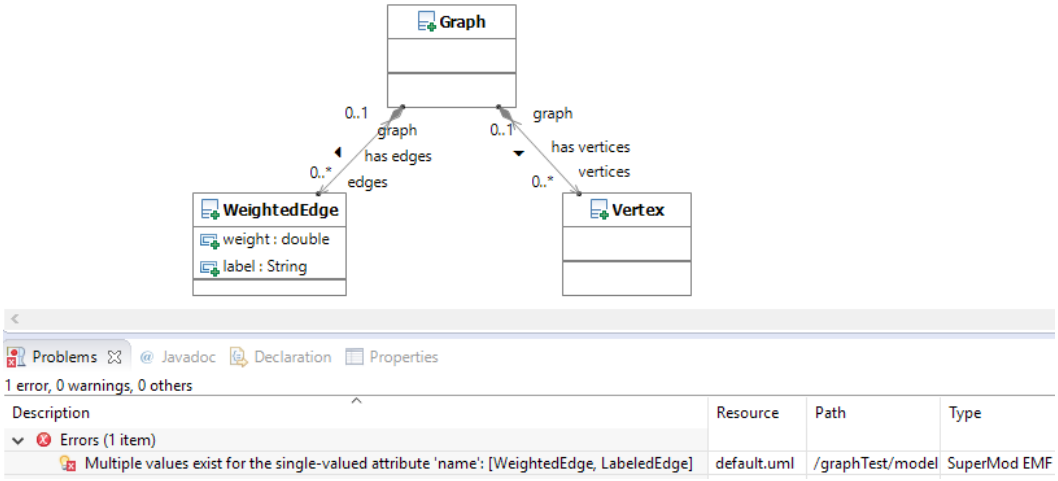


Figure 14.15: Conflict default resolution markers by example.

illustrated in Figure 14.14 above[13]. When the user selects an item, the focus of the editor opened in the workspace is automatically set to the workspace equivalent of the context element. Using his/her preferred editor, the user may now accept or manually revise the resolution action. After that, he/she can remove the marker from the list in order to indicate that the conflict has been resolved as desired.

On commit, all pending markers are deleted as it is assumed that the user accepts default resolution decisions he/she has not explicitly removed from the list. Moreover, depending on the specificity of the feature ambition, it may happen that the same conflict is reported repeatedly in a later iteration started with another check-out operation.

14.7 Related Implementation

We conclude this section by a brief discussion of tools that follow comparable implementation principles, or that use different technologies for similar purposes as SuperMod. We concentrate on tools that are built upon the Eclipse IDE.

The Ancestors of SuperMod. SuperMod has been technically influenced by two model management tools developed in precursor projects.

Firstly, the filtered MDSPLE tool *FAMILE* [BS12b; BS16a; BS16b] offers an optically similar feature model editor, which was also obtained from a generated EMF tree editor. When compared to SuperMod, this editor supports additional variability modeling capabilities, particularly *cardinality-based feature modeling* [CHE05]. Although FAMILE assumes an intrinsic multi-variant domain model, the *mapping model*, which connects domain model elements to feature expressions, is capable of virtually extending the MVDM by extrinsic *alternative mappings*. The metamodel of the mapping model shares similarities with the extrinsic EMF product dimension presented in Section 10.6.1.

13 The default resolution of order conflicts referring to ordered structural EMF features is not displayed as a marker if the corresponding meta-attribute ordered is set to false.

Secondly, *BTMerge* is an EMF-based tool for consistent three-way model merging. After a similarity-based matching, the model versions to be merged are combined into a *merged model graph*, which also resembles extrinsic EMF models used in SuperMod. This extrinsic representation is displayed to the user in the form of a three-column tree during the interactive merge. A *conflicts list*, which coarsely corresponds to the combination of conflicts dialog and markers in SuperMod, is presented to the user. In contrast, conflicts must be resolved deterministically in the dialog. The offered resolution actions are similar to the default repair actions used here; see also discussion in Section 13.5.

Feature Model Editing. For the creation as well as for the editing of feature models, many EMF-based editors have been described in the literature.

EMF tree editors are used, among others, in FAMILE [BS12b], *FeaturePlugin* [AC04], and *FeatureMapper* [HKW08].

Apart from trees, two additional forms of representation exist: In *Clafer* [BCW11] and in *PLiBS* [ZJ07], feature models are edited based on a textual syntax. Diagram-based feature model editors are provided by *SiPL* [Pie+15] and by *FeatureIDE* [Thü+14b].

Feature Model Satisfiability. SuperMod implements satisfiability checks by translating the invariants derived from a feature model into logical gates, which are then analyzed by the SAT solver Sat4j [BP10]. This strategy is shared with the tool *Feature IDE* [Thü+14b], where feature models may contain arbitrary propositional logical formulas.

Beyond SAT solvers, more specific solutions to feature model satisfiability exist. *FeaturePlugin* [AC04] utilizes *binary decision diagrams* [Ake78] for detecting conflicting relationships or constraints. Furthermore, *model checking* techniques are frequently employed for this purpose; a survey of *Alloy*-based (therefore not necessarily Eclipse-based) approaches is provided in [SPC16].

Client-Server Communication in Model VCS. SuperMod's technical solution for client-server communication for VCS relies on a REST-based web service. Alternative technical solutions (also those not relying on Eclipse) have been discussed in Section 6.2.3.

For comparison, the EMF-based model VCS *EMFStore* [KH10] uses XML-based *remote procedure calls* (RPC) in order to synchronize a central server with several clients over the physical network. In contrast to SuperMod, secure authentication – here built upon the *Secure Sockets Layer* (SSL) – is provided.

Conversely, the combination of EMF with REST has also been addressed in [EdD+16]. The technical framework *EMF-REST* allows to generate a REST-based web service API that offers direct model manipulation requests. Albeit, neither historical nor logical version management is supported.

14.8 Summary

The research prototype SuperMod is a model-driven tool for the integration of MDSE, SPLE, and version control. The tool relies on well-known formalisms (such as feature models and revision graphs) and metaphors (check-out and commit), and thereby provides an intuitive

yet decent user interface. It is also important to note that all of the three disciplines are supported optionally: By corresponding repository architectures, the feature model or the revision graph may be omitted. Furthermore, the tool may also be applied to purely source code centric projects (offering a line-oriented product granularity).

For collaborative versioning, SuperMod provides the metaphors pull and push. These have been implemented upon a REST-based web service that manages the central remote repository. The tool applies optimistic versioning. In the case of concurrent modifications, three-way merging is applied non-interactively; the context-free well-formedness is ensured not until conflicting revisions or features are combined during check-out. Conflict markers assist the user in reviewing automatic repair actions.

By the tool SuperMod, we have demonstrated that the conceptual framework elaborated in Part IV is implementable. The functionality offered by the tool covers the requirements listed in Section 2.3. The evaluation presented in the subsequent Chapter 15 aims at providing evidence that the usage of the tool offers practical benefits to the relevant stakeholders. Retrospectively, implementation gaps are critically reflected in the conclusion of this thesis.

14.9 Tool Availability

The tool SuperMod referred to in this chapter is publicly available for evaluation purposes. Here, resources for documentation and installation are listed. [14]

Client Side. The tool is available as a set of Eclipse plug-ins, consisting of a *core* application and one additional plug-in for each repository architecture (see Section 14.3). For the installation, we recommend a clean *Eclipse Modeling Mars* distribution. The plug-ins are allowed to co-exist in the same IDE; they may be retrieved from the following Eclipse update site: [15]

Server Side. In case one of the two collaborative repository architectures has been selected, it is mandatory to install in addition a server side application, which requires an *Apache Tomcat 7* web-server. A ready-to-deploy web archive can be found here: [16]

Tool Demo Video. The following tool demonstration video accompanies the descriptions given in [SW16b]: [17]

For initial experiments, it is recommended to install the plug-ins *SuperMod Core* and *Revision+Feature Layered Version Model* without server side application.

14 The availability of these links is guaranteed only as long as the research project is active.
15 http://btn1x4.inf.uni-bayreuth.de/supermod/update/
16 http://btn1x4.inf.uni-bayreuth.de/supermod/webapp/supermod-server.war
17 https://www.youtube.com/watch?v=5XOk3x5kjFc

Any program is only as good
as it is useful.

LINUS TORVALDS

Chapter 15

Evaluation

Abstract

This chapter is dedicated to the experimental investigation of the practical value of the formal approach, whose conceptual elaboration and implementation has been contributed in the preceding chapters of the thesis. Emphasis is put on those properties of the approach that immediately affect designated end users. The here presented results rely on three case studies conducted with SuperMod: an extended version of the running Graph Library example, a model-driven product line for Home Automation Systems, and a bootstrapping experiment where SuperMod itself is re-engineered as a product line based on a domain-specific modeling language. The data obtained in this way is analyzed in order to draw conclusions about the theoretical framework, in particular: the added value over filtered editing referring to the management of both the product space and the version space, impacts of the dynamic editing model, and properties of a-posteriori product-based analysis. In a retrospective critical discussion, we match the results as well as additional observations with properties postulated throughout the thesis.

Contents

15.1 Methodology

After having presented a conceptual framework for the integration of MDSE, SPLE, and version control, as well as an implementation thereof, we now transition to a discussion of the practical benefits of the approach. To this end, an evaluation based on three case studies is presented in this chapter. We provide a detailed description of the conduction of the case studies themselves, as well as a presentation and discussion of the results obtained by an analysis of these. We make a distinction between primary and secondary evaluation goals.

The case studies include an extended and collaboratively developed version of the running *Graph* example, a larger case study adopted from the SPL literature, namely a product line for *Home Automation Systems* (HAS), and finally, a *bootstrapping* case in which SuperMod's variable repository architecture is redesigned based on a domain-specific language.

The *primary* goal of the evaluation is to obtain an accountable measure of the added value that immediately affects designated end users. To this end, we apply a lightweight form of the *goal question metric* (GQM) approach [BCR94]: First, three specific goals of the conceptual framework are defined from the end user's perspective. Second, we ask four primary research questions whose answer should reflect whether the goals have been reached. Third, to answer the questions, we apply statistical metrics to the evaluation objects, which are represented by both *user action logs* and the transparent *repository contents* in their state after the conduction of the case studies.

As far as *secondary* results are concerned, we qualitatively evaluate properties of further aspects of the framework, e.g., the collaborative distributed version model, its compatibility with different modeling languages and generated source code, its ability to react to customer requests, and the effect of the fine-grained versioning strategy applied.

Distinctly and by intention, the evaluation does *not* include a technical examination of the research prototype SuperMod in terms of performance analyses, scalability experiments, or stress tests. The scientific contribution of this thesis consists in the conceptual framework, whereas the technical implementation is intended as a proof of concept and as a vehicle to conduct case studies that allow conclusions about the user-relevant properties of the

underlying conceptual approach. Thus, the evaluation presented in this chapter refers to the scientific but not to the technical contribution.

The remainder of this section is organized as follows: Subsequently, we define three primary goals, we explicitly phrase primary and secondary evaluation questions, and we sketch the metrics applied in order to answer the former. In Section 15.3, we introduce three case studies from which the experimental results have been obtained. Section 15.4 presents and discusses the applied metrics as well as results obtained; also, potential threats to validity are listed. Further discussion of secondary observations with respect to secondary questions are provided in Section 15.5. A summary concludes this chapter and part.

15.2 Goals, Questions, and Metrics

The evaluation is guided by three primary goals, as well as by four primary and by five secondary evaluation questions. The connection between goals, primary questions, and metrics is made explicit in Table 15.1. In addition, Figure 15.1 clarifies the connection between case studies and both primary and secondary evaluation questions.

15.2.1 Goals

The overall goal of the approach presented in this thesis is the integration of MDSE, SPLE, and VC in a single tool. SuperMod supplies a proof of concept, however, the question of the added value for potential users must be asked. In the following, we evaluate whether three particular goals – **G1**, **G2**, and **G3** – have been achieved, using state-of-the-art approaches (e.g., unfiltered or statically organized editing models, as well as product-based or family-based well-formedness analysis) as reference points:

G1. Reduce the effort of software product line development, compared to unfiltered editing.

Table 15.1: Goals, Questions, and Metrics guiding the primary evaluation.

Goals	Questions	Metrics
G1	**PQ1**	number of modified elements workspace/repository ratio
	PQ2	ambition complexity visibility complexity ambitions/visibilities quotient
G2	**PQ3**	share of iterations with new features in ambition explicit check-outs ratio interactive migration ratio unsatisfactory migration ratio migration effort
G3	**PQ4**	default resolution accuracy resolution scope

G2. Be less obtrusive with version definition tasks, compared to static filtered editing.

G3. Improve the well-formedness of derived product variants, compared to standard product-based SPL analysis.

15.2.2 Primary Questions and Metrics

According to Table 15.1, the first goal is refined into two primary questions, **PQ1** and **PQ2**, investigating the product and the version perspective of the SPL development effort, respectively, whereas **G2** and **G3** are straightforwardly mapped to individual questions **PQ3** and **PQ4**. The metrics informally introduced under the primary questions below are further refined in Section 15.4.

PQ1. To What Extent does the Filtered Editing Model Reduce the Complexity of Editing Products in the Workspace, Compared to Unfiltered Editing? Approaches based on *unfiltered MDSPLE* expose to the user a *multi-variant domain model*, which represents a superimposition of all available product variants. Depending on the number of variation points and the complexity of their realization, this may significantly complicate multi-variant editing (see Section 7.1.5) by overloading the user with irrelevant contents. In contrast, the here applied *filtered editing* model hides those elements from the workspace that are not relevant for the intended change.

To quantify the complexity of the edit steps themselves, we indicate the *number of modified elements* within each iteration. The reduction of cognitive complexity cannot be measured; as a replacement, we put in relation the number of (graphical or textual) elements the user is exposed to in the workspace and the number of corresponding elements belonging to the equivalent model resource or text file in the transparent repository. In this way, we

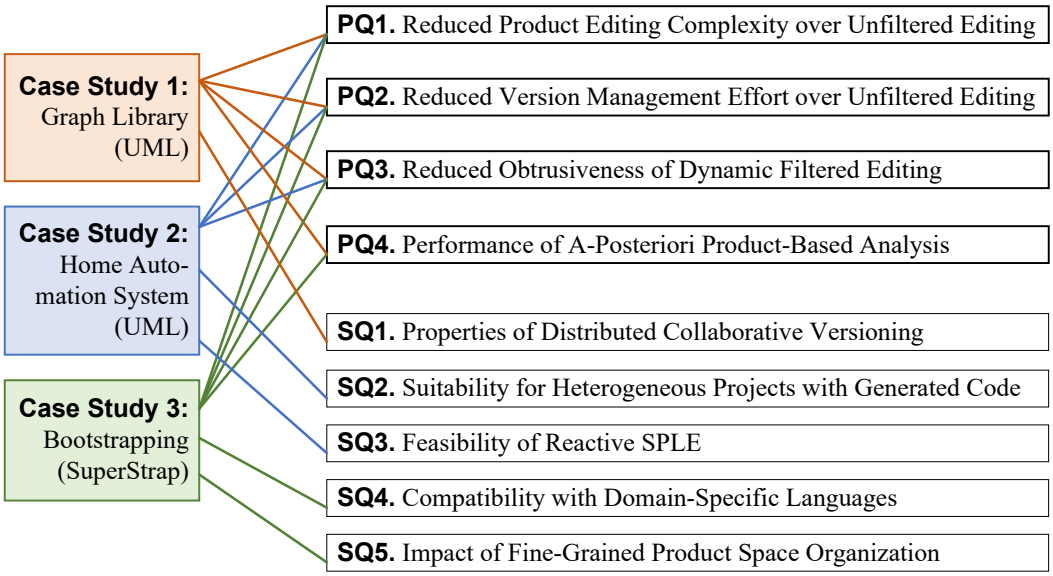


Figure 15.1: Case studies and evaluation questions in context.

obtain as metric the *workspace/repository ratio*, which is the complement to the degree of filtering applied in each specific editing model iteration.

PQ2. To What Extent does the Filtered Editing Model Reduce the Effort for Manual Version Management, Compared to Unfiltered Editing? Both the variability model and the platform, as well as traceability links in between have to be created and maintained manually when applying *unfiltered editing* as realized in state-of-the-art SPLE approaches. SuperMod and the underlying approach, in contrast, follow *filtered editing*; the developer is not exposed to multi-variant artifacts, but he/she performs modifications based on single-variant views. Traceability links are created automatically based on a user-defined version specification, the *feature ambition*, at each commit.

As a first indicator for the complexity of version management in filtered editing, we define the *ambition complexity* of each iteration as the number of clicks necessary for feature ambition specification. In analogy, the *visibility complexity* characterizes the number of elements found in the abstract syntax tree of a traceability link created manually.

To finally compare the version management effort of filtered editing with a corresponding unfiltered approach, we put in relation the number and complexity of feature ambitions defined and the number and complexity of visibilities transparently created. The *ambitions/visibility quotient* obtained in this way quantifies the degree of version management automation gained by the filtered editing model.

PQ3. To what Extent does the Dynamic Filtered Editing Model Behave Less Obtrusively than Static Filtered Editing? As explained before, the contributed conceptual framework differs from related approaches to filtered editing [WMC01; WO14] inasmuch as it applies a *dynamic* editing model, where the feature model is made available for modification in the workspace, such that new features can be introduced and realized in the same iteration. Moreover, the CHECKOUT operation is made optional by a new workspace operation MIGRATE that prepares the workspace choice for the subsequent iteration, assuming that the user wants to continue the next development iteration in the current workspace view. DFE was claimed to be less obtrusive (in terms of version definition tasks) than SFE.

The first metric applied for answering the question is the ratio of iterations in which *new features* are bound in the *ambition*.

In order to quantitatively assess the positive impact of the operation MIGRATE on the filtered editing workflow, we define and compute four additional metrics: first, the relationship between those situations where the migrated choice equals the choice desired by the evaluation subjects, and those situations where he/she had to issue an *explicit* CHECKOUT. Second, in case user interaction is required, we approximate the extra user effort of migration by the number of *user interactions* required for producing the desired choice. Third, we count the ratio of events where migration produced an *unsatisfactory* choice unsuitable for the subsequent iteration. Besides, the *migration effort* reflects the number of clicks necessary for each interactive migration.

All in all, the deduced quantities reflect the share of user interaction saved by dynamic filtered editing on the one hand as well as the precision of the operation MIGRATE on the other hand. In order to judge about the unobtrusiveness implied by the DFE model, we

compare the effort saved by the DFE model to the overall new user interaction caused by it.

PQ4. How Does A-Posteriori Product-Based Well-Formedness Analysis Perform when Compared to Ordinary Single-Product Analysis? In Chapter 13, *a-posteriori product-based* analysis has been presented. After a non-interactive default repair phase, the user may revise all well-formedness violations and the performed repairs in a single-variant workspace without having to analyze or modify multi-variant artifacts. The repair actions are, however, not restricted to the product variant presented in the workspace, but they potentially affect all variants included in the ambition.

Analyzing concrete occurrences of well-formedness repair in the case studies, this primary question is answered in a twofold way. First, we determine the *accuracy* of *default resolution* by counting the number of conflict repairs that matched the user's intents. Second, we put the *affected* product variants, e.g., those included in the ambition, in relation with the lone product variant in which the repair actions are applied. For each product conflict, we determine the *resolution scope*, i.e., the percentage of valid product configurations that become transparently affected by the applied conflict resolution action. This number serves as a measure for the efficiency of the here applied analysis strategy compared to approaches restricted to single-product repair.

In Figure 15.1, this primary question is not connected to the Home Automation case study since product conflicts do not occur there.

15.2.3 Secondary Questions: Specific Properties of the Framework

In addition to data-based evaluation, backed by the four primary questions listed above, we qualitatively evaluate and discuss properties of the conceptual framework that cannot be deduced in an objective and reproducible way in the form of metrics. These properties are reflected by the following secondary evaluation questions:

SQ1. What are the limitations and implications of the *distributed collaborative versioning* strategy?

SQ2. How suitable is the presented approach for *heterogeneous* software product lines that contain both models and text files connected by implicit and explicit relationships?

SQ3. Is *reactive SPLE* – i.e., the ability to respond to customer feedback on demand – adequately supported by the presented conceptual framework?

SQ4. Is the approach compatible with *domain-specific modeling languages*?

SQ5. What are the positive and negative implications of the applied *fine-grained versioning* strategy?

In contrast to primary questions, secondary questions are addressed by individual case studies rather than by an aggregation of quantitative results emerging from all studies (see connections in Figure 15.1). This explains why the case studies are conducted under different specific experimentation conditions (e.g., single/multi user mode, different modeling languages and resource types, and varying amount of alternative variability).

15.3 Case Studies

In this section, the case studies conducted with regard to the evaluation are presented. The evaluation subjects – represented by the author himself and a master student; see individual case studies – developed the multi-version artifacts from scratch based on a list of requirements provided beforehand. These requirements lists also contain synthetic obstacles emerging from secondary evaluation questions. For the reason of compactness, we only present relevant extracts of the resources available in the local workspace.

15.3.1 The Extended Graph Library Case Study

Throughout this thesis, excerpts of a product line for *Graph Libraries* have been used as examples for demonstrating particular aspects of the contributed conceptual framework. The case study was initially described as an evaluation case in [LHB01] and adapted for the demonstration of many SPL approaches in literature. Below, we present a complete version of the product line that has been developed by the author from scratch[1], guided by [LHB01].

Specific Experimentation Conditions. To address **SQ1**, *collaborative MDSPLE* was simulated by involving two fictional developers, Alice and Bob; the actions of both were executed by the author of this thesis. In this experiment, the server side application of SuperMod, which managed the central remote repository, was running on the host machine; Alice's and Bob's local repository and workspace were physically separated in two independent Eclipse installations.

Requirements. To make the experimentation of this synthetic case study as realistic as possible, the requirements for the product line were assigned to different product line increments realized by Alice and Bob alternatingly. Furthermore, intentional obstacles were created, such that synchronization conflicts and product well-formedness violations came into play. The list of task descriptions is reproduced here:

1. Alice: Initialize the repository. Add basic support for graphs that contain a node set and an edge set. Assume graphs are undirected, but do not define a feature for this.
2. Bob: Clone Alice's repository. Introduce a mutually exclusive distinction between directed and undirected graphs. Ensure that Alice's realization of edges is valid for undirected graphs only, and commit a new realization for directed graphs.
3. Alice: Concurrently to Bob's change above, add support for labeled and weighted graphs as two optional, independent features. Reconcile your changes with Bob's.
4. Bob: Wait until Alice pushes revision 3. Add support for colored graphs.
5. Alice: Add constructors to the classes for vertices and edges. Their parameters should match the available properties for colored/labeled/weighted/directed graphs.

1 Therefore, revision numbers and details of both the feature model and the domain model may differ from previously presented examples. Apart from this, similar case studies were reported on in [SW16b] and in [SW17b]; the experimentation was repeated here, since a more detailed analysis of the user interaction was required in this evaluation.

6. Bob: Concurrently, define new features for the graph algorithms *transpose* (requires a directed graph) and *shortest path* (requires a directed and weighted graph). Reconcile your changes with Alice's.

Preliminaries. The case study was conducted with SuperMod in combination with the UML modeling tool *Valkyrie* [Buc12].

The technical preparations were made in Alice's fictional modeling environment. After having initialized the Valkyrie project using the provided project wizard, the domain model created is represented as two models, an abstract syntax model (an instance of the Eclipse UML 2.0 metamodel), and a GMF notation model that defines concrete graphical class diagram syntax for abstract model elements.

Table 15.2: Version history underlying the *Graph Library* case study.

Rev.	Author	Feature Amb.	Change Description
1.1	Alice	**true**	Eclipse metadata and empty model files
1.2		**true**	added Graph, Vertices, Edges to feature model
1.3		Graph	added Graph to domain model
1.4		Vertices	class Vertex, assoc. has Vertices
1.5		Edges	class Edge, assoc. has Edges
1.6		Edges	added association connects
3.1	Bob	**not** Undirected	added XOR-group with Directed, Undirected; removed association connects
3.2		Directed	associations starts at, ends at
2.1	Alice	Weighted	optional features Weighted, Labeled; attribute weight
2.2		Labeled	attribute label
4.1	Bob	Colored	optional feature Colored; class Color, association has Color
5.1	Alice	Vertices	empty constructor Vertex
5.2		Colored	constructor parameter color
5.3		Edges	empty constructor Edge
5.4		Undirected	constructor parameter adjacents
5.5		Directed	constructor parameters source, target
5.6		Weighted	constructor parameter weight
5.7		Weigh. **and** Undir.	defined order for adjacents/weight
5.8		Labeled	constructor parameter label
6.1	Bob	**true**	features Algorithm, Transpose, ShortestPath
6.2		Transpose	operation transpose
6.3		ShortestPath	operation shortestPath
6.4		ShortestPath	defined order Vertex/shortestPath

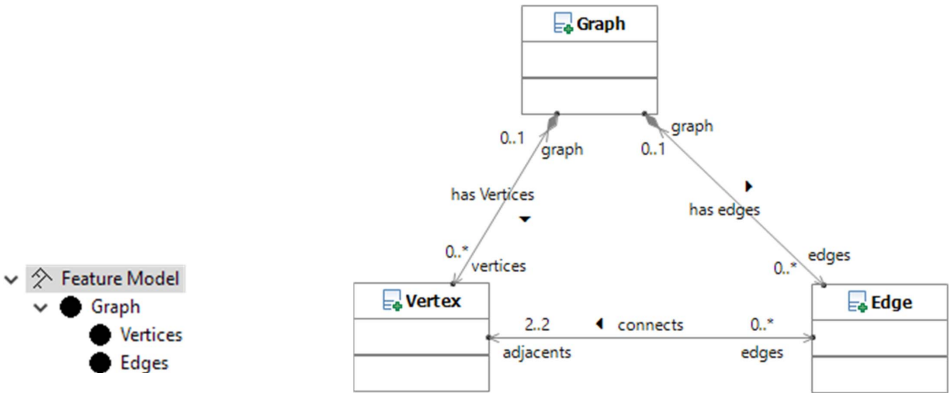
To this Eclipse project, SuperMod support was added by the Share command. After selecting the *hybrid collaborative* repository architecture in a first step, the connection to the central remote repository was established. In addition to the model files, two text-based Eclipse project metadata files (.project and .classpath) were selected in the initial resource selection dialog.

Overall Version History. Before we present the key user actions performed in the subsequent remote transactions, we summarize the complete version history of the case study in Table 15.2. Recall that public revisions are ordered by creation date rather than by push date; therefore, revision 3.2 is less recent than revision 2.2.

