



PhD-FSTC-2017-39

The Faculty of Sciences, Technology and Communication

## DISSERTATION

Defense held on 21/07/2017 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

**Benjamin BEHRINGER**

Born on 9 July 1984 in Essen (Germany)

## PROJECTIONAL EDITING OF SOFTWARE PRODUCT LINES— THE PEOPLE APPROACH

### Dissertation defense committee

**Dr Steffen Rothkugel**, Dissertation Supervisor  
*Associate Professor, Université du Luxembourg*

**Dr Thorsten Berger**  
*Assistant Professor, University of Gothenburg and Chalmers University of Technology*

**Dr Denis Zampunieris**, Chairman  
*Professor, Université du Luxembourg*

**Dr Ina Schäfer**  
*Professor, Technische Universität Braunschweig*

**Dr Martina Lehser**, Vice Chairman  
*Professor, Hochschule für Technik und Wirtschaft des Saarlandes (host institution)*



## ABSTRACT

---

The features of a software product line—a portfolio of system variants—can be realized using various variability implementation techniques. Each technique represents a feature’s software artifacts (a.k.a. feature artifacts) differently, typically classified into annotative and modular variability representations, each with distinct advantages and disadvantages. Annotative representations, such as C preprocessor annotations, are easy to apply, but clutter source code and hinder program comprehension. Modular representations, such as feature modules, support comprehension, but are difficult to realize. Most importantly, to engineer feature artifacts, developers need to choose one representation and adhere to it for evolving and maintaining the same artifacts.

We present the approach PEOPL (Projectional Editing of Product Lines), which combines the advantages of different variability representations. When engineering a feature artifact, developers can choose the most-suited representation for the given task, switch representations on demand, and even use different representations in parallel. PEOPL relies on separating a product line into an internal and external representation, the latter by providing editable projections used by the developers.

In summary, we contribute a programming-language-independent internal representation of variability, seven editable projections reflecting different variability representations (e.g., annotative, modular, variant-specific, reuse-specific), facilities for modeling, analyzing and managing variability, a supporting IDE, and a tailoring of PEOPL to Java and fault trees.

We evaluate PEOPL using three different methodologies. First, we classify PEOPL using a well-defined set of quality criteria along with other implementation techniques, finding that PEOPL provides a novel engineering flexibility, and that the approach is indeed desirable. Second, we evaluate PEOPL’s practicality, scalability, and flexibility in eight Java-based product lines, finding that all can be realized, projections are feasible, and that variant computation is fast (<45ms on average for our largest subject Berkeley DB). Third, to learn more about PEOPL’s practicality and usability, we conduct two pilot user studies, finding that the approach is usable, and that switching representations is useful. We conclude that PEOPL is ready for real users.

## ACKNOWLEDGEMENTS

---

Pursuing a PhD was a challenging and exciting endeavour that would not have been possible without the people who supported me in many different ways.

While the PEOPL approach is my own original idea, realizing a proof of concept of PEOPL's size and complexity required the help and dedication of other people. So, first and foremost, I want to thank my friends in the PEOPL team. I thank Jochen Palz, who contributed substantially to PEOPL's implementation. Jochen was always supportive during my numerous I-have-a-new-nice-idea-let's-implement-it-now episodes and spent sleepless nights with me over fixing this and that, especially, when a deadline was close. I also thank Moritz Fey, who joined the PEOPL team as a bachelor's student and contributed to several parts of PEOPL's implementation and the analysis of data ever since (e.g., pilot user studies). Moreover, I thank Thomas Mechenbier, who helped me with the variational fault tree implementation.

I also want to thank Thorsten Berger for his highly valuable support, feedback, and guidance. Thorsten was always open for my ideas and substantially helped me shape my research. His commitment to details influenced the way I think and deal with problems. It is always a great pleasure to collaborate with Thorsten.

I want to express my deepest gratitude to Martina Lehser for her versatile support. Martina encouraged me to pursue a PhD, provided me with an excellent environment, guaranteed me unrestricted freedom, and supported me financially to build the PEOPL group. It was a great experience to be able to explore and find an interesting research topic on my own in such a supportive environment.

I am very grateful to Steffen Rothkugel for giving me the chance to pursue my PhD in his group. Although our collaboration was mainly remote, Steffen always found the time to discuss my research with me. I admire his structured thinking, presentation skills and work ethics. Steffen had a great impact on the researcher I am today.

Moreover, I thank Nicolas Navet, who was one of my CET members. Nicolas gave me valuable feedback on my research ideas. I would also like to thank Ina Schäfer and Denis Zampunieris for joining my dissertation defense committee.

I am grateful to all the colleagues at the htw saar and the University of Luxembourg with whom I had the pleasure to collaborate and discuss my research, including Eric Wagner, Christoph Karls, Michael Sauer, Sarah Theobald, Mario Korherr, Jean Botev, Laurent Kirsch, and Johannes Klein. Moreover, I thank all students I had the pleasure to work with during my lectures and user studies. Their feedback

was very valuable to me. I decided not to list names hoping that the students in question will know.

I also like to thank all the participants of the 2016 FOSD meeting, which was held at the IT University of Copenhagen. It was a very fruitful event that helped me initiate new collaborations and eventually shape my work.

I thank Maria Rupprecht for taking the time to proofread my dissertation.

Last but not least, I want to thank Ele, who made numerous sacrifices and often saw me hunched over my work. “As distance tests a horse’s strength, so time reveals a person’s heart”—Chinese proverb.

# CONTENTS

---

<b>I</b>	<b>THE (IN)FLEXIBILITY OF VARIABILITY ENGINEERING</b>	<b>1</b>
1	INTRODUCTION	2
1.1	PEoPL Overview . . . . .	3
1.2	Contributions . . . . .	5
1.3	Reading Guide and Outline . . . . .	7
1.4	Publications . . . . .	9
2	ENGINEERING VARIABILITY IN CLASSICAL APPROACHES	10
2.1	Process of Engineering and Managing Variability . . .	10
2.2	Configuring Variability . . . . .	11
2.3	Modeling Variability . . . . .	12
2.4	Implementing Variability . . . . .	14
2.4.1	Overview of Implementation Approaches . .	14
2.4.2	Annotative Representations . . . . .	16
2.4.3	Modular Representations . . . . .	22
2.4.4	Variant Representations . . . . .	28
2.5	Challenges Caused by Variability . . . . .	29
2.5.1	Challenge I: Implementation . . . . .	29
2.5.2	Challenge II: Comprehension . . . . .	31
2.5.3	Challenge III: Maintenance and Evolution . .	35
2.6	Concluding Remarks . . . . .	36
<b>II</b>	<b>ENGINEERING AND MANAGING VARIABILITY IN PEoPL</b>	<b>37</b>
3	IMPLEMENTING VARIABILITY: PEoPL'S FLEXIBILITY	38
3.1	PEoPL's Core Idea: An Overview . . . . .	38
3.2	PEoPL's Benefits: Addressing the Challenges . . . . .	39
3.3	Projections I: Variability Representations . . . . .	41
3.3.1	Textual Annotation Projection . . . . .	42
3.3.2	Visual Annotation Projection . . . . .	42
3.3.3	Feature Module Projection . . . . .	45
3.3.4	Blending Annotation and Module Projections	45
3.3.5	Variant Projection . . . . .	46
3.3.6	Fade-in Feature Module Projection . . . . .	47
3.3.7	Reuse Projection . . . . .	48
3.4	Projections II: Non-Code Feature Artifacts . . . . .	50
3.4.1	Variational Mathematical Formulas . . . . .	50
3.4.2	Variational Fault Trees . . . . .	51
3.5	Projections III: Variability-aware File Explorers . . . . .	57
3.6	Concluding Remarks . . . . .	58
4	MODELING AND MANAGING VARIABILITY	59
4.1	Product Line Setup and Variant Derivation . . . . .	59
4.1.1	Simple Product Line Declaration . . . . .	59

4.1.2	Advanced Product Line Declaration . . . . .	60
4.2	Product Line Correctness . . . . .	63
4.2.1	Artifact-related Feature Dependency Analysis . . . . .	63
4.2.2	Variant-based Data-Flow Analysis . . . . .	64
4.3	Concluding Remarks . . . . .	65
5	EVALUATION I: THE PRODUCT LINE ENGINEER'S VIEW . . . . .	66
5.1	Classification I: Comparing Techniques and Tools . . . . .	66
5.1.1	Preplanning Effort . . . . .	67
5.1.2	Adoption . . . . .	68
5.1.3	Variability Implementation Flexibility . . . . .	69
5.1.4	Uniformity . . . . .	70
5.1.5	Granularity . . . . .	71
5.1.6	Modularity . . . . .	71
5.1.7	Feature Traceability . . . . .	72
5.1.8	Information Hiding . . . . .	73
5.1.9	Expressive Power . . . . .	73
5.1.10	Homogeneous Extensions . . . . .	74
5.1.11	Variant Editing . . . . .	75
5.2	Case Studies . . . . .	76
5.2.1	Objectives . . . . .	76
5.2.2	Case Study Subjects . . . . .	76
5.2.3	Investigation . . . . .	77
5.2.4	Threats to Validity . . . . .	80
5.2.5	Lessons Learned . . . . .	81
5.3	Pilot User Studies . . . . .	82
5.3.1	Objectives . . . . .	82
5.3.2	Participants and Study Format . . . . .	83
5.3.3	Material and Tasks . . . . .	84
5.3.4	Method and Analysis . . . . .	86
5.3.5	Results . . . . .	86
5.3.6	Threats to Validity . . . . .	91
5.3.7	Interpretation and Lessons Learned . . . . .	92
5.3.8	Future Research Questions . . . . .	95
5.4	Concluding Remarks . . . . .	95
III	REALIZING PEOP L USING LANGUAGE ENGINEERING . . . . .	96
6	PEOP L'S TOOLING AND ARCHITECTURE . . . . .	97
6.1	Why and How PEOPL Uses a Language Workbench . . . . .	97
6.2	The MPS Language Workbench . . . . .	98
6.2.1	Structure . . . . .	99
6.2.2	Editor . . . . .	101
6.2.3	Generator . . . . .	102
6.3	An Overview on PEOPL's Architecture . . . . .	102
6.3.1	Internal Variability Representation . . . . .	104
6.3.2	External Variability Representation . . . . .	104
6.3.3	Variability Management Facilities . . . . .	104

6.3.4	Extras . . . . .	104
7	PEOPL'S INTERNAL VARIABILITY REPRESENTATION	105
7.1	CoreVar's Language Structure . . . . .	105
7.1.1	Variability Formalism . . . . .	105
7.1.2	Variational AST Formalism . . . . .	106
7.1.3	Language Structure of Variational ASTs . . . . .	108
7.1.4	Implementation in MPS . . . . .	110
7.2	CoreVar's Generic Editing Operations . . . . .	112
7.2.1	Assign Variability Operation . . . . .	112
7.2.2	Assign Wrapper Variability Operation . . . . .	112
7.2.3	Assign Alternative Operation . . . . .	113
7.2.4	Remove Variability Operation . . . . .	113
7.2.5	Select and Appear Operation . . . . .	113
7.3	CoreVar's Variant Derivation Facilities . . . . .	114
7.3.1	Composition of Feature Modules . . . . .	114
7.3.2	Remove or Replace Placeholders . . . . .	115
7.3.3	Remove Variability from the Variational AST . . . . .	115
7.4	CoreVar's Tailoring Infrastructure and DSLs . . . . .	116
7.4.1	Annotatable Nodes Declaration . . . . .	116
7.4.2	Wrapper Declaration . . . . .	118
7.4.3	Further Declarations . . . . .	118
7.5	Tailoring CoreVar to a Target Language . . . . .	119
7.5.1	Tailoring CoreVar to Java . . . . .	119
7.5.2	Tailoring CoreVar to Fault Trees . . . . .	121
7.6	Concluding Remarks . . . . .	121
8	PEOPL'S EXTERNAL REPRESENTATIONS	123
8.1	Implementation Facilities . . . . .	123
8.1.1	Rendering Annotations . . . . .	123
8.1.2	Rendering Variants . . . . .	125
8.1.3	Rendering Reuse . . . . .	126
8.1.4	Rendering Modules . . . . .	127
8.1.5	Rendering Fade-in Feature Modules . . . . .	141
8.1.6	Blending Renderings . . . . .	141
8.2	Modeling Facilities . . . . .	141
8.2.1	Expert Product Line Configuration . . . . .	141
8.2.2	Simple Product Line Declaration and Feature Module Selection . . . . .	142
8.2.3	Advanced Product Line Declaration . . . . .	145
8.3	Concluding Remarks . . . . .	145
9	PEOPL'S VARIABILITY MANAGEMENT FACILITIES	148
9.1	Artifact-related Feature Dependency Analysis . . . . .	148
9.1.1	CoreVar: Dependency Extraction Infrastructure	148
9.1.2	JavaVar: Java-specific Dependency Extraction . . . . .	150
9.2	Variant-based Data-Flow Analysis . . . . .	152
9.2.1	Background . . . . .	152
9.2.2	Building Variant-specific Data-Flows . . . . .	153