The majority of private revisions was processed in what is referred to as "the straightforward way" below: introduce a new feature, add corresponding realization artifacts to the domain model, and commit, using an ambition where the newly introduced feature is positively bound and no further bindings are set.

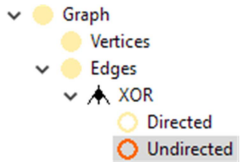
Public Revision 1. In revision 1.1, Alice commits the initial workspace contents using an empty ambition (as no features are defined yet). Revision 1.2 exclusively refers to the feature model—the result of the corresponding modifications is shown in Figure 15.2(a). In revisions 1.3 until 1.5, the features Graph, Vertices, and Edges are realized straightforwardly (cf. Table 15.2). Revision 1.6 is an evolutionary change and therefore addresses the same set of variants (ambition $\{(O_{Edges}, true)\}$). The resulting view on the domain model – which equals here the multi-variant domain model as all features are mandatory so far – is depicted in Figure 15.2(b). Alice concludes the remote transaction by a PUSH. Public revision 2 is started transparently in Alice's workspace.

Public Revision 3. Bob joins the fictional project based on a CLONE of revision 1 pushed by Alice. The remote transaction running in his workspace gets the public revision number 3 assigned.

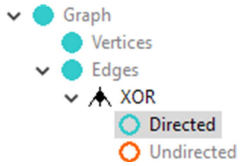


(a) feature model *(b) domain model in the lone available variant*

Figure 15.2: Alice's workspace contents after public revision 1.

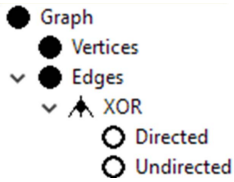


(a) feature ambition for revision 3.1

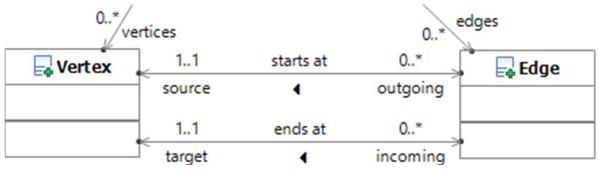


(b) feature choice migration before revision 3.2

Figure 15.3: Variant definitions provided by Bob in revision 3.



(a) feature model



(b) domain model (directed graph)

Figure 15.4: Bob's repository contents at the end of revision 3.

After defining the mutually exclusive features Directed and Undirected, Bob is faced with the problem that the realization of the second feature is already present in the workspace. The corresponding elements, therefore, need to be constrained to Undirected variants. Bob achieves this by removing the elements in question from his workspace and by committing this deletion using an ambition where Undirected is negatively selected[2]—see Figure 15.3(a).

Subsequently, Bob is prompted for the definition for an option binding for Directed in an interactive migration. Selecting $true$, this produces the correct choice for the change realized in revision 3.2, namely the definition of two associations for directed edges. An explicit check-out is not required either. Figure 15.4 depicts Bob's repository contents at push time. Public transaction 4 is started in his workspace.

Public Revision 2. Alice starts her work on the subsequent public transaction in advance to Bob's push; therefore, the version history is temporarily branched. In revision 2.1, she introduces the features Weighted and Labeled. Furthermore, she realizes the former feature in the domain model.

After committing, Alice is prompted for an interactive choice migration since the binding for feature Labeled is missing. She specifies a positive selection (cf. Figure 15.5(a)) because this matches her intention for the subsequent private transaction 2.2, where she realizes this feature straightforwardly. The final workspace contents are depicted in Figure 15.5(b).

As prescribed by the experimentation requirements, as soon as Alice attempts to push, a synchronization obstacle occurs as Alice's version history must be reconciled with Bob's public revision finished in the meantime. Following SuperMod's collaborative workflow, an *out of date* situation is signaled to Alice, who is forced to pull and to update her workspace to the latest revision. As her current workspace choice does not provide bindings for the

2 Through this "double negation", the visibilities of deleted elements will be automatically conjuncted with the feature option belonging to Undirected. The same principle has also been applied in revision 3 of the example presented in Section 9.5.2 and will be applied in revisions 35 until 37 of the Home Automation Example.

(a) feature choice migration before revision 2.2 *(b) workspace domain model (weighted, labeled graph)*

Figure 15.5: Version space and workspace presented to Alice in revision 2.

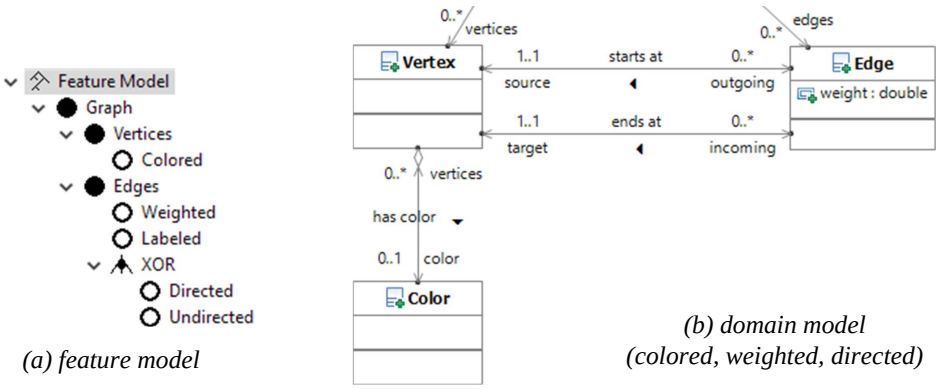
(a) choice before Alice's selections *(b) choice after Alice's selections*

Figure 15.6: Interactive choice migration after Alice's pull during revision 2.

remotely defined features Directed and Undirected, Alice is forced to complete her workspace choice interactively—see Figure 15.6.

Alice's and Bob's concurrent modifications are, however, not conflicting at product level, such that no well-formedness violations occur in Alice's workspace after pull and update. She may therefore resume her push without any further restrictions.

Public Revision 4. Bob waits for Alice to push, preventing synchronization problems beforehand. Then, he checks-out a weighted, directed, unlabeled variant. The realization of the feature Colored defined by the requirements list happens in a straightforward way—see Table 15.2. Figure 15.7 depicts Bob's workspace contents as he pushes.



(a) feature model *(b) domain model (colored, weighted, directed)*

Figure 15.7: Bob's workspace contents after public revision 4.

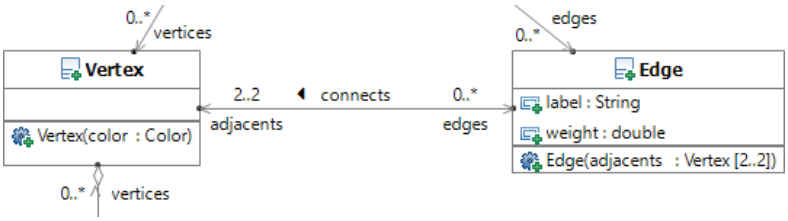


Figure 15.8: Alice's workspace domain model after private revision 5.4.

Public Revision 5. Alice, too, avoids synchronization problems by awaiting Bob's push. For defining the constructor of class Vertex, she checks-out a colored, labeled, weighted, undirected variant. The realization is split into two distinct commits (with different ambitions): In revision 5.1, the constructor is introduced without parameters for the feature Vertices. The parameter color is added conditionally for Colored variants in revision 5.2.

Still in the same workspace view, Alice defines the constructor for Edge in revision 5.3, as well as the parameter for undirected edges (adjacents of type Vertex with multiplicity 2) in revision 5.4. The intermediate workspace contents are depicted in Figure 15.8.

Thereafter, the constructor parameters for directed graphs are addressed. To this end, Alice must check-out based on a different choice, selecting feature Directed as well as all other non-conflicting optional features. In revision 5.5, the parameters source and target are added conditionally for Directed graphs. In the same view, the parameter weight is added for Weighted graphs in revision 5.6; see Figure 15.9.

In this way, the mutual order of the constructor parameters belonging to Directed and Weighted is fixed; however, no mutual order for adjacents and weight has been defined, although the features are allowed to be combined. In order to avoid well-formedness violations in future products derived, Alice uses revision 5.7 to make the order explicit. Switching back to her first choice (checking-out an undirected, colored, weighted, labeled graph), an *order conflict* is reported to Alice as expected. This is default-resolved by placing adjacents before weight, which matches Alice's preference. For the subsequent commit, she must specify an ambition that is specific enough for the performed change—she positively selects the features Undirected and Weighted, which are realized by the affected parameters.

Last, Alice straightforwardly adds a last constructor parameter for Labeled graphs. After this, a total order (except for the parameters belonging to the mutually exclusive features Directed and Undirected) is defined in the transparent multi-version collection that organizes the constructor parameters of Edge. The workspace domain model present in Alice's workspace as she pushes is shown in Figure 15.10.

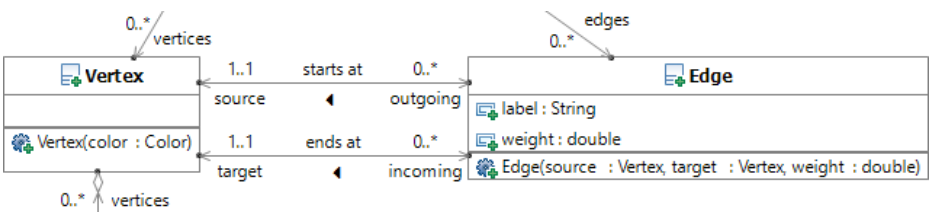


Figure 15.9: Alice's workspace domain model after private revision 5.6.

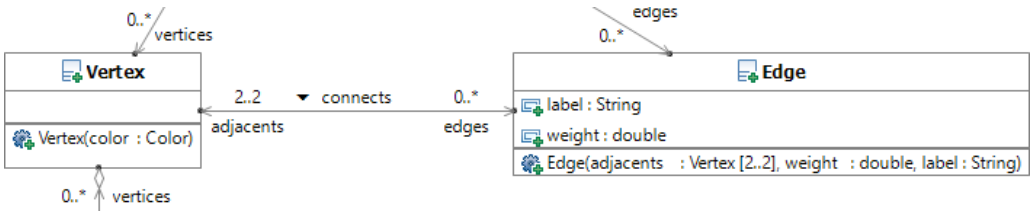
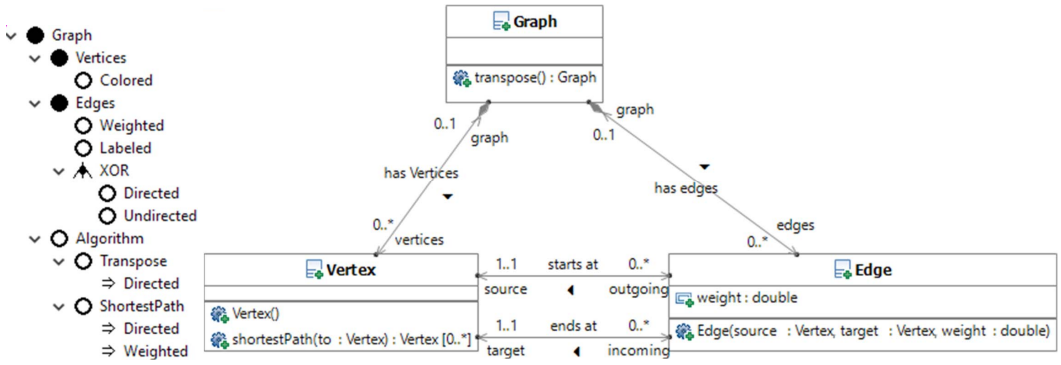


Figure 15.10: Alice's workspace domain model after private revision 5.8.



(a) feature model *(b) domain model (weighted, directed)*

Figure 15.11: Bob's final workspace contents after public revision 6.

Public Revision 6. Without awaiting Alice's push, Bob contributes the final public revision in the straightforward way. The first revision 6.1 is used to introduce features and constraints for the algorithms Transpose and ShortestPath; see Figure 15.11(a). These are positively selected in the subsequent interactive migration and then realized as operations using the corresponding features as ambitions in revisions 6.2 and 6.3. Then, Bob attempts to push, being forced to pull Alice's changes connected to public revision 5 first.

Since both Alice and Bob add elements to the operations list of class Vertex, another *order conflict* is raised here. This time, the default resolution strategy suggests an order that is not in line with the usual convention that constructors are placed before regular operations: [shortestPath, Vertex]. Therefore, Bob manually revises the repair action and switches the order. The resulting workspace content depicted in Figure 15.11(b) is committed under ambition $\{(o_{ShortestPath}, true)\}$. Bob pushes the final state of the Graph case study.

15.3.2 Home Automation System

In a second evaluation case[3], we apply SuperMod to the standard of a product line for *Home Automation Systems* (HAS) from [PBL05]. A HAS consists of several technical components communicating with each other; since most components are optional and interchangeable, this case study is frequently used for product lines. We here abstract from the communication layer and consider a requirements-centric view of the system.

3 This section is based on [SBW16a, Section 4].

Table 15.3: Version history of the use case diagram. Based on [SBW16a, Table 1].

R.	F. Amb.	Feature Model Changes	Changes to Use Case Diagram
1	HAS	feature HAS	actor Resident, component HAS
2	Id.Mech.	feature Id.Mech.	component Id.Mech., contained use cases, includes, and connected use links
3	DoorLock	feature DoorLock	comp. DoorL., use cases Lock, Unlock
4	DoorLock	—	missing links for Lock and Unlock
5	AlarmAct.	feature AlarmActivision	component AlarmAct., contained use cases, connected links
6	SMSToO.	features Ac.Sig., Vid.S., PoliceInf., SMSToO.	use case Change Phone Number
7	Heat.Cont.	feature Heat.Cont.	component Heat.Cont. and contents

Specific Experimentation Conditions. In contrast to the Graph study, where the secondary evaluation question of collaborative versioning was addressed, single-user mode was employed here. To this end, we applied the *hybrid* repository architecture. Conversely, this case study features a considerably larger version history, a domain model that is distributed over several *heterogeneous* model and non-model (Java source code) resources (**SQ2**). The case study was conducted by a master student with MDSE and SPLE background. Due to the size of the product line, the descriptions are less detailed than those referring to the Graph case study.

The larger part of the case study is organized in a rather plan-driven way. Consecutively, the activities *analysis*, *design*, and *implementation* (based on generated source code) are executed. For analysis and design, we rely on UML *use case*, *activity*, *package*, and *class diagrams* using *Valkyrie* [Buc12] (see above) and its Java code generator. At the end, fictional *customer feedback* that required to revisit design decisions in a *reactive* fashion (**SQ3**) is given.

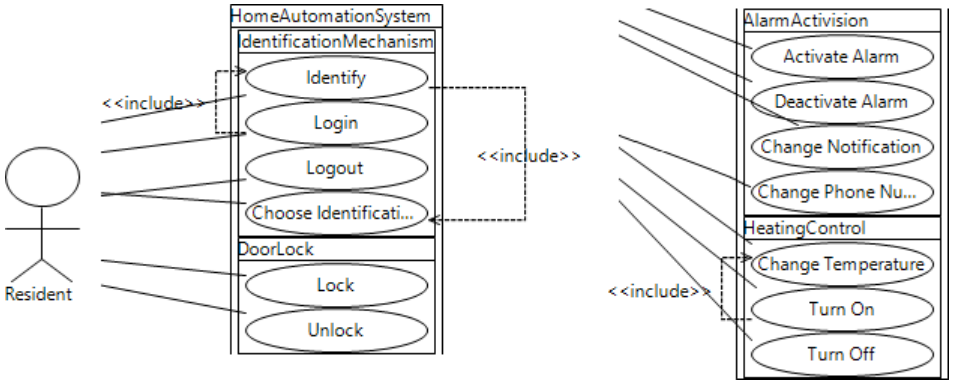


Figure 15.12: The use case diagram of the *HAS* study after revision 7, shown in a variant that includes all mandatory and optional features available. From [SBW16a, Figure 7].

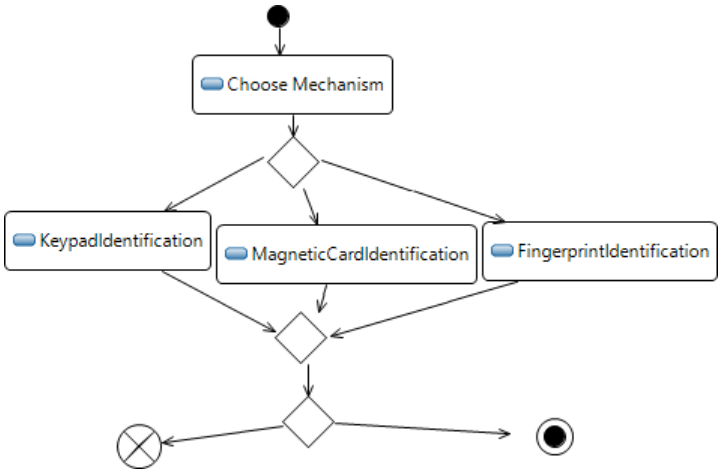


Figure 15.13: The activity diagram of the use case Identify after revision 12, shown in a variant that includes all sub-features of IdentificationMechanism. From [SBW16a, Figure 8].

Requirements Analysis. Requirements analysis is split into two phases. To begin with, residents' interactions with the HAS are documented in a use case diagram. Subsequently, one use case is representatively refined by means of an activity diagram.

After having initialized a Valkyrie project and having connected it to SuperMod version control, the first phase is initiated based on an empty use case diagram. In consecutive iterations, we introduce actors, components, use cases, and relationships as summarized in Table 15.3. The table also shows that the feature model is developed simultaneously, introducing new features on demand in order to delineate the scope of the respective changes. Figure 15.12 depicts a variant of the final use case diagram.

During the second analysis phase, the feature IdentificationMechanism is further refined by adding three concrete mechanisms, namely Keypad, MagneticCard, and FingerprintScanner. Features representing these mechanisms are organized in an OR-group, meaning that at least

Table 15.4: Version history of the activity diagram for Identify. Based on [SBW16a, Table 2].

R.	F. Amb.	Feature Model Changes	Changes to Use Case Diagram
8	HAS	added XOR groups below DoorL. and HeatingCont.	—
9	Id.Mech.	—	initialized diagram, added initial and final nodes, Choose Mech., decision/ merge nodes, and flows
10	Keypad	OR group with Keypad, Mag.Card, Fp.Scanner	added action KeypadIdentification and incoming/outgoing flow
11	M.Card	—	added action Mag.Card.Id. and incoming/outgoing flow
12	Fp.Scan.	—	added action FingerprintId. and incoming/outgoing flow

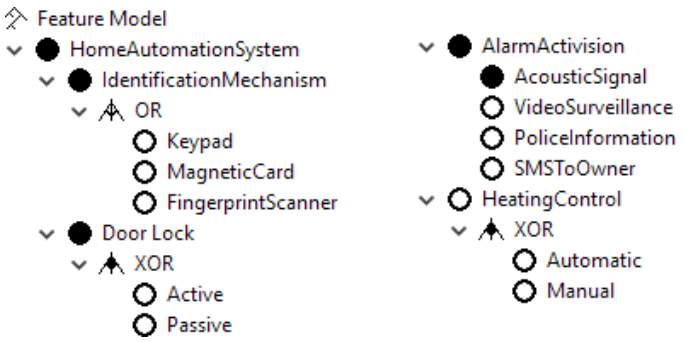


Figure 15.14: The final HAS feature model after revision 12. From [SBW16a, Figure 9].

one mechanism must be chosen in a valid configuration. In case several mechanisms are available, one of them must be chosen during identification at run-time (see below). The available selection should be restricted by the active features; this is realized in revisions 10 until 12 shown in Table 15.4. The resulting activity diagram is depicted in Figure 15.13; Figure 15.14 presents the feature model in its state at the end of the analysis phase.

Design. The static structure of the HAS product line is also developed in two phases. After modeling an initial package diagram, specific packages are refined by class diagrams.

Table 15.5 indicates that the package diagram (see Figure 15.15) is developed in an

Table 15.5: Version history of the package diagram. Additional horizontal lines indicate that an explicit check-out was necessary between the affected revisions. Based on [SBW16a, Table 3].

R.	Feature Amb.	Changes to Package Diagram
13	HomeAutomationS.	added package has and contained class HAS
14	Ident.Mechanism	package identification, class Id.Mech., interface IMech.
15	Keypad	added class Keypad
16	MagneticCard	added class MagneticCard
17	FingerprintScanner	added class FingerprintScanner
18	DoorLock	added package doorLock and interface IDoorLock
19	Active	added class ActiveLock
20	Passive	added class PassiveLock
21	AlarmActivision	package alarm, class AlarmAct., interface IAlarmService
22	AcousticSignal	added class AcousticSignal
23	VideoSurveillance	added class VideoSurveillance
24	PoliceInformation	added class PoliceInformation
25	SMSToOwner	added class SMSNotifier
26	HeatingControl	package heating, contained interface IHeatingControl
27	Automatic	added class heating::Automatic
28	Manual	added class heating::Manual

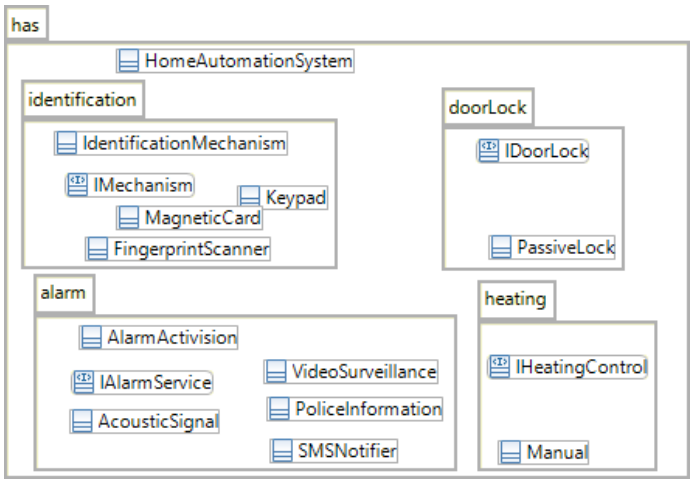


Figure 15.15: The package diagram after revision 28. The product variant shown does not include features Active and Automatic, thus not classes doorLock::ActiveLock and heating::Automatic, either. From [SBW16a, Figure 10].

Table 15.6: Revision history of the class diagram refining package identification. Based on [SBW16a, Table 4].

R.	Feature Amb.	Changes to Class Diagram for package identification
29	Ident.Mech.	initialized diagram, detailed class Id.Mech., interface IMech.
30	Keypad	interface realization from class Keypad
31	MagneticCard	interface realization from class MagneticCard
32	Fp.Scanner	interface realization from class FingerprintScanner

iterative and incremental way by realizing one feature after another. Variation points are anticipated by static modeling of appropriate design patterns, namely *Strategy* and *Command* [Gam+95], which are thereafter refined by class diagrams. Here, we refrain from introducing new features during the design phase.

As shown in Table 15.6, the package identification is refined by a class diagram, exemplifying the realization of (implicit and spontaneous) variation points during design. In revision 29, general details are added to the class IdentificationMechanism as well as to the interface IMechanism that realizes the *command* pattern. Its specific realizations are added subsequently and scoped with the respective feature. In this case, the only necessary changes are to make the respective command classes realize IMechanism (see Figure 15.16). Similar refinements might have been applied to the packages doorLock, alarm, and heating.

Implementation. In the here considered model-driven product line, the structural part of the source code can be derived from the artifacts developed in the design phase using *Valkyrie*'s Java source code generator. The main class HomeAutomationSystem shall contain the main executable as command line application. Below, we confine the presentation to the implementation of the method identify() of class IdentificationMechanism, which implements

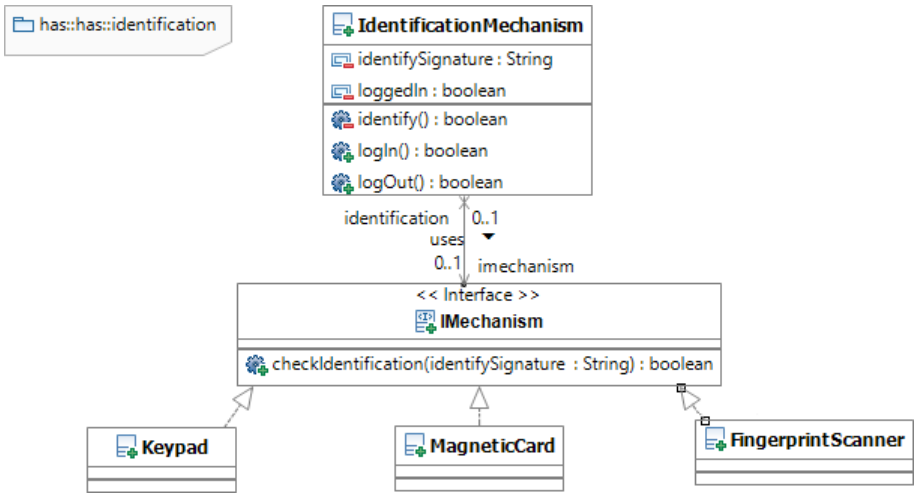


Figure 15.16: The class diagram that refines package identification (revision 32), with Keypad, MagneticCard, and FingerprintScanner selected. From [SBW16a, Figure 11].

the activity diagram shown in Figure 15.13 above.

Variability is achieved by making the declarations and usages of specific mechanism classes dependent on their respective features. As indicated in Table 15.7, after the initial code generation run in revision 33, a variant that includes all identification mechanisms available is made available in the workspace. In revision 34, a multi-variant implementation is provided by manual source code extensions (see Listing 15.1). We then connect the variable constructor calls and the concrete implementation classes to their respective features by applying the *negative implementation*, i.e., by removing the corresponding source code file and the statement containing the constructor call, and by committing against the negation of the respective feature[4]. In advance to performing these deletions, it is necessary to check-out explicitly a suitable choice where the respective features are deselected.

Table 15.7: Overall commit history of the implementation phase. Based on [SBW16a, Table 5].

R.	Feature Amb.	Changes to `IdentificationMechanism.java`
33	HomeAutomationS.	generated Java source code
34	Ident.Mechanism	multi-variant implementation of identify() (l. 96 – 101)
35	**not** Fp.Scanner	removed FingerprintScanner.java and line 99
36	**not** MagneticCard	removed MagneticCard.java and line 98
37	**not** Keypad	removed MagneticCard.java and line 97

4 Equivalently, we could have applied the positive realization and committed it against positively bound features; however, this would have required three additional code generation runs.

```
95    private void identify() {
96       List<IMechanism> mechs = new LinkedList<>();
97       mechs.add(new Keypad());
98       mechs.add(new MagneticCard());
99       mechs.add(new FingerprintScanner());
100      IMechanism mech = (...) // choose interactively
101      return mech.checkIdentification(getIdentifySignature());
102   }
```

Listing 15.1: Implementation of method `IdentificationMechanism.identify()` in revision 34. Based on [SBW16a, Listing 1].