9.3	Concluding Remarks . . . . .	156
10	EVALUATION II: THE LANGUAGE ENGINEER'S VIEW	157
10.1	Classification II: Tool Realization . . . . .	157
10.1.1	Language Independence . . . . .	157
10.1.2	Variability Representation Independence . . . . .	158
10.2	Effort for Extending PEOPL . . . . .	160
10.2.1	New Target Languages . . . . .	160
10.2.2	New Variability Representations . . . . .	160
11	RELATED WORK	162
11.1	Integration of Variability Representations . . . . .	162
11.2	Parser-based Variability Implementation Techniques . . . . .	163
11.3	Projectional Variability Implementation Techniques . . . . .	164
11.4	Other related approaches . . . . .	165
12	CONCLUSION AND FUTURE WORK	166
	BIBLIOGRAPHY	169

## LIST OF FIGURES

---

Figure 1.1	PEoPL separates internal and external variability representations . . . . .	4
Figure 1.2	Dissertation roadmap showing the main contributions . . . . .	5
Figure 2.1	Engineering and managing SPL variability: An overview . . . . .	11
Figure 2.2	Configuration menu of the Stack SPL . . . . .	12
Figure 2.3	Feature model of the Stack SPL . . . . .	13
Figure 2.4	Implementing features using different variability representations . . . . .	15
Figure 2.5	Excerpt of a CPP-based Stack SPL implementation . . . . .	17
Figure 2.6	Excerpt of a CIDE-based Stack SPL implementation . . . . .	19
Figure 2.7	And-expression logic . . . . .	20
Figure 2.8	Coloring the abstract syntax tree . . . . .	21
Figure 2.9	Excerpt of a FeatureHouse-based Stack SPL implementation . . . . .	23
Figure 2.10	Stack variant derived in FH by composing feature modules . . . . .	24
Figure 2.11	Excerpt of a delta-oriented Stack SPL implementation . . . . .	26
Figure 2.12	Delta-oriented Stack SPL declaration . . . . .	27
Figure 2.13	Editable variant representations . . . . .	28
Figure 2.14	Implementation of fine-grained feature artifacts using the CPP, CIDE and FH . . . . .	30
Figure 3.1	Excerpt of a PEoPL-based Berkeley DB implementation . . . . .	40
Figure 3.2	Visual annotation projection with support for toggling feature information . . . . .	44
Figure 3.3	Proactive variant editor with colored bars . . . . .	46
Figure 3.4	Implementing fine-grained feature artifacts in modules . . . . .	48
Figure 3.5	Feature artifacts can realize heterogeneous and homogeneous extensions . . . . .	49
Figure 3.6	Feature artifact reuse instead of cloning . . . . .	50
Figure 3.7	Annotated C code containing mathematical formulas . . . . .	51
Figure 3.8	Example textual fault tree for a robot with actuators and bumpers . . . . .	53

Figure 3.9	Example graphical fault tree for a robot with actuators and bumpers . . . . .	54
Figure 3.10	Variational fault tree for a robot rendered into different external variability representations . . . . .	56
Figure 3.11	Different variability-aware file explorers supported by PEOPL . . . . .	58
Figure 4.1	The shape of PEOPL’s advanced product line declaration . . . . .	61
Figure 4.2	Advanced product line declaration for the Berkeley DB example . . . . .	62
Figure 4.3	Enhanced visual annotation projection with a removal marker . . . . .	62
Figure 4.4	Example of an artifact-related feature dependency . . . . .	64
Figure 4.5	Example of an variant data-flow issue . . . . .	65
Figure 5.1	Editing activities compared: refactoring vs. rendering . . . . .	70
Figure 5.2	Calculation times for a full variant . . . . .	79
Figure 5.3	Results for the question: “Which projections did you use in PEOPL and how frequently?” . . . . .	87
Figure 5.4	Interaction monitoring results for the usage of shortcuts . . . . .	90
Figure 6.1	Language structure and inheritance of language concepts examples . . . . .	100
Figure 6.2	Projectional editor of the TryStatement language concept . . . . .	101
Figure 6.3	PEOPL’s feature model with the most important languages . . . . .	103
Figure 7.1	Relationships of AST nodes, fragments, and variation points . . . . .	107
Figure 7.2	Placeholders enable developers to reuse AST nodes . . . . .	107
Figure 7.3	CoreVar’s language structure . . . . .	109
Figure 7.4	Realizing PEOPL’s internal variability representation using the MPS language workbench . . . . .	111
Figure 7.5	Algorithm to remove variability from the AST . . . . .	116
Figure 7.6	The shape of the annotatable nodes declaration DSL . . . . .	117
Figure 7.7	The shape of the wrapper declaration DSL . . . . .	118
Figure 7.8	The shape of the feature group declaration DSL . . . . .	118
Figure 7.9	Tailoring CoreVar to Java in JavaVar . . . . .	120
Figure 8.1	Simplified visual annotative projectional editor for fragments . . . . .	124
Figure 8.2	Projectional editor and concrete syntax for colored else-if clauses . . . . .	125

Figure 8.3	Default editor for colored non-optional alternatives . . . . .	125
Figure 8.4	Reactive variant editor (without coloring) . . .	126
Figure 8.5	Simple editor for reuse projections . . . . .	126
Figure 8.6	Enabling modular renderings for Java . . . . .	128
Figure 8.7	AST ordering based on a refinement hierarchy	130
Figure 8.8	AST reordering example I . . . . .	131
Figure 8.9	AST reordering example II . . . . .	133
Figure 8.10	Rendering a method declaration . . . . .	134
Figure 8.11	Moving the original-keyword within a statement list and reordering the tree . . . . .	137
Figure 8.12	Moving the original-keyword into a wrapper and reordering the tree . . . . .	138
Figure 8.13	Moving the original-keyword outside a wrapper and reordering the tree . . . . .	140
Figure 8.14	Expert configuration for the Berkeley DB example . . . . .	142
Figure 8.15	ExpertConfiguration concept and editor . . . .	142
Figure 8.16	SimpleDeclaration concept and editor . . . . .	143
Figure 8.17	Editor rendering modules using its color, name, and dependencies . . . . .	143
Figure 8.18	SimpleVariantConfiguration concept and editor	144
Figure 8.19	AdvancedDeclaration concept and editor . . .	146
Figure 9.1	Basic algorithm for extracting artifact-related feature dependencies . . . . .	149
Figure 9.2	Feature module containment example . . . . .	150
Figure 9.3	Algorithm for Java-specific dependency extractions . . . . .	151
Figure 9.4	Data-flow builder for Java's BlockStatement concept and its JavaVar version . . . . .	153
Figure 9.5	Data-flow analysis of Java's WhileStatement .	154
Figure 9.6	Example DFG variants . . . . .	155
Figure 10.1	Refactoring versus rendering variability representations . . . . .	159

## LIST OF TABLES

---

Table 5.1	Classification of variability implementation techniques from the SPL developer’s perspective	67
Table 5.2	Case study subjects: Java-based product lines realized in PEOPL . . . . .	78
Table 5.3	Method declarations requiring boilerplates in pure modular approaches . . . . .	80
Table 7.1	Example module compositions for Berkeley DB	115
Table 8.1	Reordering table example . . . . .	130
Table 8.2	Yet another reordering table example . . . . .	132
Table 10.1	Classification of variability implementation techniques from the language engineer’s perspective . . . . .	157

Part I

THE (IN)FLEXIBILITY OF VARIABILITY  
ENGINEERING

## INTRODUCTION

---

A *software product line (SPL)* is a customizable portfolio of software systems engineered in a specific application domain, such as telecommunication, automotive or industrial automation [46, 146]. Ideally, the portfolio is constructed in terms of end-user-visible domain abstractions, so-called *features* [30]. These can be selected, for instance by a customer, to initiate an automated process, which reduces the portfolio to a concrete, individual software system (a.k.a. *variant* or *product*) [14]. Since different individual variants are derived from a single portfolio, they typically share common parts. For example, all variants that can be derived from a park assistant SPL may support a common set of distance sensors, but the support for the features *autonomous parking*, *assisted parking*, or even both, is variable and depends on the customer's choices.

As such, engineering an SPL amounts to leveraging the commonalities among variants, while managing the differences (a.k.a. *variabilities*) among them [14]. In doing so, we distinguish two key activities: modeling and implementing variability [14, 159]. Variability modeling reflects the stakeholders' perspective, their requirements and the definition of the individual variants that can be derived from the SPL [14]. In fact, developers capture all their abstract design decisions in a *variability model* by declaring features and dependencies between features. To implement these feature declarations (i.e., to implement variability), many language- and tool-based techniques, so-called *variability mechanisms* have emerged. This includes *variability annotations* [94, 102, 121, 177], *templates* [94], *deltas* [115, 157, 158] or *feature modules* [7, 11, 22, 147]. Each technique provides different mechanisms for constructing a feature's common and variable software artifacts (a.k.a. *feature artifacts*) [14, 30]). Although each technique represents feature artifacts differently, the majority of them can be differentiated into annotative and modular representations, each having their own advantages and disadvantages [7, 159, 198].

Annotative representations—e.g., the *C preprocessor (CPP)*, or the *Colored IDE (CIDE)* [102]—capture all feature artifacts directly in the codebase by wrapping them with annotations (i.e., textual or visual variability markers). Such annotations are easy to apply, but challenge program comprehension [135] by obscuring the structure and data-flows of source code [177]—hampering editing experience and impacting maintenance and evolution negatively [65]. Moreover, developers always see all possible variants, many of which might not be relevant for the current engineering activity.

Modular variability representations—e.g., *AHEAD* [22], and *FeatureHouse* (*FH*) [7]—capture all feature artifacts of a feature in one module. Thus, they facilitate a clear structure of the system and allow engineering features without being distracted by irrelevant ones. Yet, decomposing a system into modules is challenging since it requires finding the right decomposition strategy and since creating modules imposes substantial overhead for fine-grained program extensions.

Although annotative and modular representations are complementary [100, 103, 118, 171], existing approaches for implementing SPL variability typically focus on only one representation. Most importantly, these approaches force developers to choose one representation for developing a feature artifact and to adhere to it for evolving and maintaining this artifact. While refactorings were proposed for switching between annotative and modular representations [103], such refactorings are heavyweight and do not allow to quickly switch the representation for a feature artifact. Ideally, developers could exploit the benefits of different representations on-demand and always flexibly choose the one that suits the current engineering activity. In addition, it is desirable to maintain the possibility to switch to and edit individual variants [178, 205, 206] since implementation complexity can be reduced this way.

## 1.1 PEoPL OVERVIEW

In this dissertation, we present the approach *PEoPL* (*Projectional Editing of Product Lines*), which realizes the desired SPL engineering flexibility. *PEoPL* allows developers to flexibly choose and switch to the best suited among very different variability representations of feature artifacts. Moreover, developers can use these representations in parallel (side-by-side) even for the very same artifact.