Responding to Customer Feedback. So far, the SPL has been developed in a phase-structured and proactive way, following the classical development activities analysis, design, and implementation. In advance to a later discussion of the secondary evaluation question **SQ3**, we now investigate to which extent SuperMod and the underlying approach allow to react to a new customer request that cross-cuts all three development activities. The fictional customer feedback consists in a feature request to extend the list of identification mechanisms available by a *biometric* mechanism.

In response to this feedback, we check-out the latest revision of the HAS, choosing the customer's product variant (which includes all sub-features of IdentificationMechanism). Then, we realize the increment in one single iteration under the same feature ambition.

Analysis. A new feature Biometric is introduced to the OR-group below IdentificationMechanism (cf. Figure 15.14). The request does not affect the use cases themselves, but the activity diagram that details the use case Identify (cf. Figure 15.13): We add a new activity BiometricIdentification and connect it to the decision/merge node in analogy to the existing identification actions.

Design. We define a new class Biometric as well as an outgoing realization relationship to the interface IMechanism in the class diagram shown in Figure 15.16. This transparently extends the package diagram's view (cf. Figure 15.15).

Implementation. The (incremental) code generation is re-invoked, creating a new source file Biometric.java. To the implementation of method IdentificationMechanism.identify() (cf. Listing 15.1), we add after line 99: `mechs.add(new Biometric());`

Deployment. The current iteration is finalized by committing all pending changes to the repository under revision 38. In the feature ambition, exclusively the new feature Biometric is selected positively. Last, the same product variant as present in the developer's workspace is deployed to the customer using the EXPORT command.

The implications of this reactive SPLE increment are discussed in Section 15.5.3.

15.3.3 Bootstrapping SuperMod

In a third case study, SuperMod is applied to an instance of a custom metamodel, which in turn is based on Ecore. The considered modeling language reflects an extended subset

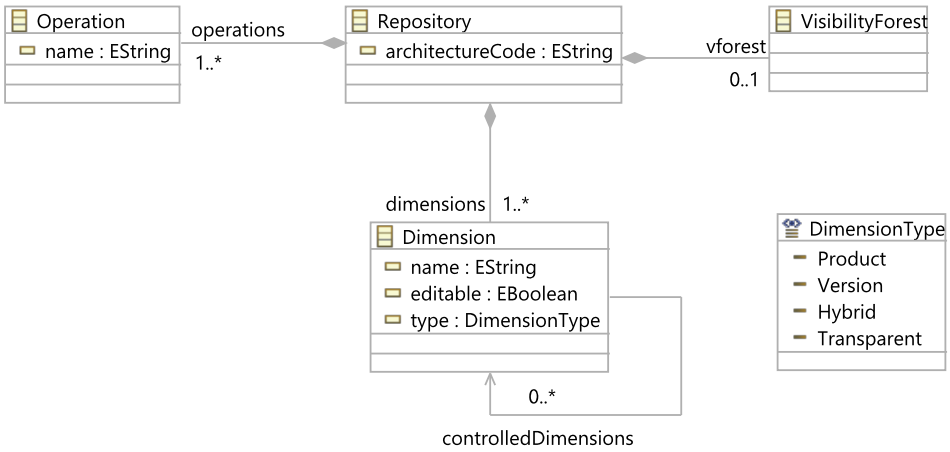


Figure 15.17: The *SuperStrap* metamodel underlying the third evaluation case.

of the conceptual framework's core metamodel introduced in Section 9.2.1; it is capable of describing the architecture provided by as well as the operations offered by a concrete SuperMod repository[5]. The architecture in turn comprises a specific combination of predefined dimensions, which can be version-related (e.g., the revision graph), product-related (e.g., the versioned file system), hybrid (e.g., the feature model), or transparent (e.g., the change space optimization presented in Section 9.6). Overall, the product line shall be capable of reproducing at least the seven repository architectures presented in Section 14.3.

The *SuperStrap* (for *Super*Mod boot*strap*) metamodel is depicted as Ecore class diagram in Figure 15.17. A repository consists of an ordered set of dimensions. These are connected by a cross-reference controlledDimensions, which indicates a versioning relationship. A set of operations represents the commands available to the user.

For editing the singleton instance of the SuperStrap metamodel versioned by SuperMod, a customized generated EMF tree editor is used in this experiment. For better precision, the generated source code has been extended such that the editor relies on UUIDs.

Specific Experimentation Conditions. The focus of this experiment lies on the satisfiability of the approach for *domain-specific languages* (**SQ4**), as well as for *fine-grained variability* (**SQ5**). For the latter reason, we add an intentional obstacle, which is reflected by the attribute architectureCode. Since this attribute is necessary for the distinction of different repository architectures in distributed mode (cf. Section 14.5.4), its value must be unique for every allowed repository architecture mapped by corresponding feature configurations. Conversely, the metamodel defines this attribute as single-valued, such that alternative values cannot be expressed in a single-version (i.e., intrinsic) model instance.

Requirements. Feature model and domain model shall be edited concurrently based on a list of domain model artifacts to be considered in consecutive blocks of iterations:

1. The specific *dimensions* and their relationships.

5 A fully-fledged re-engineering of SuperMod is out of the scope of this modeling language.

2. Pieces of *optimization* controlled by individual features: visibility forest and change space (see Chapter 9); their details are beyond the scope of this modeling language.
3. *Operations* to be offered by concrete variants of the repository. We here confine the list to CHECKOUT, COMMIT, PULL, PUSH, and AMEND (see Section 11.6.4).
4. Finally, the distinct values of the attribute architectureCode.

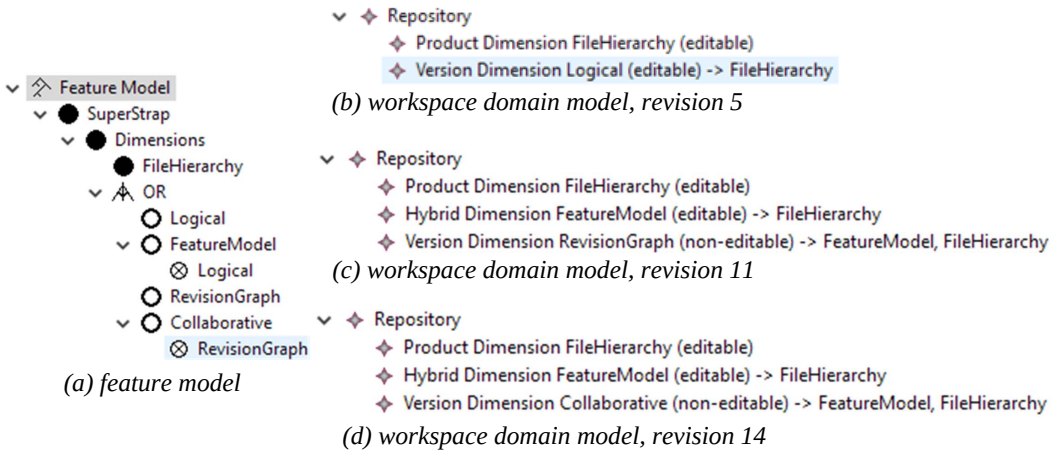
Preliminaries. In an empty Eclipse project, a new model conforming to the SuperStrap metamodel is created, instantiating a new Repository as root object. Then, SuperMod version control support is added based on the *hybrid* (non-collaborative) repository architecture. Besides the model resource, relevant project metadata (text file .project) are added to the list of files under version control. These initial contents are committed as revision 1.

Dimensions and Relationships. In the first phase of the case study, we introduce features for the human-visible dimensions (file hierarchy, low-level logical dimension, feature model, revision graph, and collaborative revision graph). As evident from the revision history shown in Table 15.8, there is a one-to-one correspondence between features and instances of Dimension. Furthermore, the relationships between the dimensions – represented as instances of controlledDimension – depend on combinations of source and target feature.

Due to the mutual exclusions between features Logical and FeatureModel, as well as RevisionGraph and Collab, it is necessary to apply the changes in three different views, such that two explicit check-outs become necessary. In these situations, the operation Migrate is

Table 15.8: Revision history of the first phase of the bootstrapping case. Additional horizontal lines indicate that an explicit check-out was necessary.

R.	Feature Ambition	Change Description
2	FileHierarchy	Features SuperStrap, Dimensions, FileHierarchy; product dimension FileHierarchy
3	Logical	OR-group, feature Logical; version dimension Logical
4	Logical **and** FileH.	link Logical → FileHierarchy (as instance of reference controlledDimensions)
5	**true**	feature FeatureModel (excludes Logical)
6	FeatureModel	hybrid dimension FeatureModel
7	FeatureM. **and** FileH.	FeatureModel → FileHierarchy
8	RevisionGraph	version dimension RevisionGraph
9	Rev.G. **and** Feat.M.	RevisionGraph → FeatureModel
10	RevisionG. **and** FileH.	RevisionGraph → FileHierarchy
11	**true**	feature Collaborative (excludes RevisionGraph)
12	Collaborative	version dimension Collaborative
13	Collab. **and** FeatureM.	Collaborative → FeatureModel
14	Collab. **and** FileH.	Collaborative → FileHierarchy



(a) feature model

(b) workspace domain model, revision 5

(c) workspace domain model, revision 11

(d) workspace domain model, revision 14

Figure 15.18: Workspace artifacts belonging to the first phase of the bootstrapping case.

not applicable since the respective bindings added in the ambition are conflicting with the subsequently intended choice.

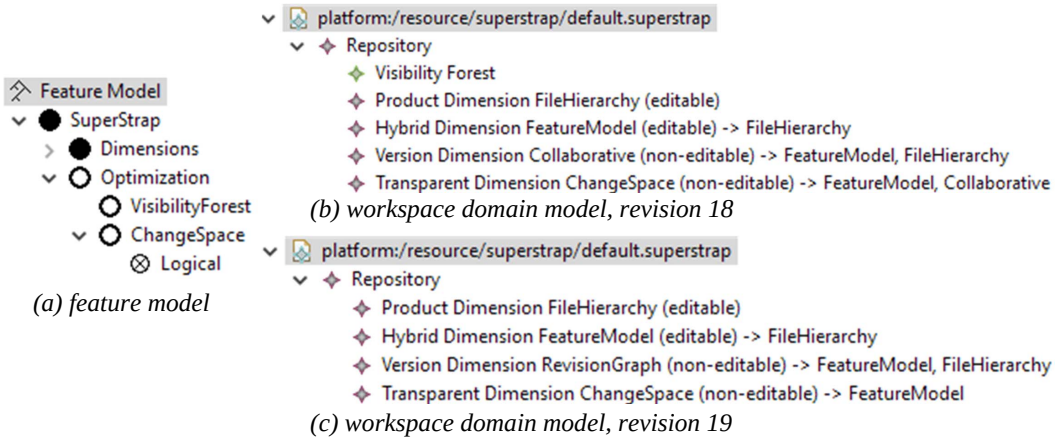
Single-version views on different revisions of the domain model, as well as the final revision of the feature model, are provided in Figure 15.18.

Optimization. The optimizing features suggested by the requirements list are reflected in the domain model in two different ways. First, the *visibility forest* is covered by an instance of a corresponding class attached to the repository (see revision 15 in Table 15.9); this modification involves a successful interactive migration step. Second, the *change space* is represented in the solution space as an additional, transparent dimension that controls all other version dimensions, except for the low-level logical dimension, to which an excludes relationship is defined in the feature model.

The corresponding controlledDimensions links are added under suitable ambitions; this requires to use two different choices in total, such that one additional check-out operation comes into play. We begin in a *collaborative* variant – this feature is still present in the workspace after revision 14 – and switch to a single-user variant (here, feature VisibilityForest

Table 15.9: Revision history for the second phase of the bootstrapping case.

R.	Feature Ambition	Change Description
15	VisibilityForest	Features Optimization, VisibilityForest; object VisibilityForest
16	ChangeSpace	Feature ChangeSpace (excludes Logical); transparent dimension ChangeSpace
17	ChangeS. **and** FeatureM.	link ChangeSpace → FeatureModel
18	ChangeS. **and** Collab.	link ChangeSpace → Collaborative
19	ChangeS. **and** RevisionG.	link ChangeSpace → RevisionGraph



(a) feature model

(b) workspace domain model, revision 18

(c) workspace domain model, revision 19

Figure 15.19: Workspace artifacts for the second phase of the bootstrapping case.

Table 15.10: Revision history for phase three of the bootstrapping case.

R.	Feature Ambition	Change Description
20	SuperStrap	Operations CheckOut, Commit
21	Collaborative	Operations Pull, Push
22	ChangeSpace **and** FeatureModel	Operation Amend

is disabled randomly) in advance to realizing revision 19. Figure 15.19 presents relevant cut-outs of the artifacts modified in the local workspace.

Operations. When compared to the two previous phases, the changes connected to the *operations* offered by the repository are less invasive. Furthermore, the operations are not explicitly modeled as features.

As indicated by Table 15.10, CHECKOUT and COMMIT are offered by all repository architectures available, whereas PULL and PUSH require the existence of a collaborative dimension. Last, the operation AMEND is exclusive for repository architectures that contain both a change space and a feature model as dimensions.

In this specific version history, it is necessary to explicitly check-out by a choice where ChangeSpace is active, since this feature is positively bound in the ambition of revision 22.

Figure 15.20 depicts a product variant with all operations enabled.

Architecture Code. In spite of its rather simple representation in the solution space – the value of a string-valued attribute – the *architecture code* appears to be the most complex detail for multi-variant realization. As listed in the requirements, the value of architectureCode is supposed to match the most adequate identifier of the seven SuperMod architectures presented in Section 14.3.

Since the different values of this attribute are mutually exclusive in the model instance presented in the workspace, seven different ambitions are necessary to match this require-

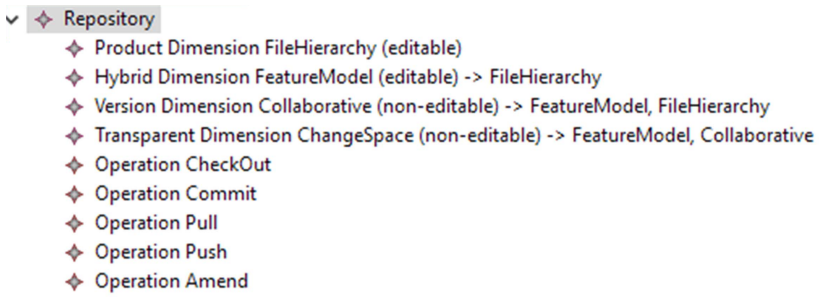


Figure 15.20: Workspace domain model, revision 22, of the bootstrapping case.

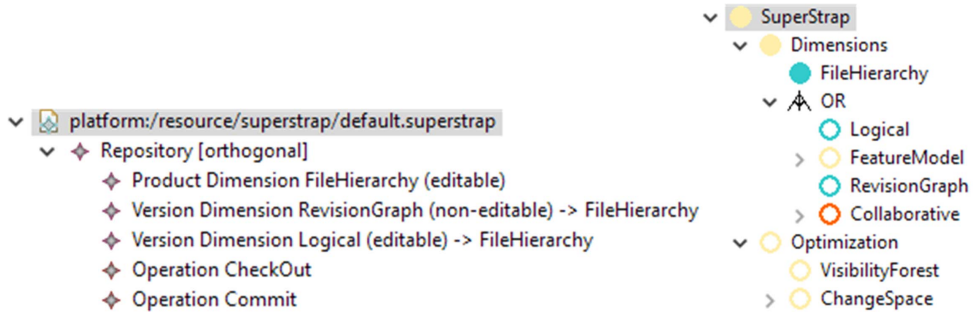
ment. These iterations can be planned in a rather static way; therefore, we use SuperMod's Scope and Check-Out command here (see Section 14.2.5), beginning with revision 24. Table 15.11 summarizes the revision history; six explicit check-outs are applied between the corresponding iterations, provided that we can stay in the view obtained after revision 22 before realizing and committing revision 23.

In the check-out performed in advance to the final revision 29, we are faced with a product conflict: the mutual order of the dimensions RevisionGraph and Logical has not been defined unambiguously during the first phase of the case study (cf. Table 15.8), as the orthogonal architecture does not assume any relationship between these dimensions. The default resolution strategy (which gives priority to the least recent value) resolves the *order conflict* by placing the logical dimension before the revision graph. Albeit, version selection is performed in the historical dimension first, thus we correct this order. Figure 15.21 shows both the revised workspace contents and the ambition used for the final commit.

Notice that the feature model allows for variants that do not represent one of the supported SuperMod repository architectures. For instance, the low-level logical dimension might be combined with a collaborative revision graph. In this case, the corresponding variant of the domain model would not have any value for its architectureCode defined. This could be solved either by defining new values for this attribute and committing them under suitable ambitions, or by introducing new feature model constraints disallowing combinations not supported by the actual SuperMod implementation.

Table 15.11: Architecture codes specified during phase four of the bootstrapping case.

R.	Feature Ambition	Arch. Code
23	FileH. **and** FeatureModel **and** Collaborative	`collabhybrid`
24	FileH. **and** FeatureModel **and** RevisionGraph	`hybrid`
25	FileH. **and** FeatureM. **and not** Collab. **and not** Rev.G.	`feature`
26	FileH. **and not** FeatureModel **and** RevisionGraph	`historical`
27	FileH. **and not** FeatureModel **and** Collaborative	`collaborative`
28	FileH. **and** Logical **and not** Collab. **and not** Rev.G.	`logical`
29	FileH. **and** Logical **and not** Collab. **and** Rev.G.	`orthogonal`



(a) revised workspace domain model *(b) feature ambition*

Figure 15.21: Workspace contents and feature ambition for revision 29.

15.4 Metrics and Results for Primary Questions

In the following, results extracted from the case studies are presented in an aggregated form. The presented data have two origins. First, the contents of the repository – which is transparent to the evaluation subjects who conducted the case studies – have been analyzed in their state at the end of the version history. Second, throughout the entire experimentation, user actions have been logged manually in order to quantify properties referring to the DFE model.

15.4.1 Synoptic Data

In advance to a thorough discussion of the actual metrics and results, we present neutral key figures in Table 15.12. These figures have a synoptic purpose rather than intending to address the primary research questions. For each of the three case studies, the number of (private) revisions and the number of features in the final feature model have been captured. Furthermore, the number of valid feature configurations gives insights into the customizability of the product lines. For quantifying the size of the product space, we use four disjoint figures: the number of versioned files, the numbers of model objects (from the abstract syntax of all versioned model resources) and of concrete syntax elements (i.e., model objects located in GMF diagram resources, or visible and editable tree elements and

Table 15.12: Key figures of the three case studies.

	Graph	HAS	SuperStrap
Revisions	23	38	29
Features	11	16	10
Valid Feature Configurations	56	448	28
Versioned Files	4	24	2
Model Objects	89	197	13
Concrete Syntax Elements	23	97	29
Lines of Text	21	374	11

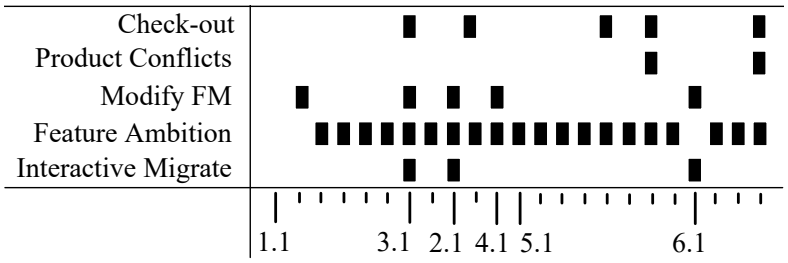


Figure 15.22: History of interactive commands for the *Graph* case study.

properties in the case of the SuperStrap editor), and the lines of text (including both project metadata and versioned source code files).

The manually recorded user interactions serve as a second source of data. The logs obtained for the individual case studies are presented in a condensed form in Figures 15.22, 15.23, and 15.24. The user actions performed (represented by black rectangles) are categorized into five different event types.

1. The operation CHECKOUT is optional in the DFE model; it is invoked either explicitly in case the presented workspace view does not match the evaluation subject's intent, or automatically after a PULL in collaborative mode. In sum, 21 check-outs have been made in all case studies.
2. *Product conflicts* are presented as soon as the user defines a choice that produces them. In three iterations, automatic repair actions are applied and conflict markers are presented to the user. In two of them, the subject revises the default resolution actions.
3. In 20 iterations overall, the *feature model* is *modified* by the subject. All modifications consist in the definition of at least one new feature. Furthermore, feature groups or requires/excludes constraints are defined in 9 distinct iterations.
4. Upon COMMIT, SuperMod requests a *feature ambition* from the user in case the feature model is not empty and changes to the domain model have been applied. When referring to the case studies, for six commit events, no feature ambition had to be defined.
5. The operation MIGRATE is a key element of the DFE model. In general, migration comes into play as soon as new features are added (here, in 20 iterations; see MOD-

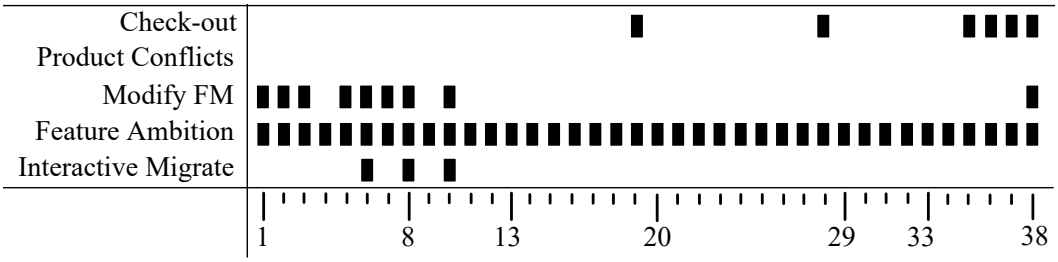


Figure 15.23: Command History of the *HAS* case study.

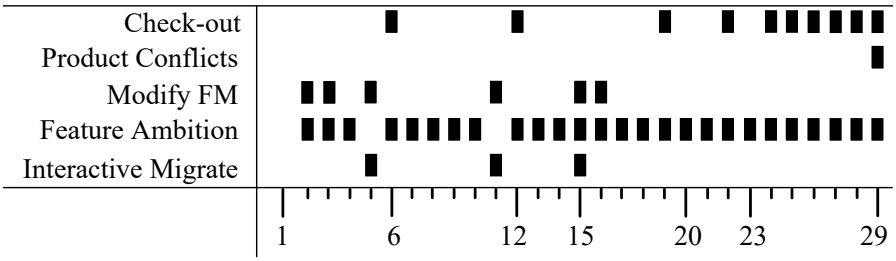


Figure 15.24: Command History of the *SuperStrap* case study.

IFY). From the action log, we can infer that a total of nine migrations required user interaction.

The data aggregated in Table 15.12 and in Figures 15.22 until 15.24 are further decomposed and refined in the subsequent subsections in order to respond to the primary evaluation questions using adequate metrics as introduced informally in Section 15.2.

15.4.2 Reduced Product Editing Complexity over Unfiltered Editing

The first question, **PQ1**, refers to the added value of the filtered editing approach over unfiltered approaches in terms of product editing. As a concrete representative of unfiltered approaches, we assume (not without the loss of generality) an explicitly mapping-based annotative tool (see Section 5.4.2) for comparison. Furthermore, we approximate the number of elements visible and editable by an unfiltered tool as the number of elements available in the transparent repository of our filtered approach.

As a first metric, we define the *number of modified elements* of a specific editing model iteration as the number of elements added, deleted, or modified therein. According to the mechanisms explained in Section 11.3.3, the visibility of each modified element, i.e., each element contained in the *write set* E_{mod}, is modified during commit. For a fair comparison, we confine the write set to the following elements:

- text files or model files;
- graphically or textually visible domain elements (i.e., nodes, connections, or lines of text);
- objects parts of the abstract syntax persisted in addition to graphical diagram resources. In particular, in case a value of a structural feature is added, updated, or deleted, the corresponding object is considered as modified.

Furthermore, we define the *workspace/repository ratio* (W/R ratio) r_{wr} of an editing model iteration as the quotient obtained by the number of *graphical or textual elements* visible in the workspace (i.e., the cardinality of the workspace element set E_w, which does neither include files nor concrete syntax objects), being divided by the number of corresponding repository elements (cardinality of E_r):

$$r_{wr} = \frac{|E_w|}{|E_r|} \tag{15.1}$$

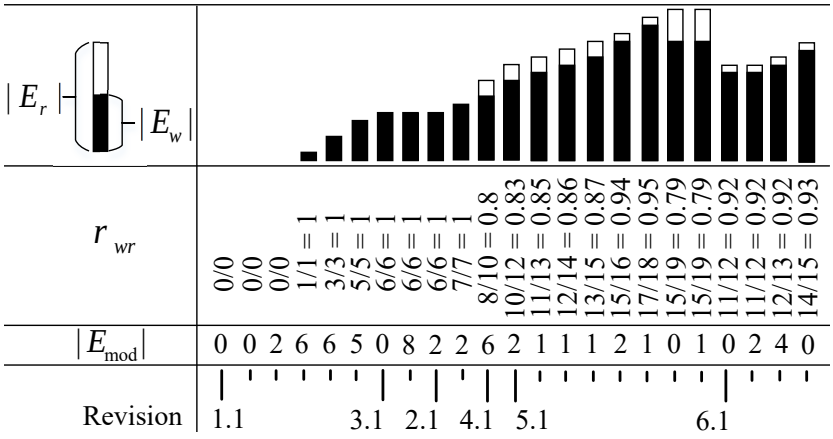


Figure 15.25: Numbers of modified elements and W/R ratios (*Graph* study).

For a fair comparison, both element sets are confined to the model resource(s) and/or text file(s) in which modifications have been applied. Furthermore, in collaborative mode, only those repository elements that are available in the active local repository copy are counted.