The core idea of *PEoPL* is to establish an internal representation of the SPL and separate it from the external representations that developers use. Figure 1.1 illustrates *PEoPL*'s key ideas and concepts. Internally, the feature artifacts are uniformly represented in a variational *abstract syntax tree* (*AST*). Externally, this variational *AST* is represented using different editable projections which developers use to engineer feature artifacts. Any of their editing activities directly changes the underlying *AST*, which immediately updates all projections. We conceive projections showing feature artifacts as (i) textual annotations (`#ifdef`), (ii) visual annotations (colored bars), (iii) feature modules, (iv) annotations blended into modules, (v) variants (i.e., hiding artifacts related to non-selected features), (vi) fine-granular fade-in feature modules, and (vii) reused code snippets (for avoiding feature-related *Type-I code clones* [153, 162]).

Based on our internal variability representation, we also conceive two variability modeling facilities (simple and advanced), which en-



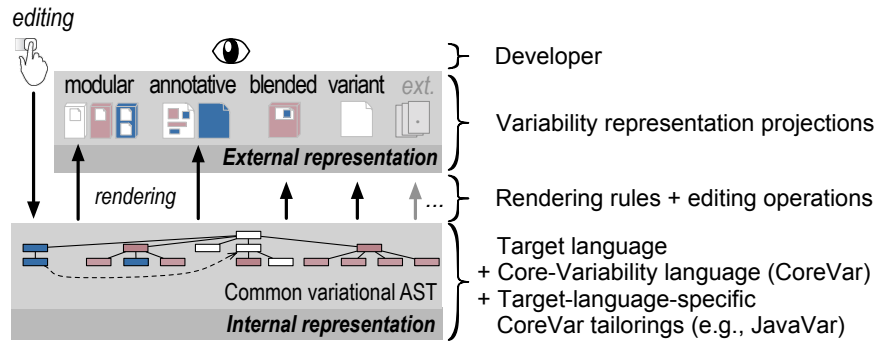


Figure 1.1: PEoPL separates internal and external variability representations

able developers to setup their SPLs and derive concrete, executable variants. Moreover, we provide variability management facilities that enable developers to handle an SPL’s inherent complexity by automating the extraction of dependencies between features and analyzing data-flows of variants.

We show the feasibility of the PEoPL approach by realizing a complete IDE, built upon the language workbench JetBrains *Meta Programming System (MPS)*<sup>1</sup>[145, 202]. The PEoPL IDE realizes our projections and operations for engineering feature artifacts while benefiting from MPS’ common program-editing facilities (e.g., for Java). To evaluate the PEoPL approach and IDE, we use three different methodologies.

First, we use and extend existing classification frameworks to classify PEoPL together with other SPL implementation approaches. Our results show that PEoPL compares well with other SPL implementation approaches while providing a novel SPL engineering flexibility.

Second, we use the PEoPL proof-of-concept IDE to adopt and implement eight Java SPLs. Our largest subject (and running example) is Berkeley DB, an SPL with 70kLOC including 42 features and 218 classes. PEoPL’s expressiveness suffices to realize all SPLs without any workarounds. The evaluation also shows that it is feasible and practical to conceive an internal representation that is projected into very different external representations. Furthermore, our approach scales: all projections can be rendered and edited without introducing significant latencies. For instance, variant editing is smooth, since computing a specific file variant (<1ms on average for all subjects) and calculating all AST nodes included in a variant is quick (<45ms for Berkeley DB on average).

Third, we conduct two explorative pilot user studies to learn more about PEoPL’s practicality and usability. Our results show that the approach is ready for real users and that choosing between multiple representations of feature artifacts is useful.

<sup>1</sup> <https://www.jetbrains.com/mps/>

## 1.2 CONTRIBUTIONS

This dissertation contributes to the domain of software product line engineering research. Figure 1.2 shows a dissertation roadmap providing an overview of our contributions and their relationships.

**CHALLENGES, CLASSIFICATION AND COMPARISON.** Based on the literature, our experience of using PEOPL and other tools, and our observations made in two pilot user studies (Sec. 5.3), we explore three major challenges that arise when being bound to a single concrete variability representation (Sec. 2.5). On this basis, we use and extend existing classification frameworks to compare PEOPL with other SPL implementation approaches [100, 118] (Sec. 5.1). Our results underline the value of a symbiosis of different variability representations.

**INTERNAL VARIABILITY REPRESENTATION.** Given the opportunities of a symbiosis, we unify variability representations using a language-oriented approach. We present a novel variability language called *CoreVar*, enabling a uniform treatment of variability independent of the end-user visible concrete syntax (Ch. 7). In a nutshell, the language concepts of *CoreVar* are used to annotate arbitrary AST

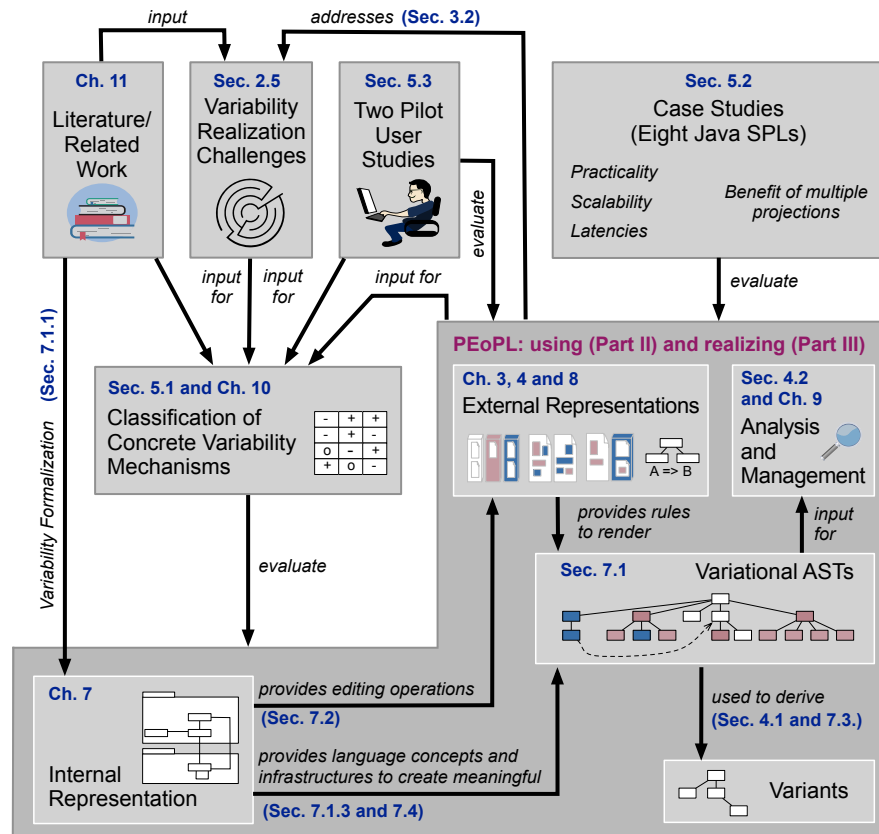


Figure 1.2: Dissertation roadmap showing the main contributions

nodes—creating a variational AST (Sec. 7.1). CoreVar is black-box generic, independent of programming languages and even artifact-types, and thus, not bound to textual representations. With CoreVar, we also introduce a set of generic editing operations to enable developers to manipulate variational ASTs (Sec. 7.2), and an infrastructure to derive variants (Sec. 7.3). For meaningful variational ASTs, CoreVar provides a tailoring infrastructure and *domain-specific languages* (DSLs) supporting the declaration of annotatable language concepts (e.g., Java’s Statement concept) and more (Sec. 7.4). Thus, CoreVar can easily be tailored to specific (programming) languages. In summary, PEOPL relies on composing CoreVar, the concrete target language (e.g., Java), and a language tailoring CoreVar to the target language (cf. Fig. 1.1). We give examples and an implementation to make full Java 1.8 and a fault-tree language variability-aware (Sec. 7.5).

**EXTERNAL VARIABILITY REPRESENTATIONS.** Based on CoreVar, we present the conceptual design and proof of concept for rendering a common, variational AST into different variability representations (Sec. 8.1). We also show that the model can be rendered into different variability modeling facilities (Sec. 8.2). To this end, we provide projections for textual and visual annotations and feature modules, as well as projections that allow blending annotations into modules, reusing elements, and editing variants. Moreover, we show the flexibility of projectional editing by proposing a fade-in module projection which integrates code of external modules to support fine-grained changes. We also present concrete usage scenarios for all projections (i.e., from the developer’s perspective) using Java, mathematical formulas, and fault trees (Ch. 3 and 4). This way, we demonstrate that the approach is, in fact, independent of languages and artifact-types.

**VARIABILITY MANAGEMENT FACILITIES.** To support developers in dealing with an SPL’s inherent complexity (Sec. 4.2 and Ch. 9), we conceive ways for extracting artifact-related feature dependencies (a.k.a. feature artifact dependencies). In fact, based on a variational AST, constraints describing dependencies between features can be extracted automatically from the SPLs implementation without prior static analysis (cf. [137]). Consequently, static declaration and maintenance of such dependencies in a variability model is unnecessary. Moreover, we support developers in performing a variant-based data-flow analysis.

**PRACTICAL APPLICABILITY.** To show the practical applicability of our entire approach, we provide a proof-of-concept implementation with the PEOPL IDE (Ch. 6). Based on the MPS language workbench, we present an industry-strength variability-aware IDE with support for a full GPL (i.e., Java). While we just use the projectional editing technology of MPS for our realization, we provide evidence that our solution scales for rendering different variability representations. To

evaluate the PEOPL approach and IDE, we conduct eight Java case studies with subjects up to 70KLOC (Sec. 5.2). In two pilot user studies, we demonstrate PEOPL’s practical usability, emphasizing that it is ready for real users (Sec. 5.3). We also contribute an online appendix<sup>2</sup> with replication packages and screencasts.

### 1.3 READING GUIDE AND OUTLINE

This dissertation is divided into three parts and twelve chapters. Part **i** contains this introduction and provides the necessary background knowledge. Part **ii** is concerned with engineering and managing variability using PEOPL (i.e., the SPL developer’s perspective). Part **iii** discusses how PEOPL is realized and how it can be used to make languages variability-aware (i.e., the language engineer’s perspective). Notice that Part **ii** and **iii** are closely related, and thus, switching between the two is possible. For instance, it is possible to start reading how projections are used by developers (Ch. 3), and then to move on with the realization details (Sec. 8.1). Notice that before reading the details, it is helpful to get to know PEOPL’s tooling and architecture (Ch. 6), and in particular, PEOPL’s feature model depicted in Figure 6.3. We now provide an overview of the remaining chapters.

#### *Part I: The (In)Flexibility of Variability Engineering*

CHAPTER 2 introduces a running example and the necessary background for understanding this dissertation. To familiarize readers with the key concepts, we discuss different variability modeling approaches and variability representations, and show how to map from modeling artifacts (i.e., features) to implementation artifacts (i.e., feature artifacts). Finally, we discuss the challenges caused by being bound to a single variability representation.

#### *Part II: Engineering and Managing Variability in PEOPL*

CHAPTER 3 presents PEOPL from the SPL developer’s perspective, discussing how it addresses the challenges caused by variability and how it can be used to engineer an SPL. We present how an SPL can be implemented, using different projections (e.g., annotations, modules, variant-specific). Realization details of this chapter are presented in Chapter 7 and Chapter 8.

CHAPTER 4 discusses how an SPL can be set up and managed in PEOPL. We outline how PEOPL helps developers to deal with an SPL’s inherent complexity using feature constraint extraction and variant-

---

<sup>2</sup> <http://www.peopl.de>

based data-flow analysis. Realization details of this chapter are presented in Section 8.2 and Chapter 9.

CHAPTER 5 intends to encourage developers to exploit PEOPL's novel flexibility. All in all, we present how we evaluate PEOPL from the SPL developer's perspective. We show PEOPL's practical applicability, novelty and usability. We catalog concrete popular approaches and compare them with PEOPL. Then, we implement eight Java-based product lines. Finally, we conduct two explorative pilot user studies with students.

*Part III: Realizing PEOPL using Language Engineering*

CHAPTER 6 briefly presents PEOPL's tooling and architecture, and therewith provides a basis for the following chapters. Moreover, we discuss the underlying platform MPS and its key properties necessary to understand the remaining implementation details.

CHAPTER 7 presents PEOPL's internal representation and introduces the programming-language-independent variability language CoreVar. We present CoreVar's variability formalizations, a corresponding language structure, generic editing operations, and variant derivation facilities. We also present how to employ CoreVar to make concrete target languages variability-aware (i.e., Java and fault trees).