In general, the W/R ratio is the complement to the *degree of filtering* r_{df} present in a specific iteration:

$$r_{df} = 1 - r_{wr} \tag{15.2}$$

When applying unfiltered editing with a mapping-based annotative tool, the W/R ratio is constantly 1. This is due to the fact that all elements of the product space are visible and potentially editable. In filtered editing, however, those elements that do not pass the choice are filtered out from the workspace. Furthermore, default conflict resolution actions may automatically remove (but never automatically add) new elements. Therefore, in the here applied filtered editing model, we generally obtain a W/R ratio between (and including) 0 and 1, where low values indicate a high degree of filtering, from which in turn a high reduction of cognitive complexity of product space editing may be deduced.

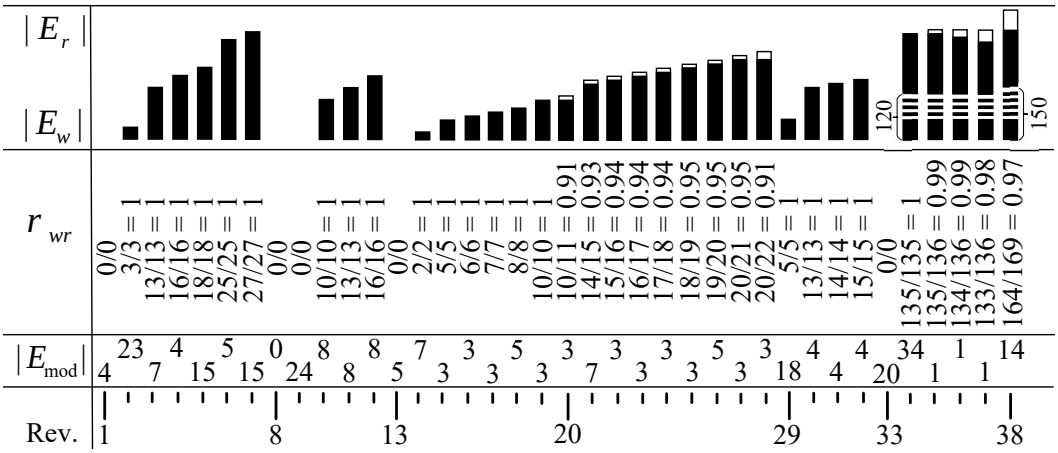


Figure 15.26: Numbers of modified elements and W/R ratios (*Home Automation* study).

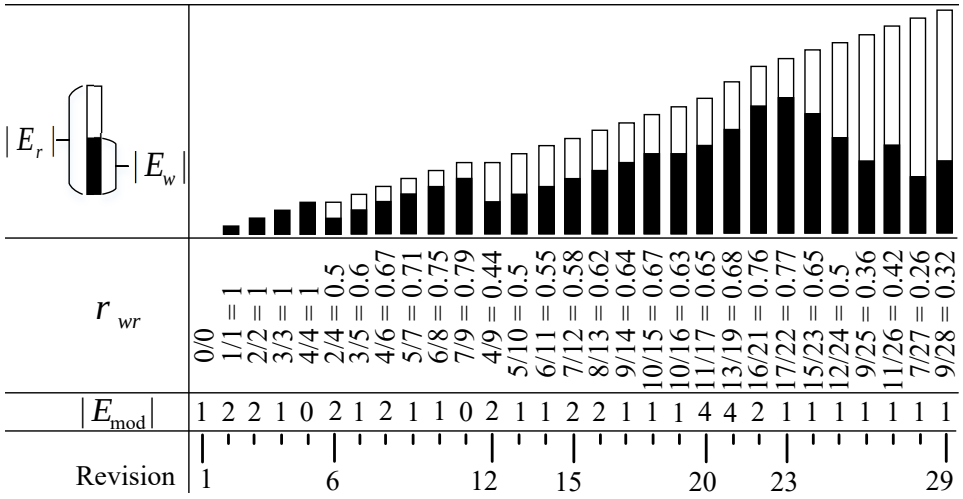


Figure 15.27: Numbers of modified elements and W/R ratios (*SuperStrap* study).

The histograms and quotients shown in Figures 15.25, 15.26, and 15.27 present the numbers of modified elements $|E_{mod}|$ and the W/R ratios r_{wr} obtained for the Graph, HAS, and SuperStrap studies, respectively. Table 15.13 contains aggregate results. We interpret them separately, beginning with the case study having the highest W/R ratio.

HAS. The average number of modified elements of 7.47 confirms that the commits were rather coarse-grained. Moreover, the average W/R ratio of 0.98 indicates almost no reduction of product space complexity when compared to unfiltered editing. This is due to two properties of the HAS study: First, the case study exposes a comparably low amount of alternative variability, which is reflected by only two XOR groups from which a mutual exclusion of domain model elements is derived. Second, the straightforward development strategy followed in the analysis phase was to introduce a new feature and to immediately realize it and therefore bind it positively in the ambition. The transparent choice migration performed after each commit encouraged the evaluation subject to stay in an almost-superimposition view. Taken together, filtered editing was not noticeably exploited here, but conversely, this functionality was not necessary due to the comparably few occurrences of alternative variation.

Graph. When compared to the HAS study, the commits were more fine-grained (2.17 elements modified in average). The minimum and average W/R ratios of 0.79 and 0.918 do not indicate a high impact of filtered editing at product level either. Nevertheless, we can observe an increase in the number of graphical elements filtered out with growing size of the revision history and occurrences of feature interaction, e.g., when considering the constructor parameters added in public revision 5.

SuperStrap. In this case study, the added value of filtered editing for MDSPLE becomes most apparent. Only in the first five revisions, we stayed in a view equivalent to the superimposition. Due to the high number of feature dependencies and occurrences of feature interaction, the W/R ratio becomes as low as 0.26 in revision 28; the average ratio is 0.643. Conversely, the high degree of alternative variability forced

Table 15.13: Aggregated data connected to the number of modified elements and workspace/repository ratios of all case studies.

	Graph	HAS	SuperStrap
Maximum $\lvert E_{mod}\rvert$	8	34	4
Average $\lvert E_{mod}\rvert$	2.17	7.47	1.41
Total $\lvert E_{mod}\rvert$	50	284	41
Maximum $\lvert E_w\rvert$	17	164	17
Average $\lvert E_w\rvert$	8.61	28.32	7.45
Maximum $\lvert E_r\rvert$	18	169	28
Average $\lvert E_r\rvert$	9.65	28.87	13.14
Minimum r_{wr}	0.79	0.91	0.26
Average r_{wr}	0.918	0.980	0.643

the evaluation subjects into comparably fine-grained commits. A maximum of four elements, in average 1.41 per iteration, were modified.

Taken together, the degree of filtering observed in the case studies is measurable but not significant. We expect, but cannot experimentally confirm, that the necessity and benefits of filtered editing will increase, and therefore, the W/R ratios will decrease, when transitioning into the maintenance phase of larger product lines.

15.4.3 Reduced Version Management Effort over Unfiltered Editing

PQ2 refers to the added value over approaches relying on unfiltered editing in terms of reduced version management effort. Like above, we assume an explicitly mapping-based annotative approach for comparison. Furthermore, we proceed under the premise that an SPL developer who uses such a hypothetical approach would create exactly the same feature annotations as transparently produced by SuperMod; in particular, we allege that the hypothetical approach supports hierarchical visibilities as defined in Section 10.2.2.

To be able to estimate the user effort implied by both approaches, we have to put in relation the events of user interaction required to establish the mapping (i.e., traceability links). In the mapping-based approach, we assume that all annotations are created or updated manually by text input. As an initial approximation, we define that the *visibility complexity* c_v, i.e., the effort for creating one text annotation is proportional to the number of elements found in the abstract syntax tree of the corresponding expression Ξ:

$$c_v(\Xi) = \begin{cases} 1 & \text{if } \Xi \text{ is a reference to a feature} \\ 1 + c_v(\Xi_N) & \text{if } \Xi \text{ is the negation of an expression } \Xi_N \\ 1 + c_v(\Xi_L) + c_v(\Xi_R) & \text{if } \Xi \text{ is a logical combination of } \Xi_L \text{ and } \Xi_R \end{cases} \quad \textbf{(15.3)}$$

In SuperMod, the definition of an ambition – which automates the creation of feature annotations – requires a certain number of clicks in the feature ambition dialog. Taking into account that the creation of negative feature bindings requires two clicks onto the

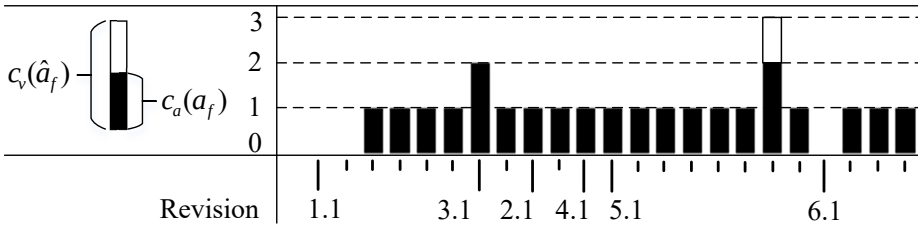


Figure 15.28: Visibility and ambition complexity measured for the *Graph* case study.

corresponding feature, we approximate the *ambition complexity* as the minimum number of clicks necessary for the creation of a feature ambition a_f as follows:

$$c_a(a_f) = \sum_{o_i \in O_f} c_b(o_i, a_f); \quad c_b(o_i, a_f) = \begin{cases} 1 & \text{if } (o_i, true) \in a_f \\ 2 & \text{if } (o_i, false) \in a_f \\ 0 & \text{otherwise} \end{cases} \tag{15.4}$$

The histograms shown in Figures 15.28, 15.29, and 15.30 present the ambition complexity values (and the corresponding visibility complexity values) connected to the feature ambition a_f (and the visibility expression $\hat{a}_f$ derived from it) of every commit performed in the three case studies. In the Graph and in the HAS study, the majority of ambitions have a complexity of 1, which is connected to the fact that these experiments were organized in a rather feature-driven way. In contrast, the SuperStrap case study contains many domain model elements connected to a combination of (positively or negatively selected) features; this is reflected in a much higher average ambition complexity, but also in a higher complexity of the derived visibility expressions. In particular, beginning with the changes related to the value of the attribute architectureCode in revisions 23f., defining a suitable ambition does not only require a certain number of clicks but also additional cognitive capacities. The values presented in the upper two compartments of Table 15.14 aggregate the data visualized in Figure 15.28 until 15.30.

The aggregate results suggest – not surprisingly – that the complexity of a feature ambition is proportional to the complexity of a corresponding visibility derived from it. Nevertheless, the mechanisms of visibility update of the compared approaches differ with respect to one decisive property: Ambition specification is required from the user *once per commit*, while manual visibility management in unfiltered editing would require to update the visibility of *each modified element*. Therefore, when computing the *ambition/visibility quotient* of an editing model iteration, the number of modified elements $|E_{mod}|$ (see above) must be taken

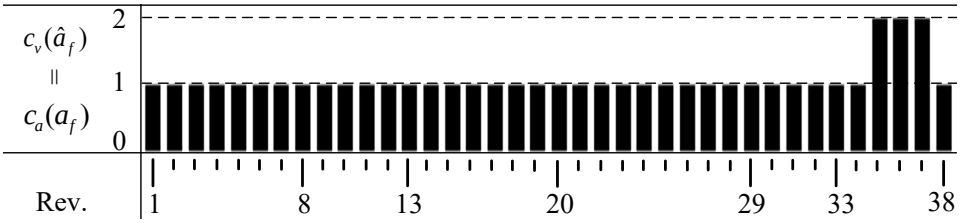


Figure 15.29: Visibility and ambition complexity of the *Home Automation* study.

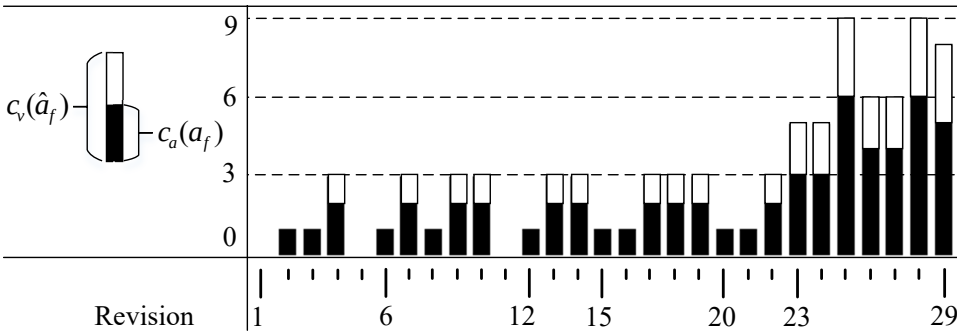


Figure 15.30: Visibility and ambition complexity of the *SuperStrap* case study.

into account in addition:

$$r_{av} = \frac{c_a(a_f)}{c_v(\hat{a}_f) \cdot |E_{mod}|} \tag{15.5}$$

With this definition, we make a rather pessimistic assumption for unfiltered editing, namely that the user repeats the same text input for every updated visibility. A less expensive alternative would be to copy the feature expression corresponding to the ambition to the hypothetical clipboard and to paste it into the visibility of all $|E_{mod}|$ modified elements. Assuming that paste operations imply a complexity of 1, we may define the *alternative ambition/visibility quotient* as:

$$r'_{av} = \frac{c_a(a_f)}{c_v(\hat{a}_f) + |E_{mod}|} \tag{15.6}$$

Some of the SPLE approaches categorized into *temporarily filtered editing* (see Section 9.8) offer a change recording mechanism that actually automates this copy-and-paste strategy.

The results of applying both the default and the alternative metric to the data obtained from the case studies are presented in the bottom compartment of Table 15.14.

In total, the low values for the r_{av} coefficient reflect significant savings of user effort. The complexity of filtered editing lies in between 13% and 68% of unfiltered editing when

Table 15.14: Aggregate complexity values and ambition/visibility quotients.

	Graph	HAS	SuperStrap
Maximum $c_v(\hat{a}_f)$	3	2	9
Average $c_v(\hat{a}_f)$	1.04	1.08	3.00
Total $c_v(\hat{a}_f)$	24	41	87
Maximum $c_a(a_f)$	2	2	6
Average $c_a(a_f)$	1.00	1.08	2.27
Total $c_a(a_f)$	23	41	66
Average r_{av}	0.44	0.13	0.68
Average r'_{av}	0.31	0.13	0.53

applying the first quotient, and between 13% and 53% for the second. The figures also suggest that filtered editing outperforms unfiltered editing especially in the HAS case, but the differences are less in the Graph and in the SuperStrap case.

15.4.4 Impact of the Dynamic Filtered Editing Model

We transition to **PQ3**, which addresses the unobtrusiveness of dynamic filtered editing, compared to the (state-of-the-art) static approach, claimed by goal **G2**. First, emphasis is put on the possibility to introduce features concurrently with their corresponding domain model elements by allowing for feature model editing during MODIFY. Implicitly, we assume alternating domain and feature model editing (see Section 11.6) for the SFE counterpart.

We also quantify the situations in which a second property of the DFE model was exploited: the operation MIGRATE, which promises to reduce the number of explicit CHECKOUT operations necessary, but which may also introduce additional events of user interaction to the DFE model. [6]

From the event log manually recorded during the experimentation, we can reproduce the following metrics that are supposed to quantify the impact of the DFE model:

New Features Bound. The number of iterations in which at least one feature was introduced and (positively or negatively) bound in the ambition used for the corresponding commit, divided by the number of commits performed in the case study.

$$r_{nfb} = \frac{\#\text{iterations} \mid \exists f_i : f_i \in E_{ins} \wedge (f_i, s_i) \in a_f}{\#\text{iterations}},\ s_i \in \{true, false\} \quad (\textbf{15.7})$$

Explicit Check-Outs. The ratio of iterations started with an explicit CHECKOUT operation, indicating that the choice used for the previous iteration was not suitable (after having applied the MIGRATE operation). Updates enforced after a PULL are not considered as explicit check-outs.

$$r_{ech} = \frac{\#\text{iterations} \mid c_i^{ch} \neq c_{i-1}^{mi}}{\#\text{iterations}}, \quad i \text{ is the revision number} \quad (\textbf{15.8})$$

Interactive Migration. The number of iterations in which the MIGRATE operation could not be executed in a deterministic way (such that user interaction became necessary), divided by the overall number of migrations executed.

$$r_{imi} = \frac{\#\text{iterations with interactive migration}}{\#\text{iterations with migration}} \quad (\textbf{15.9})$$

Unsatisfactory Migration. The percentage of iterations in which an interactive MIGRATE was enforced, but due to consistency constraints, the dialog did not allow to define a

6 The here presented data emerge from metrics similar to those employed in a preliminary evaluation of the DFE model in [SW17b, Section 8]. Albeit, the reported values differ for two reasons. First, different experiments are underlying the mentioned Graph case studies. Second, the terms *migration interaction* and *canceled/unsatisfactory migration* have different interpretations. The bootstrapping case study has not been considered in [SW17b].

choice that would have matched the subject's intents for the subsequent iteration.

$$r_{umi} = \frac{\#\text{iterations with interactive migration} \mid c_i^{mi} \neq c_{i+1}^{ch}}{\#\text{iterations with interactive migration}} \quad (\textbf{15.10})$$

In Table 15.15, the results of applying these metrics to the three case studies are listed. From those, we can draw several conclusions about the DFE model as applied in the case studies.

– In 19% of all commits concluded by an interactive feature ambition specification, the user selects a feature that has been introduced in the same iteration. The same ratio of additional iterations would be required in a comparative SFE model, since the definition of a new feature and its realization in the domain model must be separated. We conclude that DFE allows to develop the same product line in a shorter revision history than SFE.

– The deduced average ratio of 23% of explicit check-outs confirms the hypothesis that the DFE model requires less user interaction than the SFE model, where a check-out is needed in advance to 100% of the iterations. Even when taking into consideration the additional effort caused by the new operation MIGRATE (here, $\frac{9}{84} = 11\%$), the required user effort for version definition is less than a third, compared to SFE.

  The bootstrapping case study plays a special role: Explicit check-outs were necessary in advance to the final six iterations. As described in Section 15.3.3, these were executed in SuperMod's lightweight static filtered editing mode; for these individual cases, the DFE model would not expose a significant benefit.

– One potential disadvantage of the DFE model is the additional user interactions implied by the operation MIGRATE. Yet, based on the data obtained from the case studies, we can observe only a slight increase. On the one hand, only about the half of all migrations performed basically required user interaction. On the other hand, when examining in greater detail the user effort effectively required in migration dialogs, the overall result of all case studies is 19 clicks. The histogram presented in Figure 15.31 breaks down the number of clicks (approximated as *migration effort*) performed in all cases of migration.

Table 15.15: Aggregated values quantifying the impact of the DFE model.

	Graph	**HAS**	**SuperStrap**	**Total**
r_{nfb}	$\frac{3}{20} = 15\%$	$\frac{8}{38} = 21\%$	$\frac{5}{26} = 19\%$	$\frac{16}{84} = 19\%$
r_{ech}	$\frac{5}{23} = 22\%$	$\frac{6}{38} = 16\%$	$\frac{10}{29} = 34\%$	$\frac{21}{90} = 23\%$
r_{imi}	$\frac{3}{5} = 60\%$	$\frac{3}{8} = 38\%$	$\frac{3}{6} = 50\%$	$\frac{9}{20} = 45\%$
r_{umi}	$\frac{1}{3} = 33\%$	$\frac{0}{3} = 0\%$	$\frac{2}{3} = 66\%$	$\frac{3}{9} = 33\%$

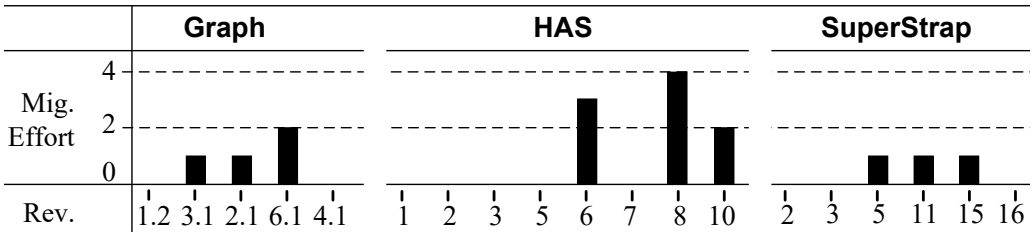


Figure 15.31: User effort required for the operation MIGRATE.

– Also, migration might be counterintuitive in some cases. When the user already knows at commit time that he/she will need a different choice for the subsequently planned change, the migration dialog may constitute a superfluous interactive event—he/she must hit the cancel button (or complete the dialog randomly) and invoke the check-out operation manually. When considering the case studies in total, there were only three such events (likeliness $\frac{3}{90} = 3.3\%$).

For the presented case studies, we may deduce that the DFE model implies considerably more savings in terms of user interaction events than it causes. In 60 of a total of 90 iterations, no user interaction is required at all. Moreover, in 21 iterations, the user effort is as high as in SFE. Last, only 9 additional user interactions are required. Altogether, this suggests that the dynamic filtered editing model behaves significantly less obtrusively than SFE. Due to the heterogeneity of the applied metrics, we cannot indicate a final percentage for quantifying the reduced obtrusiveness.

15.4.5 Performance of A-Posteriori Product-Based Analysis

The fourth primary question, **PQ4**, refers to the performance of the here presented a-posteriori product-based well-formedness analysis approach. In particular, the precision of the default resolution strategy as well as the number of variants affected by the analysis and repair, compared to purely product-based analysis, is of interest.

It has not been a goal of the case studies to provoke an exhaustive sample of product conflicts intentionally; therefore, we can only observe a total of three product well-formedness violations reported in total (see Figures 15.22 until 15.24). To increase the data basis, we take in addition the conflict resolution example presented in Section 13.4.4 under consideration. This contains three extra product conflicts, which are reported and resolved in one single iteration.

For quantifying the performance of the considered analysis strategy in the specific case studies and examples presented, we collected additional data by analyzing the recorded action logs; see upper part of Table 15.16. Two metrics applied below demand for further explanation:

Accuracy of Default Resolution. The ratio of reported conflicts (within one editing model iteration) for which the automatically applied resolution action (depending on the previously selected default resolution strategy) matches the subject's expectations,

Table 15.16: Data connected to the occurrences of product well-formedness conflicts. (*o.c.* is short for order conflict.)

	Graph (5.7)	**Graph (6.4)**	**Graph (ext.)**	**SuperStrap (29)**
Conflicts	1 (o.c.)	1 (o.c.)	3 ([8])	1 (o.c.)
Variants Affected	4	8	1	5
Variants Available	16	56	8	28
r_{dra}	$\frac{1}{1} = 100\%$	$\frac{0}{1} = 0\%$	$\frac{1}{3} = 33\%$	$\frac{0}{1} = 0\%$
r_{rs}	$\frac{4}{16} = 25\%$	$\frac{8}{56} = 14\%$	$\frac{1}{8} = 13\%$	$\frac{5}{28} = 18\%$

such that no manual revision is required.

$$r_{dra} = \frac{\#\text{conflicts} - \#\text{manually revised conflicts}}{\#\text{conflicts}} \tag{15.11}$$

Resolution Scope. The maximum number of variants potentially affected by an individual conflict resolution action. This quantity is obtained by dividing the number of variants described by the feature ambition defined at commit through the total number of variants available at this stage of the case study. Based on this ratio, the logical scope of each (automatically applied and possibly manually revised) resolution action can be quantified. [7]

$$r_{rs} = \frac{\#\text{variants affected}}{\#\text{variants available}} \tag{15.12}$$

The lower part of Table 15.16 lists values for the two metrics for each revision in which product well-formedness violations occurred. Column *Graph (ext.)* refers to the additional example from Section 13.4.4. From the presented values, we may conclude:

– Concerning the accuracy of default resolution, the presented values do not allow for a general statement; after all, different resolution strategies were applied in the case studies. In a total of four out of six cases, the subjects had to manually revert the resolution actions in the specific cases considered here.

– The resolution scope values can be interpreted as follows: In one out of four cases, the corrections were local, e.g., confined to the single variant presented in the workspace. In the remaining three cases, a considerable subset of the available variants was repaired indirectly by performing conflict resolution actions in a single-variant view. Altogether, the repair

7 Due to its filtered nature, the product validation approach may guarantee neither that the same conflict would occur in all variants included in the ambition, nor that the conflict resolution actions lead to a syntactically correct result in all affected variants. The only statement that can be made is: If the same conflict (with the same parameters) occurs in another variant included in the ambition, the performed resolution action will be applied in the same way unless it is reverted by another conflict resolution decision.

8 order conflict, single-valued feature value conflict, (feature) display name conflict

actions applied transparently in the repository have the qualities of *sample-based* analysis, although they are actually applied in a single-version context, i.e., in a *product-based* way.

It remains to be mentioned that one of the conflicts contained in the data set constitutes a special case: the feature display name conflict in *Graph (ext.)* refers to the feature model rather than to the domain model; it is therefore independent from the feature ambition. Accordingly, for this specific conflict, we may assume that all 8 variants are affected, resulting in an effective resolution scope of 100%.

15.4.6 Threats to Validity

The properties concluded from the results presented here must be understood with the following potential restrictions towards generality and validity in mind:

– The performed case studies were of academic scale and synthetic. Therefore, the numerical values presented in this section cannot serve as expectation frame for real-world applications. Furthermore, the case studies have dealt with the software development activities analysis, design, and (to a limited extent) implementation. A transfer of the positive evaluation results to the feasibility of the approach with *testing* and *maintenance* is disallowed.

– In the Graph and in the SuperStrap case, we performed the experimentation based on a fictional requirements list, where the product line was planned in advance in a coarse-grained way. Furthermore, in the Graph case, the expected result (also in the product space) was known to the evaluation subject. This is in contrast to realistic SPL engineering problems, where the requirements must be acquired manually and where the expected result is not known. To compensate for this, intentional obstacles have been scattered over the requirements lists, but it is questionable that they have the quality of real-world obstacles such as unclear requirements, or unforeseen feature interaction.

– Most of the analyses referring to the editing model rely on a manual log of user interactions. In general, when conducting the case studies, the evaluation subjects attempted to organize the subsequent iterations in such a way that as few revisions as possible have to be performed. Nevertheless, the revision histories also contain superfluous revisions, which could have been avoided by more thorough planning in advance. For instance, revision 3.1 of the Graph study and revision 4 of the HAS study could have been eliminated. We can neither assume that real-world users would stick to the same strategy (rather than organizing the history into more revisions), nor that more than the here reported superfluous revisions would be produced by less experienced users. Therefore, all reported quantities that emerge from the number of revisions are potentially affected by a bias.

– In Section 15.4.3, we assumed a hypothetical explicitly mapping-based approach as “gold standard”; compared to this, we have observed that the indirect specification of version information via ambitions requires a lower user effort than manual editing of presence conditions. The underlying assumption might be problematic inasmuch as we have defined two different metrics for the complexity of visibilities and ambitions, respectively. These metrics cannot be compared without bias; e.g., the length of feature names is ignored.

– The number of user interactions need not necessarily be proportional to the subjectively observed level of obtrusiveness of user interactions of both filtered editing models compared

in Section 15.4.4. Several factors such as the point in time when the dialogs were shown or the duration of the interaction have not been considered.

– For quantifying the manual effort for ambition specification as well as for interactive migration, we used the minimum number of clicks necessary to complete the presented feature configuration as metric. These do not necessarily reflect the actual number of clicks applied.

– Concerning the performance of a-posteriori product-based well-formedness analysis presented in Section 15.4.5, only a small subset of conflicts (belonging to four out of 16 conflict types described in Section 13.3) were observed in the case studies. To obtain reliable results for the performance of default resolution, a larger case study with an exhaustive list of conflicts would be required. Nevertheless, the reported values for the *resolution scope* are independent of the concrete conflict type.

15.5 Qualitative Discussion of Secondary Questions

From the conducted case studies, we can also draw several conclusions that cannot be directly expressed by numerical values obtained from metrics. The corresponding observations, motivated by the secondary evaluation questions **SQ1** until **SQ5** at the beginning, are discussed in a qualitative way below.

15.5.1 Properties of Distributed Collaborative Versioning

The distributed multi-user versioning model, realized by explicit synchronization events triggered by PULL and PUSH, was applied in the Graph case study as suggested by **SQ1**.

In this case, the orchestration of changes was almost unproblematic since the concurrently developed modifications addressed disjoint features. We expect the number of reported product conflicts to grow rapidly as soon as the same feature (or same combination thereof) is addressed in ambitions of concurrent private transactions.

Furthermore, the explicit synchronization strategy requires discipline to pull frequently. Otherwise, synchronization problems are delayed until push, where the pull operation would be enforced (see public revisions 2 and 6 in the case study).

Supported by the product-based analysis and repair approach, resolving multi-variant merge conflicts is as easy or as difficult as in single-system development. It may, however, turn out to be a problem that the resolution of conflicts not affecting the workspace view is delayed until an ill-formed product variant is checked-out. In this specific case study, we have not observed any "overlooked" product conflict caused by multi-user operation.

15.5.2 Suitability for Heterogeneous Projects with Generated Code

SuperMod and its underlying conceptual framework primarily target model-driven projects. The HAS case study has successfully demonstrated that SuperMod is capable of versioning realistic model-driven product lines, which consist of both models and source code, the former of which are connected by cross-resource links (see **SQ2**).

Albeit, the HAS case study has also demonstrated that the combination of filtered SPLE and *model transformations* (see Section 3.7) – here, generation of Java source code – causes a new type of problem, which we here denote as the *filter/transform dilemma* [SBW16b]: as soon as a model transformation is invoked, all (newly) produced elements obtain the same visibility (through the ambition), although the corresponding elements in the source model may carry different (and more adequate) visibilities.

In the HAS case, the initial code generation step was invoked from a variant that almost corresponds to the multi-variant class diagram (except for the realization of the features of a XOR group, i.e., Active/Passive and Automatic/Manual; here, one feature was chosen randomly for each group). We were able to derive a large portion of the corresponding multi-variant source code, but all generated files (e.g., the classes reflecting more specific features, such as PassiveLock or AcousticSignal) obtained the same visibility: $r_{33} \wedge f_{HAS}$.

To overcome the problem of incorrect visibilities, beginning with iteration 34, we removed those parts of the generated code that are specific to a feature, and committed the change against an ambition consisting in a negative reference to the feature. Corresponding actions would be necessary for the implementation of all further features. An alternative to this "double negation" strategy would be to exploit incremental code generation by activating one feature after another, re-generating code, and committing against an ambition where the feature is positively selected. This, however, would require many additional explicit check-outs.

We cannot take this as a general solution. In contrast, *multi-variant model transformations* (MVMT) [SBW16b] might address the problem correctly and reliably. In Section 16.3.3, we sketch current obstacles to the application of MVMT in the SuperMod context.

15.5.3 Feasibility of Reactive SPLE

In **SQ3**, the feasibility of the conceptual approach underlying SuperMod with *reactive* adoption paths to SPLE has been issued. The larger part of the HAS study, namely revisions 1 until 37, have been developed in a *proactive* style by carefully selecting the features during domain analysis, and further refining them during design and implementation. For the technical realization, fine-grained commits have been employed. Revision 38 reflects a fictional customer request to extend the scope of the product line by a new feature, whose analysis, design, and implementation are committed in one iteration *reactively*.

Figure 15.32 aligns the specific process phases of the HAS study with features, making obvious that the principle underlying SuperMod's editing model, namely the feature-by-feature development controlled by feature ambition, is independent of the (proactive versus reactive) SPLE style.

The case study has demonstrated that our approach is compatible with both a proactive and a reactive adoption path to SPLE, allowing for both phase-structured and feature-driven domain engineering processes—this distinction is further generalized in Section 16.3.2. Revision 38 has also demonstrated that SuperMod allows to overcome the *duplicate maintenance* problem (see Section 7.1.6): actually, the request is handled in a representative product variant, but through commit, the changes are automatically propagated to all relevant variants.

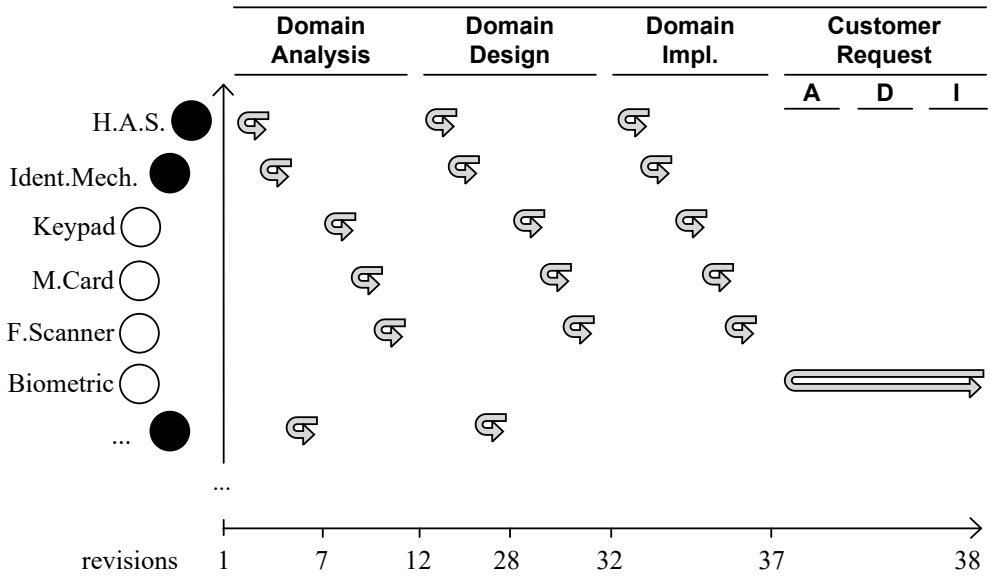


Figure 15.32: Transitioning from proactive to reactive SPLE in the HAS study. Based on [SBW16a, Figure 12].

15.5.4 Compatibility with Domain-Specific Modeling Languages

In two out of three case studies, we have shown that the approach is feasible with the general purpose modeling language UML. Both the Graph and the HAS case include a UML class diagram; in the HAS study, in addition, a package diagram, a use case diagram, and an activity diagram were managed.

Being based on EMF's reflective metadata mechanism, SuperMod provides support for arbitrary Ecore-compliant modeling languages (including Ecore itself) and is therefore suitable for domain-specific modeling languages (DSMLs). Guided by **SQ4**, in the SuperStrap case study, an instance of a user-defined metamodel was versioned successfully.

Concerning the support for different types of concrete syntax, we have learned that the approach supports both EMF tree syntax and GMF-based graphical syntax out of the box. In the latter case, diagrams are technically interpreted as regular model instances. In the case studies, we have maintained discipline in not connecting layout-related changes to specific logical ambitions; however, in general, we cannot exclude the possibility of pseudo conflicts concerning, e.g., absolute positioning of elements on the canvas.

Especially in the HAS study, text-based resources – here, Java source code – were managed in a line-oriented way based on SuperMod's extrinsic text metamodel. In general, line-oriented versioning can also be transferred to textual DSMLs. When using Xtext-based languages, however, the contents are interpreted as models (see Section 3.6). Therefore, it is the abstract syntax tree that would be versioned by SuperMod extrinsically in the repository. This was not practically applied by any of the case studies.

15.5.5 Impact of Fine-Grained Product Space Organization

The last of the secondary questions, **SQ5**, refers to the properties of the *fine-grained versioning* strategy followed by the conceptual framework. The case study to answer this question is the bootstrapping experiment described in Section 15.3.3, particularly the last phase, when seven disjoint values for the single-valued attribute architectureCode have been defined, each under an individual feature ambition.

The fact that SuperMod allows to correctly map the values to the intended logical visibility demonstrates the advantage of two design decisions applied, namely *hierarchical product space organization* (see **D18** on page 139) and *extrinsic product space representation in the repository* (**D6.1**, page 136). Taken together, they enable the existence of multiple alternative values for the attribute architectureCode. The requirement that the values must be *disjoint* is in addition fulfilled by product validation (see revision 29).

Taken together, fine-grained product space organization allows for *unconstrained variability*, which has been repeatedly claimed as one of the greatest advantages of the fully filtered editing model over unfiltered annotative SPLE approaches. Due to the distinction between extrinsic (repository-internal) and intrinsic (workspace-internal) representation, the compatibility of product variants with *existing editing tools* is not limited.

15.6 Summary

We have applied the tool SuperMod (see Chapter 14) to three different case studies in order to obtain insights about the properties of the tool itself and of the underlying conceptual framework contributed in Part IV.

The considered case studies include an extended version of the running example of a *Graph Library*; its product space is represented as a UML class diagram, which is collaboratively developed by two fictional users, Alice and Bob. In the second case study, a product line for *Home Automation Systems* (HAS), four different UML diagram types came into consideration; furthermore, we examined the management of the Java source code generated from the static model as well as the handling of a customer request. Last, in a bootstrapping case study, we defined a domain-specific modeling language (*SuperStrap*), an instance of which was versioned by SuperMod; when compared to the previous two cases, this study exposes a finer object granularity with respect to the versioned model details.

We analyzed the final repository contents and a manually recorded user action log by the application of several metrics. The positive answers to the *primary research questions* asked beforehand confirm beneficial properties of the presented approach:

PQ1. When compared to unfiltered editing, the here applied filtered editing approach slightly reduces the complexity of product editing in the workspace. The desired and actually applied degree of filtering depends on the modeling language as well as on the amount of alternative variation in the product space.

PQ2. The presented filtered editing model requires only a small proportion of the user effort for version management, when compared to unfiltered editing. The low values obtained for the *ambition complexity* confirm that the concept *feature ambition* is a useful and not too complicated indirection to automatically create or modify

the traceability links (visibilities) of product space elements affected by a specific modification.

PQ3. Next, we have compared the here presented *dynamic* filtered editing (DFE) approach to the so-considered standard, *static* filtered editing (SFE). The results suggest that the newly introduced operation MIGRATE obviates repeated check-outs in many cases. Allowing to use newly introduced features in the ambition slightly reduces the number of iterations necessary. Moreover, the extra user effort caused by additional interactions issued by the DFE model (e.g., when migration produces unsatisfactory results) can be neglected.

PQ4. The contributed a-posteriori product-based well-formedness analysis strategy is as easy to apply as any purely product-based strategy; though, due to the ambition mechanism, the applied repair actions affect a comparably larger set of versions, which is reflected by a noticeable *resolution scope*. The approach has similar qualities as sample-based product line analysis.

Altogether, the positive answers indicate that the goals have been fulfilled in the case studies.

G1. The effort of SPL editing is lower when compared to unfiltered approaches based on annotative variability.

G2. The dynamic editing model is less obtrusive than its static counterpart.

G3. A-posteriori product based analysis slightly increases the well-formedness of products.

From the experience made throughout the experimentation, we can qualitatively deduce several *secondary* properties of the approach.

SQ1. Collaborative MDSPL editing appears to be unproblematic as long as the users agree on disjoint ambitions in order to avoid conflicts. Concurrent modifications are merged in a single-version view.

SQ2. SuperMod readily supports heterogeneous projects, where interconnected models and source code are mixed. Yet, as soon as the approach is combined with model transformations, the developer is faced with the *filter/transform dilemma*.

SQ3. The conceptual approach allows for both proactive and reactive SPLE, and in particular, it encourages a transition from the proactive to the reactive style.

SQ4. SuperMod is applicable to text files as well as to tree-based and diagram-based model files; its suitability for textual modeling languages remains to be examined.

SQ5. By relying on a fine-grained version strategy, the approach supports unconstrained variability, while not negatively affecting the compatibility of single-version models with external modeling tools.

The next chapter concludes the thesis and critically reflects, among others, the evaluation. A large share of the future work suggested builds on the results gained in this chapter.

Part VI

Reflections

Look at me: still talking
when there's science to do!

JONATHAN COULTON (2007)

Chapter 16

Conclusions and Outlook

Abstract

The thesis is concluded by a summary of its contents and contributions as well as by a discussion of the the added value promised by the presented scientific work. The benefits, which emerge from particular design decisions underlying the framework, go hand in hand with conceptual limitations, whose consequences are reflected here. In a retrospective discussion, we critically visit the tool's barriers to entry, the framework's relation to SPLE processes, and the connection to multi-variant model transformations. We also identify potential topics for future work in terms of technical extensions of SuperMod. Alongside of final remarks, the relevance of the presented approach for research and for software engineering practitioners is reflected.

Contents

16.1 Achievements

In the following, the thesis is summed up from a retrospective point of view, before the contributions and the implied benefits are made explicit.

16.1.1 Summary

Abstraction, evolution, and variability are three phenomena the stakeholders of almost every software engineering project are faced with. Traditionally, abstraction is achieved through the application of model-driven engineering tools, evolution is addressed by version control systems, and variability is either organized in an ad-hoc way (e.g., by branching or conditional compilation), or explicitly addressed by means of software product line engineering approaches. Traditionally, developers must use three different tools to meet the peculiarities of all three phenomena.

With SuperMod, we have presented "one tool to rule them all": an integrated solution that obviates the need for context switches between tools for historical and logical (model) version management. Though, tools are ephemeral—therefore, the focus of this thesis has been put on the underlying conceptual framework. It builds on top of the Uniform Version Model (UVM), which utilizes formalisms as simple as propositional logic and set theory; furthermore, we have utilized Ecore metamodels – which are compatible with the OMG MOF standard and therefore promise to be long-lasting – for the static modeling of the approach. The elements of the framework are connected by means of a formal editing model, whose interactive operations have been defined as algorithms.

SuperMod encourages the development of a model-driven software product line in an iterative way, organized by a sequence of development increments, which are controlled by metaphors common in version control. Version selections, having a historical and a logical component, are performed in a revision graph and in a feature model, respectively. For the development of product lines, the framework relies on a *filtered editing* strategy: One version – being described by the *choice*, consisting of a selection in the *revision graph* and of the definition of a *feature configuration* – is edited representatively in a local *workspace*. The modifications are written back to the repository automatically, and the *visibilities* (which reflect the SPLE concepts of presence conditions, traceability links, or feature annotations) of versioned elements are updated in such a way that they are visible in future revisions of those variants included in a so called *feature ambition*, a partial selection in the feature model delineating the logical scope of a change. After each iteration, the defined choice is *migrated* – if necessary, interactively – in order to immediately prepare the subsequent revision without the need of manually starting the next iteration.

When speaking in SPLE terms, SuperMod allows developers to "perform application engineering, and get domain engineering for free", following a *product-based product line development* paradigm. The analogous strategy, *product-based product-line analysis* is applied for the detection and resolution of *well-formedness violations*. These may emerge either from unintended feature interaction or from conflicting modifications applied by different developers concurrently. Collaborative MDSPLE, in turn, is offered by a distributed replication strategy organized by the metaphors *pull* and *push*.

Here, the terms "model-driven" and "product line" should by no means be understood

as restrictions: the approach and tool are applicable to non-model resources just as well – concretely, entire Eclipse projects can be versioned – and the usage of a feature model is optional. The tool may even be applied without historical or collaborative versioning in case this is intended to be covered by an external VCS.

Taken together, in this thesis, we have shown that an integrated approach combining MDSE, SPLE, and VC is feasible, that it brings significant advantages over an off-the-shelf tool combination, and that it does not add noticeable cognitive obstacles.

16.1.2 Contributions Made Explicit

Below, we revisit the chapters of this thesis and summarize the theoretical and practical contributions presented therein. These include:

- A *conceptual overview* of the software engineering disciplines *model-driven software engineering* (MDSE), *version control* (VC), and *software product line engineering* (SPLE). See Chapters 3 until 5.
- A *literature review* of state-of-the-art tools addressing the *integrating disciplines* MDSPLE (*model-driven software product line engineering*), MVC (*model version control*), and SPLVC (*software product line version control*) in Chapter 6.
- The exploration of the *requirements* and of the *design choices* of an integrated system towards combined MDSE, SPLE, and VC. (Chapters 2 and 7)
- The design and the analysis of a technology-independent integrating *conceptual framework*. It relies on *EMF models*, *feature models*, and *revision graphs* as user abstractions, on the UVM as formal foundation, and on Ecore-compliant *metamodels* as design formalism. The framework consists of the following components:
  - A *hybrid version model* for the integration of historical and logical versioning. Extensional versioning is realized on top of intensional versioning. (Chapter 9)
  - An *extrinsic product model* providing for unconstrained variability of versioned file systems (containing EMF and non-model resources) and feature models. (Chapter 10)
  - A *consistency-preserving dynamic filtered editing model* that transparently manages the connection between the version space and the product space. See Chapter 11.
  - The framework's extension toward *collaborative versioning*, realized by two-level revision graphs and a distributed replication strategy, both shown in Chapter 12.
  - An *a-posteriori product-based well-formedness analysis* approach that relies on *default resolution* actions, which can be revised manually; repair actions affect a larger scope of versions. See Chapter 13.
- SuperMod, a model-driven *implementation* of the conceptual framework. (Chapter 14)
- An *evaluation* of the user-relevant properties of the approach based on three case studies to which the tool SuperMod has been applied; see Chapter 15. The results confirm the benefits of the approach (see below).

The remaining chapters were connected to introductory purposes (Chapter 1) and to the definition of formal foundations (Chapter 8), respectively.

16.1.3 Benefits Re-Explained

In Section 2.8, we have claimed four key benefits of an integrated conceptual framework over state-of-the-art solutions. After having presented the conceptual and technical solution as well as its evaluation, we revisit this list, assigning concrete design decisions and consequences to the respective benefits.

B1. Uniform Versioning. Externally, a revision graph and a feature model are presented to the user, suggesting two orthogonal version management mechanisms. Internally, however, the framework relies on a hybrid repository architecture based on mappings to concepts inherited from the UVM [WMC01]: options, version rules, choices, and ambitions. Although these internals are never exposed to the user, he/she can profit from their uniformity. For instance, the operation COMMIT allows to map a historical increment added in a well-defined transaction to a specific set of logical versions, blurring the distinction between revisions and variants for the user's sake.

B2. Reduction of Cognitive Complexity. The approach and tool relieve the SPL developer from two complex tasks: the management of multi-variant solution artifacts and the definition of traceability links between problem space and solution space. The product variant available in the workspace contains the realization of only those features that are relevant to the intended change. Through commit, the transparent repository contents are updated automatically. In the evaluation, we have confirmed that through the ambition mechanism, the effort for version membership management is considerably lower when compared to the manual creation of traceability links.

B3. Unconstrained Variability. Although the workspace – by intention – contains only single-version models, the transparent multi-version repository relieves the managed artifacts from single-version consistency rules impeding alternative variation. The extrinsic product model obviates the need for explicit, language-based (e.g., aspects) or tool-based (e.g., model transformations) mechanisms for the definition of variation points. As confirmed by the evaluation, this benefit becomes particularly important when a fine object granularity is demanded in the product space.

B4. Tool Independence. Since the comparison-based versioning strategy does not require change logs, SuperMod does by no means prescribe the usage of a custom tool, nor a specific class of tools, to be used for modifications in the workspace, inside which a standardized intrinsic model representation is employed. Technically supported are all (model or non-model) editors that integrate with Eclipse.

16.2 Limitations

The positive properties listed above ensue from particular design decisions made in advance to the elaboration of the conceptual framework; see Section 7.3. Nevertheless, for a fair résumé, we have to consider also the other sides of the coins. The benefits are inadvertently connected to particular drawbacks, which are detailed in the following. Figure 16.1 illustrates how design decisions, benefits, and limitations are connected to each other. [1]

1 This subsection shares material with [Schwä+15, Section 6].

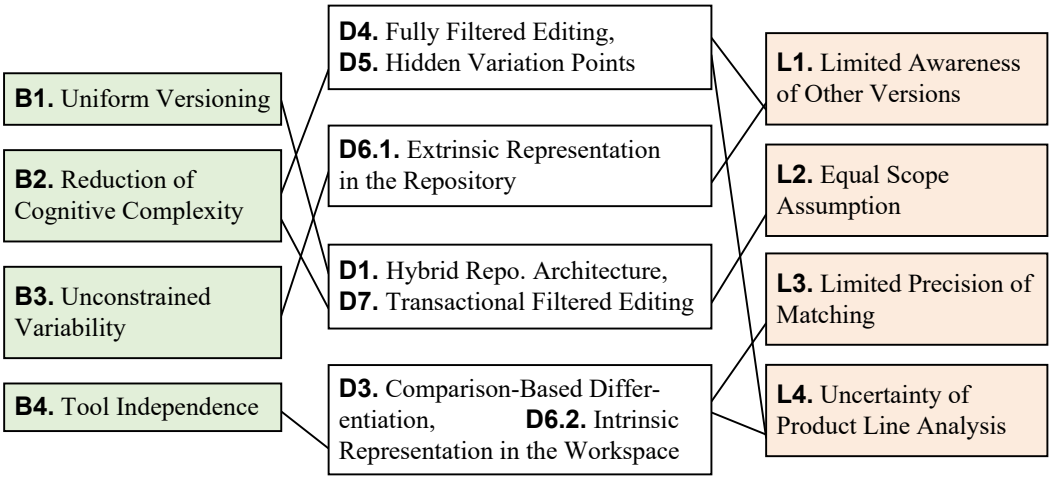


Figure 16.1: Connection between benefits and limitations of the conceptual approach.

L1. Limited Awareness of Other Versions. A key advantage of filtered SPLE is reduced complexity in the sense that artifacts not relevant for a specific change are hidden from the single-version workspace. On the downside, filtered editing also reduces *awareness* of other versions being composed of elements that are not visible under the current choice and that are inaccessible in the transparent repository. Furthermore, version membership information, which is encoded in the visibilities of versioned elements, is faded out. The unawareness of hidden product space artifacts may complicate multi-variant design decisions and lead to maintenance problems such as doubly introduced domain model elements, heterogeneous realization of variants belonging to the same variation point. In general, the advantage of reduced complexity is instantaneously linked to the shortcoming of limited awareness.

L2. Equal Scope Assumption. In the hybrid and transactional editing model presented here, all modifications performed to the workspace domain model within an edit session are written back to the repository at commit time under one common feature ambition. Therefore, it is implicitly assumed that these modifications have the same logical scope. As mentioned before, this forces developers into rather fine-grained commits; however, we cannot exclude the possibility of developers accidentally realizing multiple features (or combinations thereof) in a session. In such a situation, the iteration must be reverted in order to avoid incorrect visibilities being created at commit. The *equal scope assumption* may cause frustration among users.

L3. Limited Precision of Matching. The advantage of tool independence is due to the comparison-based differentiation strategy. On the downside, refraining from employing a change recording mechanism also reduces the precision of matching, which is performed transparently at commit. Due to the usage of UUIDs, this limitation has not negatively affected the case studies presented in the evaluation, but it may become problematic for resource types that do not support UUIDs (e.g., textually persisted Xtext models).

L4. Uncertainty of Product Line Analysis. A-posteriori product-based analysis can guarantee the syntactical well-formedness of no other than one product variant: the contents available in the workspace. After commit, the applied repair actions prevent the same conflict from being reported in other variants, but there is no guaranty for the absence of related conflicts, and furthermore, for the syntactical correctness of other variants transparently managed in the repository.

Another potential drawback of the strategy, which relies on filtered product reconciliation, is that product conflicts not relevant for the current workspace view may be *overlooked*, particularly when emerging from collaboration.

Altogether, the benefit of being able to apply single-product validation techniques in a single-version workspace view is paid with uncertainty related to the well-formedness of product elements not visible in the workspace.

16.3 Retrospective Discussions

Several topics are situated rather at the edge of the scope of this thesis. They have been repeatedly referred to, but the explanations provided in previous chapters did not give satisfying answers to related questions. Here, we provide brief retrospective discussions.

After a general discussion of the *barriers to entry* implied by the presented approach, we reflect the topics of *SPL development processes*, as well as the connection of the presented filtered editing model with *model transformations*.

16.3.1 Barriers to Entry

The HAS case study (see Section 15.3.2) has been conducted by a master student after a training session, in which an earlier version of the Graph Library case study has been demonstrated. We interpret the most important issues that have been communicated in a feedback session:

- For users familiar with the concepts of revision control (i.e., the operations check-out and commit, revision graphs, and commit messages) as well as of SPLE (i.e., feature models, feature configurations, and automatic product derivation), the only new concepts that must be studied are *feature ambitions* – probably the most difficult part of the editing model – and interactive *choice migration*—which, conceptually, is not much different from completing a partial feature configuration.
- It requires discipline to keep the logical scope within an iteration equal, i.e., not to mingle logically disjoint changes, which actually require different ambitions, in the same commit. When compared to conventional version control, this causes much more frequent commits.
- By intention, the here contributed conceptual framework blurs domain engineering and application engineering; the former is actually derived from the latter. Users familiar with annotative variability may be tempted to keep a multi-variant domain model in the workspace, which impedes the definition of variation points on product level.

All in all, this suggests that the training effort required for a tool like SuperMod is not higher when compared to explicitly mapping-based approaches (where the mechanisms for

assigning feature annotations must be taught) and to tools relying on transformational or compositional variability (where new composition languages or extensions to the domain language must be taught).