CHAPTER 8 presents PEOPL's external representations. We present several variability representation projections and the concrete realization of each in PEOPL. We also present the realization of PEOPL's modeling facilities.

CHAPTER 9 shows how to extract feature constraints from a variational AST, and how to realize the variant-based data-flow analysis.

CHAPTER 10 evaluates PEOPL from the language engineer's perspective comparing language independence and variability representation independence. Moreover, we discuss the effort to realize and extend PEOPL.

CHAPTER 11 provides an overview on closely related work.

CHAPTER 12 concludes the dissertation and provides a discussion on future directions of the PEOPL approach.

## 1.4 PUBLICATIONS

This dissertation builds on and contains material from the following publications:

1. Benjamin Behringer, Jochen Palz, Thorsten Berger. *PEoPL: Projectional Editing of Product Lines*. In Proceedings of the 39th International Conference on Software Engineering (ICSE), pages 563-574. ACM, Mai, 2017.  
(All chapters)
2. Benjamin Behringer, Steffen Rothkugel. *Integrating Feature-based Implementation Approaches using a Common Graph-based Representation*. In Software Engineering Track of the 31st ACM Symposium on Applied Computing (SAC), pages 1504-1511. ACM, April, 2016.  
(Chapter 7)
3. Benjamin Behringer, Moritz Fey. *Implementing Delta-Oriented SPLs using PEoPL: An Example Scenario and Case Study*. In Proceedings of the 7th International Workshop on Feature-Oriented Software Development (FOSD), pages 28-38. ACM, October, 2016.  
(Chapters 3 and 4)
4. Benjamin Behringer. *Integrating Approaches for Feature Implementation*. In Doctoral Symposium Track of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), pages 768-771. ACM, November, 2014.  
(Chapters 1 and 2)
5. Benjamin Behringer, Laurent Kirsch, Steffen Rothkugel. *Separating Features using Colored Snippet Graphs*. In Proceedings of the 6th International Workshop on Feature-Oriented Software Development (FOSD), pages 9-16. ACM, September, 2014.  
(Chapter 1)
6. Benjamin Behringer, Martina Lehser, Steffen Rothkugel. *Towards Feature-Oriented Fault Tree Analysis*. In Proceedings of the 8th IEEE International Workshop on Quality Oriented Reuse of Software (QUORS), pages 522-527. IEEE Computer Society, July, 2014.  
(Chapter 3)

## ENGINEERING VARIABILITY IN CLASSICAL APPROACHES

---

We start by outlining the process of engineering and managing SPL variability (Sec. 2.1). Then, we discuss the idea of configuring variability using a running example (Sec. 2.2). Thereafter, we focus on the variability engineering process and its differentiation into modeling variability and implementing variability. With feature modeling, we present a popular variability modeling approach (Sec. 2.3). Subsequently, we introduce different well-known approaches to implement SPL variability (Sec. 2.4). Finally, we discuss the challenges of implementing variability (Sec. 2.5), and conclude that a flexible implementation approach is missing, yet desirable (Sec. 2.6).

### 2.1 PROCESS OF ENGINEERING AND MANAGING VARIABILITY

Classical software engineering is concerned about single *software products* (a.k.a. *software variants*) that are tailored to a customer's requirements manually. To build different but related software products, for instance for other customers, software engineers typically copy or branch the code to be reused and customized, which is an ad hoc strategy called *clone-and-own* (a.k.a. *grow-and-prune*) [3, 64, 75, 136]. Although engineers can make quick changes in the beginning, evolving and maintaining all cloned variants is a challenging task. For example, it is difficult to fix a bug in a multitude of clones [91].

In contrast, SPL engineering facilitates systematic reuse and automated software customization [14, 46, 146]. An SPL is a portfolio of software variants, which is typically constructed in terms of stakeholder-visible features [14, 30]. From a feature selection, an individual variant can be derived automatically. Hence, the variants can be distinguished by the features they realize. Since variants may share features, commonalities among variants are possible.

As such, engineering SPL variability amounts to engineering features. Figure 2.1 illustrates the engineering process, which is typically distinguished into two major engineering activities: modeling variability and implementing variability (cf. [14, 159]). Modeling variability amounts to constructing a variability model—a specification of features and their relationships on an abstract level. The variability model captures the SPL engineer's scoping decisions—the features that are to be supported by the SPL—as well as all satisfiable feature selections. To realize the scoping decisions in code, developers implement the desired features in terms of concrete feature artifacts. In fact,

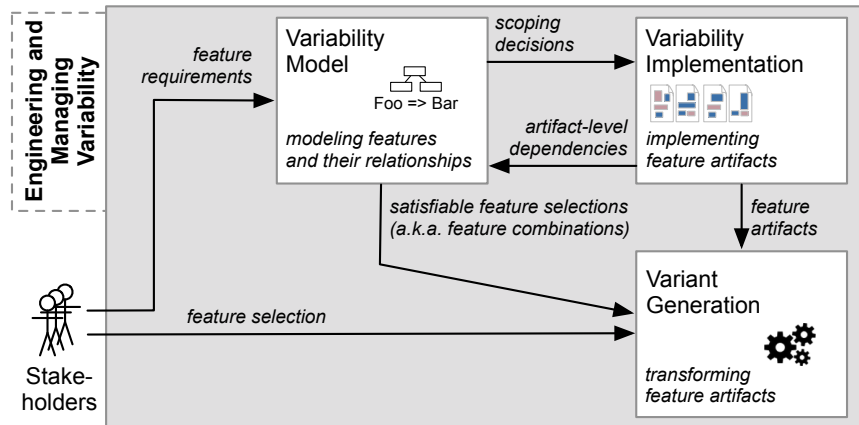


Figure 2.1: Engineering and managing SPL variability: An overview

each feature artifact is implemented by using exactly one of various SPL implementation techniques [14, 159]. Notice that each technique represents a feature artifact differently, and thus, the variability representation differs among these techniques (cf. Sec. 2.4).