16.3.2 Agile Filtered SPLE Processes

According to the state-of-the-art exploration provided in Section 5.3, traditional SPL development processes are tailored towards *proactive* SPL adoption paths and follow a rather plan-driven paradigm. In general, software engineering processes range between *plan-driven* and *agile*—an analogous classification should be made also for the special case of SPLE.

As motivated in Section 7.1.1, the necessity to quickly respond to customer feedback and to adapt to changing requirements also raises arguments for an *agile* style of SPLE, which is still subject to research [TC06; GM08]. Agile principles are also desirable in *reactive SPLE* [Ap+13b], given the fact that features can be added to the product line at any time after products have been derived.

The separation of development activities into domain engineering (DE) and application engineering (AE) is independent of the process paradigm. Let us consider DE first: In the HAS case study, we have observed a transition from *phase-structured* to *feature-driven* domain engineering (DE); see below. Both paradigms are supported by SuperMod, but the feature-driven style is assumed as the default.

Phase-Structured Domain Engineering. Here, DE is performed in a sequential way as shown in Figure 16.2. In the beginning of each iteration, during *domain analysis*, features are identified and captured in the variability model. These features are further designed, implemented, and tested during the subsequent activities, where the platform is created or extended. Phase-structured DE typically consists of long-running iterations, which, on the one hand, have to be thoroughly planned in advance, and on the other hand, address multiple features at the same time. [2]

Feature-Driven Domain Engineering. This is characterized in Figure 16.3 as counterpart to the phase-structured way. Here, both the feature model and the platform are evolved

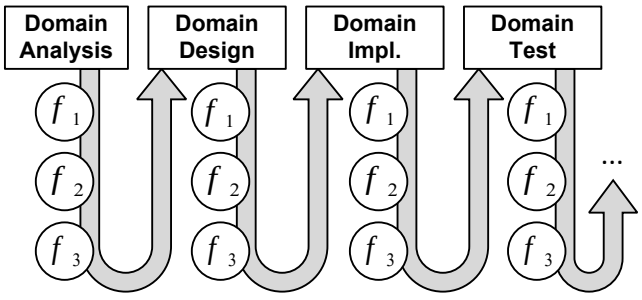


Figure 16.2: Phase-structured domain engineering. The identifiers f_i refer to different features and their connected realization artifacts in the platform. From [Schwä+16, Figure 4].

2 In the HAS case study, the subsequent phases have been organized into a sequence of fine-grained commits each. These, however, do not correspond to iterations in the process management sense.

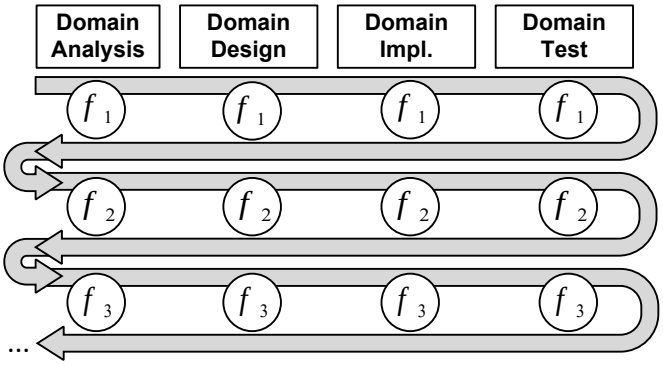


Figure 16.3: Feature-driven domain engineering. Based on [Schwä+16, Figure 5].

feature by feature. By introducing one feature (i.e., one increment) at a time, this results in comparably short-running iterations.

As far as AE is concerned, SuperMod's filtered editing model actually derives this activity from DE by relying on a product-based product line development approach. In the summary, it has been provocatively claimed that SuperMod allows to "perform application engineering, and get domain engineering for free". But is it really pure AE that is performed in the edit sessions? By definition, AE addresses development activities referring to single product variants; when transferred to SuperMod, this would correspond to iterations in which the choice equals the ambition, such that all changes are product-specific. Rather than entirely replacing DE by repeated AE, the here presented editing model blurs the distinction between both activities.

All remarks provided above refer to the development activities analysis, design, and implementation. Nevertheless, a fully-fledged development process, especially an agile one, should also cover *maintenance*. This was out of the scope of this thesis, but should be taken into consideration when defining an agile SPLE process.

16.3.3 Multi-Variant Model Transformations

In Section 15.5.2, we have discussed an occurrence of the *filter/transform dilemma* in one of the evaluation case studies. The presented solution – relying on an "almost-multi"-variant domain model and the concept of negative implementation – has been assessed as neither fully satisfactory nor generalizable.

Let us formally explain the general filter/transform dilemma. As sketched in Figure 16.4, the combination of product derivation (*filter*) and the application of a model transformation (*transform*) can be achieved in two ways. On the one hand, the filter can be repeatedly applied first, and the single-variant products can be transformed to corresponding target models by an existing model transformation. On the other hand, the transformation may be applied at first, transforming the multi-variant source model into a multi-variant target model, which can thereafter be configured into single-variant instances of the target product line.

In [SBW16b], we argue that the order *transform→filter* has significant advantages over

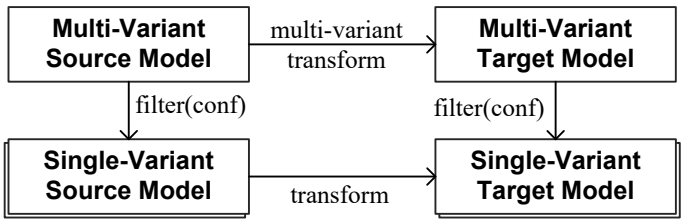


Figure 16.4: Modes of execution of the operations *filter* and *transform*. In a correct MVDM, the diagram should commute. From [SBW16b, Figure 2].

the off-the-shelf *filter→transform* solution, giving rise to *multi-variant model transformations* (MVMT). Albeit, the problem has been solved only for special cases of modeling languages or model transformation approaches in the literature. For instance, in [GSW17], a reuse-oriented approach towards MVMT based on the MDSPLE tool FAMILE [BS12b] (see Section 6.1.1) and the model-to-model transformation language ATL [Jou+08] (see Section 3.7.2) has been presented. Related approaches to MVMT, conducted under different premises, have been published by [Sal+14], by [Are+10], and by [Fam+15].

When transferred to SuperMod, MVMT are connected to additional challenges. First, multi-variant models are represented *extrinsically*, such that they do not conform to the source and target metamodel expected by the transformation engine; due to unconstrained variability, this does not only imply a conversion problem, but also a potential information loss as alternative variation must be resolved. Second, the transformation must also correctly propagate *traceability links* attached to elements of the source model to elements of the target model. This contradicts with the base mechanism of filtered editing, according to which the visibilities of new elements (part of the target model) are determined by the ambition.

16.4 Future Work

Subsequently to the conceptual discussions provided above, we present rather technical suggestions for future extensions to *SuperMod* (cf. Chapter 14). These should be implemented in order to make the approach feasible in real-world productive scenarios.

More VCS Commands. Being a research prototype, the tool offers a set of commands sufficient to conduct scientific experiments. In order to meet industrial requirements, the palette should be extended. Many VCS that follow an optimistic synchronization paradigm support in addition a LOCK operation that prevents specific resources from being modified remotely. Furthermore, *visual difference reports* are essential for understanding changes. Last, *branches* have been intentionally forbidden in the current version of SuperMod in favor of intensional versioning. Nevertheless, many continuous integration strategies employed in industry build upon branches, such that a corresponding command should be offered. In addition to the session-based *check-out/commit* model, an alternative *instant sharing* mode as offered, e.g., by [Dem+15] might increase the acceptance of the tool.

Improved Version Selection through Constraint Propagation. In the version space base layer (see Section 9.2.2), *invariants* have been introduced as propositional logical expressions that decide about the consistency of versions, whereas *preferences* and *defaults* assist the user in version selection. In many cases, preferences enforce related invariants, but in general, the underlying mechanisms are disjoint.

There are, however, situations in which the states of missing bindings can be determined from invariants directly. For instance, in a XOR feature group, when selecting one grouped feature positively, the only allowed selection state for all sibling features is negative. Such situations can be detected and enforced by satisfiability analysis and *constraint propagation*, respectively [Bat05].

Secure Authentication and Privileges Management. The server side application does not support secure authentication currently; everybody who knows the URI of a remote repository may access it. This, of course, does not meet any security guidelines.

Read or write access to specific resources (or even to their contents on a fine-grained basis) may be restricted to specific users or groups thereof.

Additional Version Dimensions. Privileges management as motivated by the above item might be modeled as an additional version dimension; read/write access rules for newly added/modified artifacts would be defined at commit. Furthermore, *requirements tracing* may be covered as follows: Each requirement is mapped to an option; on commit, the addressed requirements are indicated by the developer. As the underlying requirements model is subject to historical evolution, this might be another example of a hybrid dimension.

Apart from additional dimensions, new applications might involve a new composition of existing dimensions. SuperMod's flexible architecture generally allows for multiple instantiations of dimensions; therefore, e.g., feature model variability might be realized by a two-level hierarchy of feature models where one instance serves as variability model for the other instance.

Selections in the Product Space. In SuperMod, the root of a versioned SPL is an Eclipse project. It is possible to check-out the whole project, but not a part (i.e., a specific sub-directory) thereof. In order to meet the requirements of larger-scaled projects, selections in the product space should be supported.

Split Ambitions. The conceptual problem underlying limitation **L2** (*equal scope assumption*) might be solved by providing the possibility to *split* the commit into different *logical scopes*, each being described by an individual feature ambition. Conceptually, supporting *split ambitions* would require to introduce a human-interpretable representation of *write sets* and the possibility to decompose them in a disjoint way.

Import of Existing Product Line Projects. With the command EXPORT, SuperMod offers the possibility of making a specified version of a managed project available for external tools. The inverse operation, IMPORT, has not been realized, though. It would offer benefits such as the integration with unfiltered SPLE tools, such that developers might switch between

single-version and multi-version views, e.g., when being restricted by *limited awareness* (**L1**, see above).

Due to a lack of standardization, this is difficult from a technical point of view. Different SPLE tools have their individual language for feature models, their own limitations with respect to the structure of the multi-variant domain model (or components in the case of compositional variability), or they follow entirely different paradigms (such as extensional versioning in extractive *clone-and-own* approaches). Moreover, the most widespread SPL implementation technique, conditional compilation by preprocessor directives, is irreconcilable with the conceptual level of the SuperMod approach.

Further Content Types for Multi-Variant Files. In analogy to EMF instances and plain text files, additional file types can be supported in future. These include, among others, XML files, CSV files, specific formats of configuration files, or database entities.

AST-based Versioning of Source Code. According to Section 6.2, line-oriented versioning is inadequate for XMI-serialized model resources. But also for text files, the object granularity "line" is no more than an approximation. Better precision could be achieved by a structured versioning approach based on the *abstract syntax tree* (AST) of source code.

Using *reverse engineering* plug-ins such as *MoDisco* [Bru+10], source code conforming to particular programming languages can be represented as model instances immediately reflecting the corresponding AST. The combination of AST-based versioning and SuperMod promises a finer object granularity for the versioning of source code.

Eclipse-Independent Tool Interface. SuperMod's user interface is tightly coupled into Eclipse, although the base functionality is generic and has been designed and implemented in a reusable way; see Section 14.4. For scientific applications, being tied to Eclipse may not constitute a decisive obstacle, but industrial stakeholders may prefer to use their individual IDEs. To better integrate with other tools, it would become necessary to develop an Eclipse-independent user interface that relies, e.g., on a command line interface as back-end, and that connects to different IDEs.

Evaluation Perspectives. In many regards, the evaluation presented in Chapter 15 must be understood as a preliminary proof of concept that should be extended to a quantitative comparison of the approach against state-of-the-art tools. In order to learn more about the benefits and limitations of the approach, the experimentation might be extended into multiple directions.

– First, the assessment of the product well-formedness analysis approach was of limited significance inasmuch as only three out of 16 possible *product conflict types* were investigated. In an additional case study, the remaining conflicts might be intentionally produced in order to investigate, e.g., the precision of default resolution and the meaningfulness of the conflict descriptions produced.

– Second, the HAS case study was organized along the classical development phases requirements analysis, design, and implementation. SPL processes additionally include testing as

an essential activity. *SPL testing* involves, e.g., test case modularization or feature interaction handling [ER11]. It is worth investigating how the filtered SPL editing paradigm harmonizes with these challenges.

– Third, in Chapter 15, the evaluation of memory consumption and runtime behavior was disregarded intentionally. Although it was not a goal of this project to develop a "faster Subversion", insights into corresponding numbers might reflect the feasibility and scalability of the approach, and therefore provide important input for future work.

– Last, for a fair comparison with related SPL editing approaches, *independent comparative studies* should be conducted, for instance, in a contest format à la *Transformation Tool Contest* [GKR16]. In this way, we expect to learn more about the barriers of entry and the intuitiveness of the here presented approach.

To address many of these perspectives, the tool SuperMod must be extended with support for automatic quantitative evaluation, which will obviate the manual analysis of repositories and user interaction logs that served as a basis for the evaluation presented here.

16.5 Relevance for Research and Industry

The thesis is concluded by a natural question: Who may profit from the research presented here? The answer is twofold, pointing out to the academic and the industrial relevance of the research, respectively.

On the one hand, in the course of the development of the conceptual framework, it has turned out that the connection between evolution and variability, as well as the connection of both with abstraction, has not been fully investigated in the literature yet. The research presented here aims at increasing the theoretical body of knowledge in concepts and tool support for the integration of the considered software engineering sub-disciplines. Moreover, the research presented here has contributed to the upcoming research area of *variation control systems* [WO14; Stă+16; LBG17]. The implications of product-based product line development and of the underlying software process motivate future scientific work.

On the other hand, with SuperMod, we provide a tool that exploits synergy effects between SPLE and VC. With feature models, we employ an established mechanism for logical versioning that decisively goes beyond the state-of-the-art VC variability mechanism of branches. By adopting the filtered editing strategy and the VCS metaphors *check-out*, *modify*, and *commit*, the tool relieves SPL developers from cognitively complex multi-variant decisions. Version management is managed automatically behind the curtains, while only a single-version workspace is presented to the user, who may use arbitrary editing tools. The strict separation between DE and AE is blurred in favor of the compatibility with agile development practices and reactive SPL adoption. The presented solution holistically supports collaborative model-driven and/or software product line engineering. Software engineers, regardless of whether or not they align themselves with the model-driven paradigm, are equipped with an integrated yet unobtrusive tool that automatically and consistently manages the evolution and the variability of both models and source code.

List of Figures

List of Tables

List of Algorithms

List of Listings

Bibliography

References to publications by others are provided first. On page 403, thesis-related publications are listed. Further publications by the author of this thesis are listed on page 405.

Publications by Others

[AC04] Michal Antkiewicz and Krzysztof Czarnecki. "FeaturePlugin: feature modeling plug-in for Eclipse". In: *Proceedings of the 2004 OOPSLA workshop on Eclipse Technology eXchange, ETX 2004, Vancouver, British Columbia, Canada, October 24, 2004*. ACM, 2004, pp. 67–72. DOI: 10.1145/1066129.1066143.

[AK09] 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.

[AK15] Colin Atkinson and Thomas Kühne. "In Defence of Deep Modelling". In: *Information and Software Technology* 64 (2015), pp. 36–51. DOI: 10.1016/j.infsof.2015.03.010.

[AP03] Marcus Alanen and Ivan Porres. "Difference and Union of Models". In: *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings*. Springer, 2003, pp. 2–17. DOI: 10.1007/978-3-540-45221-8_2.

[AP11] Kerstin Altmanninger and Alfonso Pierantonio. "A categorization for conflicts in model versioning". In: *Elektrotechnik und Informationstechnik* 128.11-12 (2011), pp. 421–426. DOI: 10.1007/s00502-011-0063-z.

[ASW09] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. "A survey on model versioning approaches". In: *International Journal of Web Information Systems* 5.3 (2009), pp. 271–304. DOI: 10.1108/17440080910983556.

[Ach+11] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. "Slicing feature models". In: *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*. IEEE, 2011, pp. 424–427. DOI: 10.1109/ASE.2011.6100089.

[Ada+07] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. "The Evolution of the Linux Build System". In: *ECEASST* 8 (2007).

[Aho+06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Prentice Hall, 2006. ISBN: 0321486811.

[Ake78] Sheldon B. Akers. "Binary Decision Diagrams". In: *IEEE Transactions on Computers* 27.6 (1978), pp. 509–516. DOI: 10.1109/TC.1978.1675141.

[Alt+08] Kerstin Altmanninger, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Martina Seidl, Wieland Schwinger, and Manuel Wimmer. "AMOR - Towards Adaptable Model Versioning". In: *1st International Workshop on Model Co-Evolution and Consistency Management (MCCM'08), Workshop at MODELS'08, Toulouse, France, 2008*. ACM, 2008.

[Alt07] Kerstin Altmanninger. "Models in Conflict - Towards a Semantically Enhanced Version Control System for Models". In: *Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers*. Springer, 2007, pp. 293–304. DOI: 10.1007/978-3-540-69073-3_31.

[Ap+09a] Sven Apel, Christian Kästner, and Christian Lengauer. "FeatureHouse: Language-independent, automated software composition". In: *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 2009, pp. 221–231. DOI: 10.1109/ICSE.2009.5070523.

[Ap+09b] Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. "Model Superimposition in Software Product Lines". In: *Theory and Practice of Model Transformations, Second International Conference, ICMT 2009, Zurich, Switzerland, June 29-30, 2009. Proceedings*. Springer, 2009, pp. 4–19. DOI: 10.1007/978-3-642-02408-5_2.

[Ap+10] Sven Apel, Christian Kästner, Armin Größlinger, and Christian Lengauer. "Type safety for feature-oriented product lines". In: *Automated Software Engineering* 17.3 (2010), pp. 251–300. DOI: 10.1007/s10515-010-0066-8.

[Ap+13a] Sven Apel, Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. "Exploring Feature Interactions in the Wild: The New Feature-interaction Challenge". In: *5th International Workshop on Feature-Oriented Software Development, FOSD '13, Indianapolis, IN, USA, October 26, 2013*. ACM, 2013, pp. 1–8. DOI: 10.1145/2528265.2528267.

[Ap+13b] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013. ISBN: 3642375200.

[Ap07] Sven Apel. "The role of features and aspects in software development: similarities, differences, and synergetic potential". PhD thesis. Otto-von-Guericke University Magdeburg, Germany, 2007.

[Are+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. "Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations". In: *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*. Springer, 2010, pp. 121–135. DOI: 10.1007/978-3-642-16145-2_9.

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley, 2004. ISBN: 0321278658.

[BBH69] Friedrich L. Bauer, Louis Bolliet, and H. J. Helms. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee*. Ed. by Peter Naur and Brian Randell. Garmisch, Germany: Scientific Affairs Division, NATO, 1969.

[BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. "The Goal Question Metric Approach". In: *Encyclopedia of Software Engineering*. Wiley, 1994.

[BCW11] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wasowski. "Feature and Meta-models in Clafer: Mixed, Specialized, and Coupled". In: *Software Language Engineering - 3rd International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*. Springer, 2011, pp. 102–122. DOI: 10.1007/978-3-642-19440-5_7.

[BDW12] Thomas Buchmann, Alexander Dotor, and Bernhard Westfechtel. “MOD2-SCM: A model-driven product line for software configuration management systems”. In: *Information and Software Technology* 55.3 (2012), pp. 630–650. DOI: 10.1016/j.infsof.2012.07.010.

[BG16] Thomas Buchmann and Sandra Greiner. “Handcrafting a Triple Graph Transformation System to Realize Round-trip Engineering Between UML Class Models and Java Source Code”. In: *Proceedings of the 11th International Conference on Software Paradigm Trends*. SCITEPRESS, 2016, pp. 27–38. DOI: 10.5220/0005957100270038.

[BLF14] Omar Bahy Badreddin, Timothy C. Lethbridge, and Andrew Forward. “A Novel Approach to Versioning and Merging Model and Code Uniformly”. In: *MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7 - 9 January, 2014*. SCITEPRESS, 2014, pp. 254–263. DOI: 10.5220/0004699802540263.

[BP08] Cédric Brun and Alfonso Pierantonio. “Model Differences in the Eclipse Modelling Framework”. In: *UPGRADE, The European Journal for the Informatics Professional* IX.2 (2008), pp. 29–34.

[BP10] Daniel Le Berre and Anne Parrain. “The Sat4j library, release 2.2”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 7.2-3 (2010), pp. 59–6.

[BPB17] Benjamin Behringer, Jochen Palz, and Thorsten Berger. “PEoPL: projectional editing of product lines”. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE/ACM, 2017, pp. 563–574. DOI: 10.1109/ICSE.2017.58.

[BPV10a] Mark van den Brand, Zvezdan Protić, and Tom Verhoeff. “Fine-grained Metamodel-assisted Model Comparison”. In: *Proceedings of the 1st International Workshop on Model Comparison in Practice, Malaga, Spain, 2010*. ACM, 2010, pp. 11–20. DOI: 10.1145/1826147.1826152.

[BPV10b] Mark van den Brand, Zvezdan Protić, and Tom Verhoeff. “Generic tool for visualization of model differences”. In: *Proceedings of the 1st International Workshop on Model Comparison in Practice, Malaga, Spain, 2010*. ACM, 2010, pp. 66–75. DOI: 10.1145/1826147.1826160.

[BS02] Mike A. Beedle and Ken Schwaber. *Agile Software Development with Scrum*. Prentice Hall, Feb. 2002. ISBN: 0130676349.

[BS81] F. Bancilhon and N. Spyratos. “Update Semantics of Relational Views”. In: *ACM Transactions on Database Systems* 6.4 (1981), pp. 557–575. DOI: 10.1145/319628.319634.

[BSR04] Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. “Scaling Step-Wise Refinement”. In: *IEEE Transactions on Software Engineering* 30.6 (2004), pp. 355–371. DOI: 10.1109/TSE.2004.23.

[BW12] Thomas Buchmann and Bernhard Westfechtel. “Mapping feature models onto domain models: ensuring consistency of configured domain models”. In: *Software & Systems Modeling* 13.4 (2012), pp. 1–33. DOI: 10.1007/s10270-012-0305-5.

[Bab86] Wayne A. Babich. *Software Configuration Management: Coordination for Team Productivity*. Addison-Wesley Longman, 1986. ISBN: 0-201-10161-0.

[Bat05] Don Batory. “Feature Models, Grammars, and Propositional Formulas”. In: *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*. Springer, 2005, pp. 7–20. DOI: 10.1007/11554844_3.

[Bec+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. *Manifesto for Agile Software Development*. 2001. URL: http://www.agilemanifesto.org/.

[Beu06] Ottmar Beucher. *MATLAB und Simulink (Scientific Computing)*. German. Pearson Studium, Aug. 2006. ISBN: 3827372062.

[Bru+10] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. "MoDisco: a generic and extensible framework for model driven reverse engineering". In: *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*. ACM, 2010, pp. 173–174. DOI: 10.1145/1858996.1859032.

[Buc12] Thomas Buchmann. "Valkyrie: A UML-Based Model-Driven Environment for Model-Driven Software Engineering". In: *ICSOFT 2012 - Proceedings of the 7th International Conference on Software Paradigm Trends, Rome, Italy, 24 - 27 July, 2012*. SCITEPRESS, 2012, pp. 147–157.

[Bur09] Bill Burke. *RESTful Java with Jax-RS*. O'Reilly, 2009. ISBN: 0596158041.

[Béz05] Jean Bézivin. "On the unification power of models". In: *Software & Systems Modeling* 4.2 (2005), pp. 171–188. DOI: 10.1007/s10270-005-0079-0.

[CA05] Krzysztof Czarnecki and Michał Antkiewicz. "Mapping Features to Models: A Template Approach Based on Superimposed Variants". In: *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proceedings*. Springer, 2005, pp. 422–437. DOI: 10.1007/11561347_28.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM / Addison-Wesley, 2000. ISBN: 0-201-30977-7.

[CFP04] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly, 2004. ISBN: 978-0-596-00448-4.

[CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. "Staged Configuration Using Feature Models". In: *Software Product Lines*. Vol. 3154. Lecture Notes in Computer Science. Springer, 2004, pp. 266–283. DOI: 10.1007/978-3-540-28630-1_17.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. "Formalizing cardinality-based feature models and their specialization". In: *Software Process: Improvement and Practice* 10.1 (2005), pp. 7–29. DOI: 10.1002/spip.213.

[CM00] M. Calder and E. Magill. *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press, 2000. ISBN: 1586030655.

[CM91] Reidar Conradi and Carl Chr. Malm. "Cooperating Transactions and Workspaces in EPOS: Design and Preliminary Implementation". In: *Proceedings of the 3rd International Conference on Advanced Information Systems Engineering (CAiSE'91)*. 1991, pp. 375–392.

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. ISBN: 9780201703320.

[CP06] Krzysztof Czarnecki and Krzysztof Pietroszek. “Verifying Feature-based Model Templates Against Well-formedness OCL Constraints”. In: *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings*. ACM, 2006, pp. 211–220. DOI: 10.1145/1173706.1173738.

[CR06] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins (2nd Edition)*. Addison-Wesley, 2006. ISBN: 032142672X.

[CW97] Reidar Conradi and Bernhard Westfechtel. “Towards a Uniform Version Model for Software Configuration Management”. In: *System Configuration Management, ICSE’97 SCM-7 Workshop, Boston, MA, USA, May 18-19, 1997, Proceedings*. Springer, 1997, pp. 1–17. DOI: 10.1007/3-540-63014-7_1.

[CW98] Reidar Conradi and Bernhard Westfechtel. “Version Models for Software Configuration Management”. In: *ACM Computing Surveys* 30.2 (June 1998), pp. 232–282. DOI: 10.1145/280277.280280.

[Car+13] Eric Cariou, Olivier Le Goaer, Franck Barbier, and Samson Pierre. “Characterization of Adaptable Interpreted-DSML”. In: *Modelling Foundations and Applications - 9th European Conference, ECMFA 2013, Montpellier, France, July 1-5, 2013. Proceedings*. Springer, 2013, pp. 37–53. DOI: 10.1007/978-3-642-39013-5_4.

[Cha+96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. “Change Detection in Hierarchically Structured Information”. In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*. ACM, 1996, pp. 493–504. DOI: 10.1145/233269.233366.

[Cha09] Scott Chacon. *Pro Git*. Apress, 2009. ISBN: 1430218339.

[Con+04] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. “Thirty Years Of Graph Matching In Pattern Recognition”. In: *International Journal on Pattern Recognition and Artificial Intelligence* 18.3 (2004), pp. 265–298. DOI: 10.1142/S0218001404003228.

[Cza+12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. “Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches”. In: *6th International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings*. ACM, 2012, pp. 173–182. DOI: 10.1145/2110147.2110167.

[Dar91] Susan A. Dart. “Concepts in Configuration Management Systems”. In: *Proceedings of the 3rd International Workshop on Software Configuration Management, Trondheim, Norway, June 12-14, 1991*. ACM, 1991, pp. 1–18. DOI: 10.1145/111062.111063.

[Dem+15] Andreas Demuth, Markus Riedl-Ehrenleitner, Alexander Nöhrer, Peter Hehenberger, Klaus Zeman, and Alexander Egyed. “DesignSpace: an infrastructure for multi-user/multi-tool engineering”. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*. ACM, 2015, pp. 1486–1491. DOI: 10.1145/2695664.2695697.

[Dhu+08] Deepak Dhungana, Thomas Neumayer, Paul Grünbacher, and Rick Rabiser. “Supporting Evolution in Model-Based Product Line Engineering”. In: *Proceedings of the 12th International Conference on Software Product Lines (SPLC 2008), Limerick, Ireland, September 8-12*. IEEE, 2008, pp. 319–328. DOI: 10.1109/SPLC.2008.26.

[Dij72] Edsger W. Dijkstra. “The Humble Programmer”. In: *Communications of the ACM* 15.10 (1972), pp. 859–866. DOI: 10.1145/355604.361591.

[Dot11] Alexander Dotor. “Entwurf und Modellierung einer Produktlinie von Software-Konfigurations-Management-Systemen”. German. PhD thesis. University of Bayreuth, 2011.

[Dub+13] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. “An Exploratory Study of Cloning in Industrial Software Product Lines”. In: *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*. IEEE, 2013, pp. 25–34. DOI: 10.1109/CSMR.2013.13.

[EC94] Jacky Estublier and Rubby Casallas. “The Adele Configuration Manager”. In: *Configuration Management*. Vol. 2. Trends in Software. John Wiley & Sons, 1994, pp. 99–134.

[EDL10] Jacky Estublier, Idrissa A. Dieng, and Thomas Leveque. “Software Product Line Evolution: The Selecta System”. In: *Proceedings of the 2010 ICSE Workshop on Product Line Approaches in Software Engineering, PLEASE 2010, Cape Town, South Africa, May 2, 2010*. ACM, 2010, pp. 32–39. DOI: 10.1145/1808937.1808942.

[EJ01] D. Eastlake and P. Jones. *US Secure Hash Algorithm 1 (SHA1)*. RFC 3174. Internet Engineering Task Force, Sept. 2001.

[ER11] Emelie Engström and Per Runeson. “Software product line testing - A systematic mapping study”. In: *Information & Software Technology* 53.1 (2011), pp. 2–13. DOI: 10.1016/j.infsof.2010.05.011.

[EW11] Martin Erwig and Eric Walkingshaw. “The Choice Calculus: A Representation for Software Variation”. In: *ACM Transactions on Software Engineering and Methodology* 21.1 (2011), pp. 1–31. DOI: 10.1145/2063239.2063245.

[EWC13] Martin Erwig, Eric Walkingshaw, and Sheng Chen. “An abstract representation of variational graphs”. In: *5th International Workshop on Feature-Oriented Software Development, FOSD ’13, Indianapolis, IN, USA, October 26, 2013*. ACM, 2013, pp. 25–32. DOI: 10.1145/2528265.2528270.

[EdD+16] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, Abel Gómez, Massimo Tisi, and Jordi Cabot. “EMF-REST: generation of RESTful APIs from models”. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*. ACM, 2016, pp. 1446–1453. DOI: 10.1145/2851613.2851782.

[Fah+14] Uli Fahrenberg, Mathieu Acher, Axel Legay, and Andrzej Wasowski. “Sound Merging and Differencing for Class Diagrams”. In: *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Springer, 2014, pp. 63–78. DOI: 10.1007/978-3-642-54804-8_5.

[Fam+15] Michalis Famelis, Levi Lucio, Gehan M. K. Selim, Alessio Di Sandro, Rick Salay, Marsha Chechik, James R. Cordy, Jürgen Dingel, Hans Vangheluwe, and S. Ramesh. “Migrating Automotive Product Lines: A Case Study”. In: *Theory and Practice of Model Transformations - 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L’Aquila, Italy, July 20-21, 2015. Proceedings*. Springer, 2015, pp. 82–97. DOI: 10.1007/978-3-319-21155-8_7.

[Fei91] Peter H. Feiler. *Configuration management models in commercial environments*. Tech. rep. CMU/SEI-91-TR-7. Software Engineering Institute, Carnegie Mellon University, 1991.

[Fie00] Roy T. Fielding. “REST: Architectural Styles and the Design of Network-based Software Architectures”. PhD thesis. University of California, Irvine, 2000.

[Fis+15] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. “The ECCO Tool: Extraction and Composition for Clone-and-Own”. In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*. IEEE, 2015, pp. 665–668. DOI: 10.1109/ICSE.2015.218.

[Fit91] Melvin Fitting. “Kleene’s Logic, Generalized”. In: *Journal of Logic and Computation* 1.6 (1991), pp. 797–810. DOI: 10.1093/logcom/1.6.797.

[Fow03] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman, 2003. ISBN: 0321193687.

[Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley, 2010. ISBN: 0321712943.

[GKR16] Antonio García-Domínguez, Filip Krikava, and Louis M. Rose, eds. *Proceedings of the 9th Transformation Tool Contest, co-located with the 2016 Software Technologies: Applications and Foundations (STAF 2016), Vienna, Austria, July 8, 2016*. Vol. 1758. CEUR Workshop Proceedings. CEUR-WS.org, 2016. URL: http://ceur-ws.org/Vol-1758.

[GLS08] Alexander Gruler, Martin Leucker, and Kathrin D. Scheidemann. “Modeling and Model Checking Software Product Lines”. In: *Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference, FMOODS 2008, Oslo, Norway, June 4-6, 2008, Proceedings*. 2008, pp. 113–131. DOI: 10.1007/978-3-540-68863-1_8.

[GM08] Yaser Ghanam and Frank Maurer. “An Iterative Model for Agile Product Line Engineering”. In: *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*. IEEE, 2008, pp. 377–384.

[GS04] Hassan Gomaa and Michael E. Shin. “Tool Support for Software Variability Management and Product Derivation in Software Product Lines”. In: *Workshop on Software Variability Management for Product Derivation, Software Product Line Conference (SPLC), Boston, MA, August 30 - September 2, 2014*. Springer, 2004, pp. 73–84.

[GV07] Iris Groher and Markus Völter. “XWeave: models and aspects in concert”. In: *Proceedings of the 10th international workshop on Aspect-oriented modeling, Vancouver, BC, Canada, March 12 - 16, 2007*. ACM, 2007, pp. 35–40.

[Gam+95] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, 1995. ISBN: 0-201-63361-2.

[Gom05] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. ACM, 2005. ISBN: 978-0-201-77595-2.

[Gos+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification - Java SE8 Edition*. Sun Microsystems. 2015. URL: http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, 2009. ISBN: 0321534077.

[HDH02] Per Brinch Hansen, Edsger W. Dijkstra, and C. A. R. Hoare. *The Origins of Concurrent Programming: From Semaphores to Remote Procedure Calls*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2002. ISBN: 0387954015.

[HJH06] Scott Hendrickson, Bryan Jett, and André van der Hoek. "Layered Class Diagrams: Supporting the Design Process". In: *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*. Springer, 2006, pp. 722–736. DOI: 10.1007/11880240_50.

[HK05] Martin Hitz and Gerti Kappel. *UML @ Work*. German. dpunkt, 2005. ISBN: 3898642615.

[HKW08] Florian Heidenreich, Jan Kopcsek, and Christian Wende. "FeatureMapper: Mapping features to models". In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*. ACM, 2008, pp. 943–944. DOI: 10.1145/1370175.1370199.

[HM08] Terry Halpin and Tony Morgan. *Information Modeling and Relational Databases*. Morgan Kaufmann, 2008. ISBN: 0123735688.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman, 2006. ISBN: 0321455363.

[HP04] P. Hnetynka and F. Plasil. "Distributed versioning model for MOF". In: *Proceedings of WISICT 2004, Cancun, Mexico*. 2004, pp. 489–494.

[HS77] James W. Hunt and Thomas G. Szymanski. "A Fast Algorithm for Computing Longest Common Subsequences". In: *Communications of the ACM* 20.5 (1977), pp. 350–353. DOI: 10.1145/359581.359603.

[Hab+12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. "Evolving Delta-Oriented Software Product Line Architectures". In: *Large-Scale Complex IT Systems. Development, Operation and Management*. Vol. 7539. LNCS. Springer, 2012, pp. 183–208. DOI: 10.1007/978-3-642-34059-8_10.

[Hau+04] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, and Arnor Solberg. "An MDA-based framework for model-driven product derivation". In: *Proceedings of the IASTED Conference on Software Engineering and Applications, November 9-11, 2004, MIT, Cambridge, MA, USA*. IASTED/ACTA Press, 2004, pp. 709–714.

[Hau+08] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. "Adding Standardized Variability to Domain Specific Languages". In: *oftware Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*. IASTED/ACTA Press, 2008, pp. 139–148. DOI: 10.1109/SPLC.2008.25.

[Hec78] Paul Heckel. "A Technique for Isolating Differences Between Files". In: *Communications of the ACM* 21.4 (1978), pp. 264–268. DOI: 10.1145/359460.359467.

[Hei09] Florian Heidenreich. "Towards systematic ensuring well-formedness of software product lines". In: *Proceedings of the 1st International Workshop on Feature-Oriented Software Development, FOSD 2009, Denver, Colorado, USA, October 6, 2009*. ACM, 2009, pp. 69–74. DOI: 10.1145/1629716.1629730.

[Hof+10] Wanja Hofer, Christoph Elsner, Frank Blendinger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Toolchain-independent Variant Management with the Leviathan Filesystem". In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development, FOSD 2010, Eindhoven, Netherlands, October 10, 2010*. ACM, 2010, pp. 18–24. DOI: 10.1145/1868688.1868692.

[HŞW08] Florian Heidenreich, Ilie Şavga, and Christian Wende. "On Controlled Visualisations in Software Product Line Engineering". In: *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*. IEEE, 2008, pp. 335–341.

[IEEE05] Institute of Electrical and Electronics Engineers (IEEE). *IEEE Standard for Software Configuration Management Plans*. Revision of IEEE Standard 828-1998. 2005, pp. 1–30. DOI: 10.1109/IEEESTD.2005.96464.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006. ISBN: 0262101149.

[Jay+07] Praveen K. Jayaraman, Jon Whittle, Ahmed M. Elkhodary, and Hassan Gomaa. "Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis". In: *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*. Springer, 2007, pp. 151–165. DOI: 10.1007/978-3-540-75209-7_11.

[Jou+08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. "ATL: A model transformation tool". In: *Science of Computer Programming* 72.1-2 (2008), pp. 31–39. DOI: 10.1016/j.scico.2007.08.002.

[KC14] Charles W. Krueger and Paul C. Clements. "Systems and software product line engineering with gears from BigLever software". In: *18th International Software Product Lines Conference - Companion Volume for Workshop, Tools and Demo papers, SPLC '14, Florence, Italy, September 15-19, 2014*. ACM, 2014, pp. 121–125. DOI: 10.1145/2647908.2655976.

[KE14] Amanuel Koshima and Vincent Englebert. "Collaborative Editing of EMF/Ecore Metamodels and Models - Conflict Detection, Reconciliation, and Merging in DiCoMEF". In: *MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7 - 9 January, 2014*. SCITEPRESS, 2014, pp. 55–66. DOI: 10.5220/0004709500550066.

[KH10] Maximilian Kögel and Jonas Helming. "EMFStore: a model repository for EMF models". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. ACM, 2010, pp. 307–308. DOI: 10.1145/1810295.1810364.

[KKP07] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. "A Formal Investigation of Diff3". In: *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science, 27th International Conference, New Delhi, India, December 12-14, 2007, Proceedings*. Springer, 2007, pp. 485–496. DOI: 10.1007/978-3-540-77050-3_40.

[KKT13] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. "Consistency-preserving edit scripts in model versioning". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. IEEE, 2013, pp. 191–201. DOI: 10.1109/ASE.2013.6693079.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman, 2003. ISBN: 032119442X.

[KWN05] Udo Kelter, Jürgen Wehren, and Jörg Niere. "A Generic Difference Algorithm for UML Models". In: *Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik, 8.-11.3.2005 in Essen*. Gesellschaft für Informatik, 2005, pp. 105–116.

[Kan+90] 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. Software Engineering Institute, Carnegie Mellon University, Nov. 1990.

[Kel17] Steven Kelly. "Collaborative Modeling with Version Control". In: *Proceedings of the 5th BigMDE Workshop*. To be published. 2017.

[Key07] Jessica Keyes. *Software Configuration Management*. Taylor & Francis, 2007. ISBN: 0-8493-1976-5.

[Kic96] Gregor Kiczales. "Aspect-Oriented Programming". In: *ACM Computing Surveys* 28.4 (1996), p. 154. DOI: 10.1145/242224.242420.

[Kru03] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Boston, MA, USA: Addison-Wesley Longman, 2003. ISBN: 0321197704.

[Kru84] Vincent J. Kruskal. "Managing Multi-Version Programs with an Editor". In: *IBM Journal of Research and Development* 28.1 (1984), pp. 74–81. DOI: 10.1147/rd.281.0074.

[Käs+09] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don S. Batory. "Guaranteeing Syntactic Correctness for All Product Line Variants: A Language-Independent Approach". In: *Objects, Components, Models and Patterns, 47th International Conference, TOOLS EUROPE 2009, Zurich, Switzerland, June 29-July 3, 2009. Proceedings*. Springer, 2009, pp. 175–194. DOI: 10.1007/978-3-642-02571-6_11.

[Käs10] Christian Kästner. "Virtual Separation of Concerns: Towards Preprocessors 2.0". PhD thesis. University of Magdeburg, May 2010.

[Küh06] Thomas Kühne. "Matters of (Meta-) Modeling". In: *Software & Systems Modeling* 5.4 (2006), pp. 369–385. DOI: 10.1007/s10270-006-0017-9.

[LBG17] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. "A Classification of Variation Control Systems". In: *Proceedings of the 16th International Conference on Generative Programming, GPCE 2017, Vancouver, BC, Canada, 23 - 24 October 2017*. ACM, 2017, pp. 49–62. DOI: 10.1145/3136040.3136054.

[LC13] Miguel A. Laguna and Yania Crespo. "A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring". In: *Science of Computer Programming* 78.8 (2013), pp. 1010–1034. DOI: 10.1016/j.scico.2012.05.003.

[LELH16] Lukas Linsbauer, Alexander Egyed, and Roberto Erick Lopez-Herrejon. "A Variability Aware Configuration Management and Revision Control Platform". In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. ACM, 2016, pp. 803–806. DOI: 10.1145/2889160.2889262.

[LHB01] Roberto E. Lopez-Herrejon and Don S. Batory. "A Standard Problem for Evaluating Product-Line Methodologies". In: *Generative and Component-Based Software Engineering, 3rd International Conference, GCSE 2001, Erfurt, Germany, September 9-13, 2001, Proceedings*. Springer, 2001, pp. 10–24. DOI: 10.1007/3-540-44800-4_2.

[Lee90] Jan van Leeuwen. *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*. MIT Press, 1990. ISBN: 0-444-88071-2.

[Lin04] Tancred Lindholm. "A three-way merge for XML documents". In: *Proceedings of the 2004 ACM Symposium on Document Engineering, Milwaukee, Wisconsin, USA, October 28-30, 2004*. ACM, 2004, pp. 1–10. DOI: 10.1145/1030397.1030399.

[Lot+10] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. "Evolution of the Linux Kernel Variability Model". In: *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*. Springer, 2010, pp. 136–150. DOI: 10.1007/978-3-642-15579-6_10.

[MBG09] Alix Mougenot, Xavier Blanc, and Marie-Pierre Gervais. "D-Praxis : A Peer-to-Peer Collaborative Model Editing Framework". In: *Distributed Applications and Interoperable Systems, 9th IFIP WG 6.1 International Conference, DAIS 2009, Lisbon, Portugal, June 9-11, 2009. Proceedings*. Springer, 2009, pp. 16–29. DOI: 10.1007/978-3-642-02164-0_2.

[MD15] Leticia Montalvillo and Oscar Díaz. "Tuning GitHub for SPL development: branching models & repository operations for product engineers". In: *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*. ACM, 2015, pp. 111–120. DOI: 10.1145/2791060.2791083.

[ME08] Ralf Mitschke and Michael Eichberg. "Supporting the Evolution of Software Product Lines". In: *ECMDA Traceability Workshop (ECMDA-TW), Berlin, Germany, 2008, Proceedings*. SINTEF, 2008, pp. 87–96.

[MGMR02] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. "Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching". In: *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*. IEEE, 2002, pp. 117–128. DOI: 10.1109/ICDE.2002.994702.

[ML12] Tanja Mayerhofer and Philip Langer. "Moliz: A Model Execution Framework for UML Models". In: *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards, Innsbruck, Austria, 2012*. ACM, 2012, 3:1–3:2. DOI: 10.1145/2448076.2448079.

[MM03] Joaquin Miller and Jishnu Mukerji. *MDA Guide Version 1.0.1*. Tech. rep. omg/03-06-01. Object Management Group, 2003.

[MVA10] Jeff McAffer, Paul VanderLei, and Simon Archer. *OSGi and Equinox: Creating Highly Modular Java Systems*. Addison-Wesley Professional, 2010. ISBN: 0321585712.

[MWC09] Marcílio Mendonça, Andrzej Wasowski, and Krzysztof Czarnecki. "SAT-based analysis of feature models is easy". In: *Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings*. ACM, 2009, pp. 231–240. DOI: 10.1145/1753235.1753267.

[Mel+04] Stephen J. Mellor, Scott Kendall, Axel Uhl, and Dirk Weise. *MDA Distilled*. Addison-Wesley Longman, 2004. ISBN: 0201788918.

[Mun+93] Bjørn P. Munch, Jens-Otto Larsen, Bjørn Gulla, Reidar Conradi, and Even-André Karlsson. "Uniform Versioning: The Change-Oriented Model". In: *4th International Workshop on Software Configuration Management, Baltimore, May 1993, Proceedings*. Springer, 1993, pp. 188–196.

[Mun93] Bjørn P. Munch. "Versioning in a Software Engineering Database — The Change Oriented Way". PhD thesis. Tekniske Høgskole Trondheim Norges, 1993.

[Mun96] Bjørn P. Munch. "HiCoV: Managing the Version Space". In: *System Configuration Management, ICSE'96 SCM-6 Workshop, Berlin, Germany, March 25-26, 1996, Proceedings*. Springer, 1996, pp. 110–126. DOI: 10.1007/BFb0023084.

[Mül+00] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne D. Storey, Scott R. Tilley, and Kenny Wong. "Reverse engineering: a roadmap". In: *22nd International Conference on on Software Engineering, Future of Software Engineering Track, ICSE 2000, Limerick Ireland, June 4-11, 2000.* ACM, 2000, pp. 47–60. DOI: 10.1145/336512.336526.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. "The FUJABA environment". In: *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000*. ACM, 2000, pp. 742–745. DOI: 10.1145/337180.337620.

[Ngu+05] Tien Nhut Nguyen, Ethan V. Munson, John Tang Boyland, and Cheng Thao. "An Infrastructure for Development of Object-oriented, Multi-level Configuration Management Services". In: *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. ACM, 2005, pp. 215–224. DOI: 10.1145/1062455.1062504.

[OMG08] Object Management Group (OMG). *MOF Model To Text Transformation Language (MOFM2T), Version 1.0*. formal/08-01-16. Needham, MA, 2008. URL: http://www.omg.org/spec/MOFM2T/1.0/.

[OMG12] Object Management Group (OMG). *Common Variability Language (CVL) OMG Revised Submission*. Needham, MA, Aug. 2012. URL: http://www.omgwiki.org/variability/lib/exe/fetch.php?media=cvl-revised-submission.pdf.

[OMG13] Object Management Group (OMG). *Documents Associated With Concrete Syntax For A UML Action Language: Action Language For Foundational UML (ALF), Version 1.0.1*. formal/2013-09-01. Needham, MA, 2013. URL: http://www.omg.org/spec/ALF/1.0.1/.

[OMG14] Object Management Group (OMG). *Documents Associated With Object Constraint Language (OCL), Version 2.4*. formal/2014-02-03. Needham, MA, 2014. URL: http://www.omg.org/spec/OCL/2.4/.

[OMG15] Object Management Group (OMG). *Documents Associated With Unified Modeling Language (UML), Version 2.5*. formal/15-03-01. Needham, MA, 2015. URL: http://www.omg.org/spec/UML/2.5/.

[OMG15a] Object Management Group (OMG). *Documents Associated With XML Metadata Interchange (XMI), Version 2.5.1*. formal/2015-06-07. Needham, MA, 2015. URL: http://www.omg.org/spec/XMI/2.5.1/.

[OMG16] Object Management Group (OMG). *Documents Associated With Meta Objet Facility (MOF) 2.0 Query/View/Transformation, V1.3*. formal/2016-06-03. Needham, MA, 2016. URL: http://www.omg.org/spec/QVT/1.3/.

[OMW08] Hamilton L. R. Oliveira, Leonardo Gresta Paulino Murta, and Cláudia Werner. "Odyssey-VCS: a flexible version control system for UML model elements". In: *Proceedings of the 12th International Workshop on Software Configuration Management, SCM 2005, Lisbon, Portugal, September 5-6, 2005*. ACM, 2008, pp. 1–16. DOI: 10.1145/1109128.1109129.

[PBL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin, Germany: Springer, 2005. ISBN: 9783642063640.

[PF01] Steve R. Palmer and Mac Felsing. *A Practical Guide to Feature-Driven Development*. Pearson Education, 2001. ISBN: 0130676152.

[PH08] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface*. Morgan Kaufmann, 2008. ISBN: 0123744938.

[Pfo+16] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. "Synchronizing software variants with VariantSync". In: *Proceedings of the 20th International Systems and Software Product Line Conference, SPLC 2016, Beijing, China, September 16-23, 2016*. ACM, 2016, pp. 329–332. DOI: 10.1145/2934466.2962726.

[Pie+15] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. "SiPL - A Delta-Based Modeling Framework for Software Product Line Engineering". In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE, 2015, pp. 852–857. DOI: 10.1109/ASE.2015.106.

[Pop09] Gunther Popp. *Konfigurationsmanagement mit Subversion, Maven und Redmine: Grundlagen für Softwarearchitekten und Entwickler*. German. dpunkt, 2009. ISBN: 978-3-89864-521-8.

[RCC13] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. "Managing Cloned Variants: A Framework and Experience". In: *17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan - August 26 - 30, 2013*. ACM, 2013, pp. 101–110. DOI: 10.1145/2491627.2491644.

[RS77] Daniel R. Ries and Michael Stonebraker. "Effects of Locking Granularity in a Database Management System". In: *ACM Transactions on Database Systems* 2.3 (1977), pp. 233–246. DOI: 10.1145/320557.320566.

[RV08] José Eduardo Rivera and Antonio Vallecillo. "Representing and Operating with Model Differences". In: *Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008, Zurich, Switzerland, June 30 - July 4, 2008. Proceedings*. Springer, 2008, pp. 141–160. DOI: 10.1007/978-3-540-69824-1_9.

[Raj06] Vaclav Rajlich. "Changing the Paradigm of Software Engineering". In: *Communications of the ACM* 49.8 (2006), pp. 67–70. DOI: 10.1145/1145287.1145289.

[Rei95] Christoph Reichenberger. "VOODOO - A Tool for Orthogonal Version Management". In: *Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops, Selected Papers*. Springer, 1995, pp. 61–79. DOI: 10.1007/3-540-60578-9_4.

[Roc75] Marc J. Rochkind. "The source code control system". In: *IEEE Transactions on Software Engineering* 1.4 (1975), pp. 364–370. DOI: 10.1109/TSE.1975.6312866.

[Roy70] Walker W. Royce. "Managing the development of large software systems: concepts and techniques". In: *Proceedings, 9th International Conference on Software Engineering, Monterey, California, USA, March 30 - April 2, 1987*. ACM, 1970, pp. 328–339.

[Roz97] Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific Publishing, 1997. ISBN: 98-102288-48.

[Rut+09] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. "A Category-Theoretical Approach to the Formalisation of Version Control in MDE". In: *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Springer, 2009, pp. 64–78. DOI: 10.1007/978-3-642-00593-0_5.

[SBK88] Neil Sarnak, Robert L. Bernstein, and V. Kruskal. "Creation and Maintenance of Multiple Versions." In: *Proceedings of the International Workshop on Software Version and Configuration Control, January 27-29, 1988, Grassau, Germany*. Vol. 30. Teubner, 1988, pp. 264–275.

[SH04] Mark Staples and Derrick Hill. "Experiences Adopting Software Product Line Development without a Product Line Architecture". In: *11th Asia-Pacific Software Engineering Conference (APSEC 2004), 30 November - 3 December 2004, Busan, Korea*. IEEE, 2004, pp. 176–183. DOI: doi.ieeecomputersociety.org/10.1109/APSEC.2004.50.

[SHA12] Christoph Seidl, Florian Heidenreich, and Uwe Aßmann. "Co-evolution of Models and Feature Mapping in Software Product Lines". In: *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 1*. ACM, 2012, pp. 76–85. DOI: 10.1145/2362536.2362550.

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. "Feature Diagrams: A Survey and a Formal Semantics". In: *14th IEEE International Conference on Requirements Engineering (RE 2006), 11-15 September 2006, Minneapolis/St.Paul, Minnesota, USA*. IEEE, 2006, pp. 136–145. DOI: 10.1109/RE.2006.23.

[SK03] Shane Sendall and Wojtek Kozaczynski. "Model Transformation: The Heart and Soul of Model-Driven Software Development". In: *IEEE Software* 20.5 (2003), pp. 42–45. DOI: 10.1109/MS.2003.1231150.

[SPC16] Anjali Sree-Kumar, Elena Planas, and Robert Clarisó. "Analysis of Feature Models Using Alloy: A Survey". In: *Proceedings 7th International Workshop on Formal Methods and Analysis in Software Product Line Engineering, FMSPLE@ETAPS 2016, Eindhoven, The Netherlands, April 3, 2016*. ArXive, 2016, pp. 46–60. DOI: 10.4204/EPTCS.206.5.

[SS03] Robert Sedgewick and Michael Schidlowsky. *Algorithms in Java, Part 5: Graph Algorithms*. Addison-Wesley Longman, 2003. ISBN: 0201361213.

[SS08] Julio Sincero and Wolfgang Schröder-Preikschat. "The Linux Kernel Configurator as a Feature Modeling Tool". In: *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*. IEEE, 2008, pp. 257–260.

[SSA14a] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. "Capturing Variability in Space and Time with Hyper Feature Models". In: *8th International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '14, Sophia Antipolis, France, January 22-24, 2014*. ACM, 2014, 6:1–6:8. DOI: 10.1145/2556624.2556625.

[SSA14b] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. "Integrated Management of Variability in Space and Time in Software Families". In: *18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014*. ACM, 2014, pp. 22–31. DOI: 10.1145/2648511.2648514.

[SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development: Technology, Enginering, Management*. John Wiley & Sons, 2006. ISBN: 9780470025703.

[SZN04] Christian Schneider, Albert Zündorf, and Jörg Niere. "CoObRA - a small step for development tools to collaborative environments". In: *Workshop on Directions in Software Engineering Environments in 26th international conference on software engineering, ICSE 2004, Edinburgh, UK, 28 May 2004, Proceedings*. IEEE, 2004.

[Sal+14] Rick Salay, Michalis Famelis, Julia Rubin, Alessio Di Sandro, and Marsha Chechik. "Lifting model transformations to product lines". In: *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. ACM, 2014, pp. 117–128. DOI: 10.1145/2568225.2568267.

[Schae+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. "Delta-Oriented Programming of Software Product Lines". In: *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*. Springer, 2010, pp. 77–91. DOI: 10.1007/978-3-642-15579-6_6.

[Schi17] Stefan Schill. "Untersuchung und Implementierung einer Adaption des Heckel-Algorithmus für gerichtete Graphen". Bachelor Thesis. University of Bayreuth, 2017.

[Schü94] Andy Schürr. "Specification of Graph Translators with Triple Graph Grammars". In: *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings*. Vol. 903. Springer, 1994, pp. 151–163. DOI: 10.1007/3-540-59071-4_45.

[Smo92] Gert Smolka. "Feature-constraint logics for unification grammars". In: *The Journal of Logic Programming* 12.1 (1992), pp. 51–87. DOI: 10.1016/0743-1066(92)90039-6.

[Som06] Ian Sommerville. *Software Engineering (8th Edition)*. Addison-Wesley Longman, 2006. ISBN: 0321313798.

[Stă+16] Ştefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. "Concepts, Operations and Feasibility of a Projection-Based Variation Control Systems". In: *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. IEEE, 2016, pp. 323–333. DOI: 10.1109/ICSME.2016.88.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. German. Springer, 1973. ISBN: 0387811060.

[Ste+09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF — Eclipse Modeling Framework*. Addison-Wesley, 2009. ISBN: 9780321544292.

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. 4th. Addison-Wesley Professional, 2013. ISBN: 0321563840.

[TC06] Kun Tian and Kendra Cooper. "Agile and software product line methods: are they so different?" In: *1st International Workshop on Agile Product Line Engineering, Co-Located with 10th International Conference on Software Product Lines, Baltimore, Maryland, August 21-24, 2006, Proceedings*. IEEE, 2006. DOI: 10.1109/SPLINE.2006.1691593.

[TP11] Rasha Tawhid and Dorina C. Petriu. "Product Model Derivation by Model Transformation in Software Product Lines". In: *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, ISORC Workshops 2011, Newport Beach, CA, USA, March 28-31, 2011*. IEEE, 2011, pp. 72–79. DOI: 10.1109/ISORCW.2011.18.

[Tae+10] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. "Conflict Detection for Model Versioning Based on Graph Modifications". In: *Graph Transformations - 5th International Conference, ICGT 2010, Enschede, The Netherlands, September 27 - October 2, 2010. Proceedings*. Springer, 2010, pp. 171–186. DOI: 10.1007/978-3-642-15928-2_12.

[Tae+14] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. "A Fundamental Approach to Model Versioning Based on Graph Modifications: From Theory to Implementation". In: *Software & Systems Modeling* 13.1 (2014), pp. 239–272. DOI: 10.1007/s10270-012-0248-x.

[Tae04] Gabriele Taentzer. "AGG: A Graph Transformation Environment for Modeling and Validation of Software". In: *Applications of Graph Transformations with Industrial Relevance*. Vol. 3062. Lecture Notes in Computer Science. Charlottesville, VA, USA: Springer, 2004, pp. 446–453. DOI: 10.1007/978-3-540-25959-6_35.

[Thü+12] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. "Family-based Deductive Verification of Software Product Lines". In: *Generative Programming and Component Engineering, GPCE'12, Dresden, Germany, September 26-28, 2012*. ACM, 2012, pp. 11–20. DOI: 10.1145/2371401.2371404.

[Thü+14a] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. "A Classification and Survey of Analysis Strategies for Software Product Lines". In: *ACM Computing Surveys* 47.1 (2014), 6:1–6:45. DOI: 10.1145/2580950.

[Thü+14b] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. "FeatureIDE: An Extensible Framework for Feature-oriented Software Development". In: *Science of Computer Programming* 79 (2014), pp. 70–85. DOI: 10.1016/j.scico.2012.06.002.

[Tha12a] C. Thao. "Managing evolution of software product line". In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE, 2012, pp. 1619–1621. DOI: 10.1109/ICSE.2012.6227224.

[Tha12b] Cheng Thao. "A configuration management system for software product lines". PhD thesis. University of Wisconsin Milwaukee, 2012.

[Tic84] Walter F. Tichy. "The String-to-string Correction Problem with Block Moves". In: *ACM Transactions on Computer Systems* 2.4 (1984), pp. 309–321. DOI: 10.1145/357401.357404.

[Tic85] Walter F. Tichy. "RCS — a System for Version Control". In: *Journal of Software: Practice and Experience* 15.7 (1985), pp. 637–654. DOI: 10.1002/spe.4380150703.

[Tis+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. "On the Use of Higher-Order Model Transformations". In: *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings*. Springer, 2009, pp. 18–33. DOI: 10.1007/978-3-642-02674-4_3.

[Uhr11] Sabrina Uhrig. "Korrespondenzberechnung auf Klassendiagrammen". German. PhD thesis. University of Bayreuth, 2011.

[VG07] Markus Völter and Iris Groher. "Product Line Implementation Using Aspect-Oriented and Model-Driven Software Development". In: *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*. IEEE, 2007, pp. 233–242. DOI: 10.1109/SPLC.2007.28.

[Van08] Robbie Vanbrabant. *Google Guice: Agile Lightweight Dependency Injection Framework*. APress, 2008. ISBN: 1590599977.

[Ves06] Jennifer Vesperman. *Essential CVS*. O'Reilly, 2006. ISBN: 9780596109394.

[Völ+06] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006. ISBN: 9780321544292.

[Völ+13] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. 2013. ISBN: 978-1-4812-1858-0. URL: http://www.dslbook.org.

[WC09] Bernhard Westfechtel and Reidar Conradi. "Multi-variant Modeling - Concepts, Issues and Challenges". In: *Proceedings of the 1st International Workshop on Model-Driven Product Line Engineering (MDPLE 2009)*. CTIT Proceedings, 2009, pp. 57–67.

[WK98] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998. ISBN: 0201379406.

[WMC01] Bernhard Westfechtel, Björn P. Munch, and Reidar Conradi. "A Layered Architecture for Uniform Version Management". In: *IEEE Transactions on Software Engineering* 27.12 (2001), pp. 1111–1133. DOI: 10.1109/32.988710.

[WO14] Eric Walkingshaw and Klaus Ostermann. "Projectional editing of variational software". In: *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014*. ACM, 2014, pp. 29–38. DOI: 10.1145/2658761.2658766.

[Wes14] Bernhard Westfechtel. "Merging of EMF models - Formal foundations". In: *Software & Systems Modeling* 13.2 (2014), pp. 757–788. DOI: 10.1007/s10270-012-0279-3.

[Wes91] Bernhard Westfechtel. "Structure-Oriented Merging of Revisions of Software Documents". In: *Proceedings of the 3rd International Workshop on Software Configuration Management, Trondheim, Norway, June 12-14, 1991*. ACM, 1991, pp. 68–79. DOI: 10.1145/111062.111071.

[Whi+09] Jon Whittle, Praveen Jayaraman, Ahmed Elkhodary, Ana Moreira, and João Araújo. "MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation". In: *Transactions on Aspect-Oriented Software Development*. Vol. 5560. Lecture Notes in Computer Science. Springer, 2009, pp. 191–237. DOI: 10.1007/978-3-642-03764-1_6.

[Wil+17] David Wille, Tobias Runge, Christoph Seidl, and Sandro Schulze. "Extractive Software Product Line Engineering Using Model-based Delta Module Generation". In: *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2017, Eindhoven, Netherlands, February 1-3, 2017*. ACM, 2017, pp. 36–43. DOI: 10.1145/3023956.3023957.

[Wim+11] Manuel Wimmer, Andrea Schauerhuber, Gerti Kappel, Werner Retschitzegger, Wieland Schwinger, and Elizabeth Kapsammer. "A Survey on UML-based Aspect-oriented Design Modeling". In: *ACM Computing Surveys* 43.4 (2011), 28:1–28:33. DOI: 10.1145/1978802.1978807.

[Wir08] Niklaus Wirth. "A Brief History of Software Engineering". In: *Annals of the History of Computing, IEEE* 30.3 (2008), pp. 32–39. DOI: 10.1109/MAHC.2008.33.

[Wir71] Niklaus Wirth. "Program Development by Stepwise Refinement". In: *Communications of the ACM* 14.4 (1971), pp. 221–227. DOI: 10.1145/362575.362577.

[Wu+90] Sun Wu, Udi Manber, Gene Myers, and Webb Miller. "An O(NP) Sequence Comparison Algorithm". In: *Information Processing Letters* 35.6 (1990), pp. 317–323. DOI: 10.1016/0020-0190(90)90035-V.

[XS05] Zhenchang Xing and Eleni Stroulia. "UMLDiff: an algorithm for object-oriented design differencing". In: *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*. ACM, 2005, pp. 54–65. DOI: 10.1145/1101908.1101919.

[ZHJ04] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. "Towards a UML Profile for Software Product Lines". In: *Software Product-Family Engineering*. Vol. 3014. Lecture Notes in Computer Science. Springer, 2004, pp. 129–139. DOI: 10.1007/978-3-540-24667-1_10.

[ZJ07] Tewfik Ziadi and Jean-Marc Jézéquel. "PLiBS: an Eclipse-based tool for Software Product Line Behavior Engineering". In: *Proceedings of the 3rd Workshop on Managing Variability for Software Product Lines, SPLC 2007, Kyoto, Japan, 2007*. HAL/INRIA, 2007.

[ZS97] Andreas Zeller and Gregor Snelting. "Unified Versioning Through Feature Logic". In: *ACM Transactions on Software Engineering and Methodology* 6.4 (1997), pp. 398–441. DOI: 10.1145/261640.261654.

[Zsc+10] Steffen Zschaler, Pablo Sánchez, João Santos, Mauricio Alférez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo, and Uirá Kulesza. "VML* — A Family of Languages for Variability Management in Software Product Lines". In: *Software Language Engineering*. Vol. 5969. Lecture Notes in Computer Science. Springer, 2010, pp. 82–102. DOI: 10.1007/978-3-642-12107-4_7.

Thesis-Related Publications

[SBW15] Felix Schwägerl, Thomas Buchmann, and Bernhard Westfechtel. "SuperMod – A Model-Driven Tool that Combines Version Control and Software Product Line Engineering". In: *ICSOFT-PT 2015 - Proceedings of the 10th International Conference on Software Paradigm Trends, Colmar, Alsace, France, 20-22 July, 2015.* SCITEPRESS, 2015, pp. 5–18. DOI: 10.5220/0005506600050018.

[SBW16a] Felix Schwägerl, Thomas Buchmann, and Bernhard Westfechtel. "Filtered Model-Driven Product Line Engineering with SuperMod: The Home Automation Case". In: *Software Technologies*. Vol. 586. Communications in Computer and Information Science. Springer, 2016, pp. 19–41. DOI: 10.1007/978-3-319-30142-6_2.

[SW16a] Felix Schwägerl and Bernhard Westfechtel. "Collaborative and Distributed Management of Versioned Software Product Lines". In: *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 2: ICSOFT-PT, Lisbon, Portugal, July 24 - 26, 2016*. SCITEPRESS, 2016, pp. 83–94. DOI: 10.5220/0005971300830094.

[SW16b] Felix Schwägerl and Bernhard Westfechtel. "SuperMod: tool support for collaborative filtered model-driven software product line engineering". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. ACM, 2016, pp. 822–827. DOI: 10.1145/2970276.2970288.

[SW17a] Felix Schwägerl and Bernhard Westfechtel. "A Consistency-Preserving Editing Model for Dynamic Filtered Editing of Model-Driven Product Lines". In: *Software Technologies*. Communications in Computer and Information Science. To appear. Springer, 2017.

[SW17b] Felix Schwägerl and Bernhard Westfechtel. "Maintaining Workspace Consistency in Filtered Editing of Dynamically Evolving Model-Driven Software Product Lines". In: *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, Porto, Portugal, February 19-21, 2017.* SCITEPRESS, 2017, pp. 15–28. DOI: 10.5220/0006071800150028.

[SW17c] Felix Schwägerl and Bernhard Westfechtel. "Perspectives on Combining Model-driven Engineering, Software Product Line Engineering, and Version Control". In: *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2017, Eindhoven, Netherlands, February 1-3, 2017*. ACM, 2017, pp. 76–83. DOI: 10.1145/3023956.3023969.

[SW18] Felix Schwägerl and Bernhard Westfechtel. "Integrated Revision and Variation Control for Evolving Model-Driven Software Product Lines". In: *Software & Systems Modeling* (2018). Under Review.

[Schwä+15] 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: *MODELSWARD 2015 - Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, Angers, France, 9-11 February, 2015.* SCITEPRESS, 2015, pp. 5–18. DOI: 10.5220/0005195000050018.

[Schwä+16] Felix Schwägerl, Thomas Buchmann, Sabrina Uhrig, and Bernhard Westfechtel. "Realizing a Conceptual Framework to Integrate Model-Driven Engineering, Software Product Line Engineering, and Software Configuration Management". In: *Model-Driven Engineering and Software Development*. Vol. 580. Communications in Computer and Information Science. Springer, 2016, pp. 21–44. DOI: 10.1007/978-3-319-27869-82.

Further Publications Co-Authored

[BS12a] Thomas Buchmann and Felix Schwägerl. "Ensuring well-formedness of configured domain models in model-driven product lines based on negative variability". In: *4th International Workshop on Feature-Oriented Software Development, FOSD '12, Dresden, Germany - September 24 - 25, 2012*. ACM, 2012, pp. 37–44. DOI: 10.1145/2377816.2377822.

[BS12b] Thomas Buchmann and Felix Schwägerl. "FAMILE: Tool support for evolving model-driven product lines". In: *Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications*. Technical University of Denmark (DTU), 2012, pp. 59–62.

[BS13] Thomas Buchmann and Felix Schwägerl. "Using Meta-code Generation to Realize Higher-order Model Transformations". In: *ICSOFT 2013 - Proceedings of the 8th International Joint Conference on Software Technologies, Reykjavík, Iceland, 29-31 July, 2013*. SCITEPRESS, 2013, pp. 536–541. DOI: 10.5220/0004522305360541.

[BS14] Thomas Buchmann and Felix Schwägerl. "A Model-Driven Approach to the Development of Heterogeneous Software Product Lines". In: *Proceedings of the 9th International Conference on Software Engineering Advances (ICSEA 2014), Nice, France, October 12-16, 2014*. IARIA, 2014, pp. 300–308.

[BS15a] Thomas Buchmann and Felix Schwägerl. "Developing Heterogeneous Software Product Lines with FAMILE — a Model-Driven Approach". In: *International Journal on Advances in Software* 8 (2015), pp. 232 –246.

[BS15b] Thomas Buchmann and Felix Schwägerl. "On A-posteriori Integration of Ecore Models and Hand-written Java Code". In: *ICSOFT-PT 2015 - Proceedings of the 10th International Conference on Software Paradigm Trends, Colmar, Alsace, France, 20-22 July, 2015.* SCITEPRESS, 2015, pp. 95–102. DOI: 10.5220/0005552200950102.

[BS16a] Thomas Buchmann and Felix Schwägerl. "A repair-oriented approach to product consistency in product lines using negative variability". In: *Computer Science — Research and Development* (2016). Online first. DOI: 10.1007/s00450-016-0329-0.

[BS16b] Thomas Buchmann and Felix Schwägerl. "Advancing Negative Variability in Model-Driven Software Product Line Engineering". In: *Evaluation of Novel Approaches to Software Engineering*. Vol. 703. Communications in Computer and Information Science. Springer, 2016, pp. 1–26. DOI: 10.1007/978-3-319-56390-9_1.

[BS16c] Thomas Buchmann and Felix Schwägerl. "Breaking the Boundaries of Meta Models and Preventing Information Loss in Model-Driven Software Product Lines". In: *ENASE 2016 - Proceedings of the 11th International Conference on Evaluation of Novel Approaches to Software Engineering, Rome, Italy, 27-28 April, 2016.* SCITEPRESS, 2016, pp. 73–83. DOI: 10.5220/0005789100730083.

[GSW17] Sandra Greiner, Felix Schwägerl, and Bernhard Westfechtel. "Realizing Multi-variant Model Transformations on Top of Reused ATL Specifications". In: *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, Porto, Portugal, February 19-21, 2017*. SCITEPRESS, 2017, pp. 362–373. DOI: 10.5220/0006137803620373.

[SBW16b] Felix Schwägerl, Thomas Buchmann, and Bernhard Westfechtel. "Multi-Variant Model Transformations — A Problem Statement". In: *ENASE 2016 - Proceedings of the 11th International Conference on Evaluation of Novel Approaches to Software Engineering, Rome, Italy, 27-28 April, 2016*. SCITEPRESS, 2016, pp. 203–209. DOI: 10.5220/0005878702030209.

[SUW13a] Felix Schwägerl, Sabrina Uhrig, and Bernhard Westfechtel. "Demonstration of a Tool for Consistent Three-way Merging of EMF Models". In: *Proceedings of the Joint Track "Tools, Demos and Posters" of ECOOP, ECSA and ECMFA, Montpellier, France, 2013*. Technical University of Denmark (DTU), 2013, pp. 26–28.

[SUW13b] Felix Schwägerl, Sabrina Uhrig, and Bernhard Westfechtel. "Model-based Tool Support for Consistent Three-way Merging of EMF Models". In: *Proceedings of the workshop on ACadeMics Tooling with Eclipse, ACME@ECOOP 2013, Montpellier, France, July 2, 2013*. ACM, 2013, 2:1–2:10. DOI: 10.1145/2491279.2491281.

[SUW15] Felix Schwägerl, Sabrina Uhrig, and Bernhard Westfechtel. "A graph-based algorithm for three-way merging of ordered collections in EMF models". In: *Science of Computer Programming* 113.1 (2015), pp. 51–81. DOI: 10.1016/j.scico.2015.02.008.

[Schwä12] Felix Schwägerl. "Mapping-basierte Modellierung von Softwareproduktlinien". German. MA thesis. University of Bayreuth, 2012. URL: https://epub.uni-bayreuth.de/234/.

[US13] Sabrina Uhrig and Felix Schwägerl. "Tool Support for the Evaluation of Matching Algorithms in the Eclipse Modeling Framework". In: *MODELSWARD 2013 - Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development, Barcelona, Spain, 19 - 21 February, 2013*. SCITEPRESS, 2013, pp. 101–110. DOI: 10.5220/0004310801010110.

Abbreviations

ADL	Architectural Description Language
AE	Application Engineering
AGG	Attributed Graph Grammar
AHEAD	Algebraic Hierarchical Equations for Application Design
Alf	Action language for foundational UML
AMOR	Adaptable MOdel veRsioning
AOM	Aspect-Oriented Modeling
AOP	Aspect-Oriented Programming
API	Application Programming Interface
AST	Abstract Syntax Tree
ATL	Atlas Transformation Language
CASE	Computer Aided Software Engineering
CDO	Connected Data Objects
CFG	Context-Free Grammar
CIDE	Colored IDE (see IDE)
CIM	Component Independent Model
CoV	Change-oriented Versioning
CSV	Comma-Separated Values
CVS	Concurrent Versions System
DAG	Directed Acyclic Graph
DE	Domain Engineering
DFE	Dynamic Filtered Editing (see FE)
DM	Domain Model
DSL	Domain-Specific Language
DSML	Domain-Specific Modeling Language
DVCS	Distributed Version Control System (see VCS)
EBNF	Extended Backus-Naur Form
ECCO	Extraction and Composition for Clone-and-Own
EMF	Eclipse Modeling Framework
EMOF	Essential MOF (see MOF)
FC	Feature Configuration
FDD	Feature-Driven Development
FE	Filtered Editing
FM	Feature Model
FOP	Feature-Oriented Programming
FOSD	Feature-Oriented Software Development
GMF	Graphical Modeling Framework
GQM	Goal Question Metric
HAS	Home Automation System
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IHLV	Integrated Historical and Logical Versioning
JET	Java Emitter Templates
JVM	Java Virtual Machine
LCS	Longest Common Subsequence
M2M	Model-to-Model

M2T	Model-to-Text
MATA	Modeling Aspects using a Transformational Approach
MDA	Model-Driven Architecture
MDSE	Model-Driven Software Engineering
MDSPLE	Model-Driven Software Product Line Engineering
MOD2-SCM	MODular MODel-driven Software Configuration Management
MODPL	MOdel-Driven Product Lines
MOF	Meta Object Facility
MOFM2T	MOF Model-to-Text (see MOF, M2T)
MVDM	Multi-Version Domain Model
MVFS	Multi-Version File System
MVPE	Multi-Version Personal Editor
MPS	Meta Programming System
OCL	Object Constraint Language
OMG	Object Management Group
OOP	Object-Oriented Programming
OSGi	Open Services Gateway initiative
OVM	Orthogonal Variability Model
PEoPL	Projectional Editing of Product Lines
PIM	Platform-Independent Model
PLiBS	Product Line Behavioral Synthesis
PLUS	Product Line UML based Software Engineering
PSM	Platform-Specific Model
QVT	Queries/Views/Transformations
RCS	Revision Control System
REST	REpresentational State Transfer
RPC	Remote Procedure Call
RUP	Rational Unified Process
SCCS	Source Code Control System
SCM	Software Configuration Management
SFE	Static Filtered Editing (see FE)
SHA	Secure Hash Algorithm
SISD	Stepwise and Incremental Software Development
SQL	Structured Query Language
SPL	Software Product Line
SPLE	Software Product Line Engineering
SPLVC	Software Product Line Version Control
SSL	Secure Sockets Layer
TODAG	Totally Ordered Directed Acyclic Graph
UML	Unified Modeling Language
UVM	Uniform Version Model
UUID	Universally Unique IDentifier
VC	Version Control
VCS	Version Control System
WAR	Web ARchive
XMI	XML Metadata Interchange (see XML)
XML	eXtensible Markup Language
XP	eXtreme Programming

Index

Acknowledgments

Probably nobody has ever written a Ph.D. thesis without the help of others—neither did I. With the last lines of my thesis, I would like to spread some words of gratitude, which cannot compensate for all the support I received and all the things I learned whatsoever.

To begin with, from the first day of my occupation as scientific assistant, Monika Glaser and Bernd Schlesier ensured that my organizational and technical needs were always fulfilled. The predecessor in my position, Alexander Dotor, my former colleague Sabine Winetzhammer, and my mentors and co-authors of many papers, Sabrina Uhrig and Thomas Buchmann, encouraged me to apply for the position and to pursue a Ph.D. It has also been a pleasure to work with Sandra Greiner and Nikita Dümmel.

Many Bachelor and Master students who worked with related topics also deserve to be mentioned: Lisa Hartl, Heiko Bächmann, Daniel Kober, Sebastian Petter, Philipp Schmidt, Stefan Schill, Alexander Zimmer, Johannes Schröpfer and Marco Dmitrow.

I had the chance to meet many people who inspired my research and whose feedback I really appreciate. These include the anonymous reviewers of many conference and journal papers, but also discussions after conference talks and off-line conversations. I heard so many insightful talks at diverse scientific events. To only mention a few persons, I list the participants of an informal yet productive meeting about variation control systems that took part in early 2017 in Eindhoven: Andrzej Wąsowski, Eric Walkingshaw, Thorsten Berger, Ştefan Stănciulescu, Timo Kehrer, Christoph Seidl, and Ina Schaefer.

Precious support came from my Ph.D. advisors and reviewers. Since July 2012, I have been a scientific assistant at the chair of Software Engineering led by Bernhard Westfechtel. From the beginning, he gave me important inspiration and feedback, while allowing me to choose my research topics according to my own interests. Furthermore, with his earlier work on the Uniform Version Model, my "doctor father" inherited to me an important piece of theoretical groundwork. I particularly appreciate the various and detailed comments I received in many review sessions. With Sven Apel, one of the most prominent researchers on software product lines joined the endeavor as co-supervisor. During my one-day visit at his chair of Software Engineering at the University of Passau, he helped me narrow down and identify the strengths of my own work. Later on, he gave me important feedback concerning the initial drafts of the evaluation chapter. Gratefully, Andy Schürr, leader of the Real-Time Systems group at TU Darmstadt and inventor of the triple graph grammars, consented to provide a third evaluation of my thesis. His valuable and detailed comments helped me fine-tuning the text.

I would like to mention three of my former fellow students still connected to me by friendship: Lutz Lukas, Johannes Völkel, and Lars Ackermann.

Family is priceless. My parents supported me unconditionally during school and encouraged me to take up computer science studies. Besides all the financial and motivating support, they helped me find out what really is important in life. After all, it was during my Ph.D. time that I married Andrea, who believed in my success from the beginning and gave me moral uplift after some of the more demanding days of work.

To everybody I listed here – but also to those I could not mention by name – *Thanks a Million*! I am happy to have you as supervisor, colleague, friend, or family.