Since a feature artifact may use or refer to artifacts of other features, dependencies between features may arise. Such artifact-related feature dependencies are typically extracted into the variability model to maintain the satisfiable feature selections. This variability management is crucial since feature artifacts serve as input for variant generation, which fails if an artifact-related feature dependency is not satisfied. For example, imagine a method call `foobar` being a feature artifact of a feature `Foo`. Now assume that `foobar` refers to a corresponding method declaration which, however, is a feature artifact of a feature `Bar`. Note that we introduced an artifact-related feature dependency. The feature `Foo` depends on `Bar` to function correctly, and thus, variant derivation fails if only `Foo` but not `Bar` is selected. In fact, a dangling reference appears as `foobar` cannot be resolved. That is, no declaration matches the call’s signature.

## 2.2 CONFIGURING VARIABILITY

Throughout this dissertation, we discuss real-life SPL examples taken from our case studies, whenever suitable. In this Chapter, to ease understanding the different ways of engineering variability, we mainly use a Stack SPL, which has been similarly used in previous work by others [67, 100, 102, 147, 162, 165]. Implementing a stack is a common computer science problem, and thus, simple enough to easily acquire domain-knowledge. Yet, it is also complex enough to illustrate different variability representations and their respective strength and weaknesses.

The Stack SPL is a portfolio of stack libraries, each of which is distinguished by the features it realizes. Based on a feature selection,



the Stack SPL can be tailored to a specific stack library. One way to enable feature selections are configuration menus. Figure 2.2 shows an example menu for our Stack SPL. Radio buttons enable choosing either the feature *Array* or *Stack*. The rationale is that the stack's data can be either stored in an array or a list, but not in both at the same time. Thus, stakeholders need to decide between performance (array) and flexibility (list). Moreover, optional features are selectable via checkboxes. For instance, *Synchronization*, which enables accessing the stack from multiple threads, and *Undo*, which allows undoing the last operation—that is, push or pop.

Realizing a configuration menu from scratch is very time consuming. For instance, artifact-related feature dependencies must be implemented manually. In fact, we would need to write an algorithm that (un)checks dependant features in the menu automatically. For instance, to find the target object in the stack, assume that a feature artifact of the feature *Search* uses an iterator defined in the feature *Iterable*. Then, *Search* depends on the feature *Iterable* to function correctly and thus, we need an algorithm that selects *Iterable* automatically when *Search* has been selected. Otherwise, the stack library variant to be derived would be invalid.

In summary, it is desirable to reduce the effort of implementing such menus. Optimally, we could derive the configuration menu and the algorithms to handle dependencies automatically. Variability modeling approaches address this desire.

**Stack Software Product Line: Feature Selection Menu**

---

**Storage:**  Array  List

**Iterable:**  provide means to iterate over the stack elements

**Synchronization:**  support synchronized stack access

---

**Commands**

---

**Clear:**  allow clearing the stack

**Undo:**  allow undoing the last operation

**Search:**  allow searching for an element

**StringBuilder:**  allow building a string from the stack's content

---

**Logging:** log operations to  terminal, and  file

**Test:**  provide several test routines to test the stack library

Figure 2.2: Configuration menu of the Stack SPL

## 2.3 MODELING VARIABILITY

Variability modeling enables engineers to express an SPL's features and their dependencies. Several approaches for modeling variability have been proposed, such as *feature models* [99], *grammars* [21, 51, 95], *propositional formulas* [21], *clafers* [18], and *KConfig models* [129]. We fo-

cus on the widely used feature models since they help understanding this dissertation.

A feature model is a hierarchical configuration menu that represents features in a tree-like structure [99]. As such, individual features have a parent-child relationship. Each feature is either abstract or concrete [185]. Abstract features do not have any feature artifacts as they are typically used to structure the feature model. Moreover, each feature is either optional or mandatory with the latter being selected by default. The children (a.k.a. *sub-features*) of a feature may be grouped in an or-group or an alternative-group. In an optional or-group, none, one, many or all features may be selected. In contrast, in an alternative group, which represents a one-out-of-many relationship, only exactly one of the group’s features may be selected.

Figure 2.3 shows a feature model for our Stack SPL, which conforms to the configuration menu discussed above (Fig. 2.2), while the feature model provides more design details. The abstract feature *Logging* has the concrete sub-features *Terminal* and *File*. The two features are in an or-group—that is, either *Terminal* or *File*, or both features can be selected. Since the feature *Logging* is optional, neither *Terminal* nor *File* must be selected. In fact, there can be variants without logging support. The abstract feature *Storage* in turn is mandatory, and thus, one of its children must be included in a variant to produce a valid stack library. Its sub-features *Array* and *List* are in an alternative-group—that is, only one of the two features can be selected for the stack’s underlying data storage. As a result, either *Array* or *List* must be selected (modeled via radio-buttons in Fig. 2.2).

As detailed above, simple artifact-related feature dependencies may appear in the variability implementation. However, variability modeling is not restricted to expressing such dependencies. In fact, engineers can constrain the selection of features in a more general sense to enforce the correct behavior or “appearance” of variants. The tree-like structure of feature models already provides means to express simple constraints, such as alternative-groups. So-called

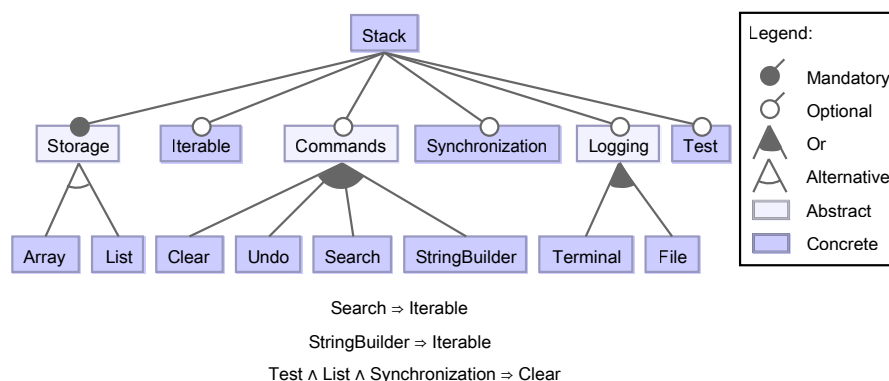


Figure 2.3: Feature model of the Stack SPL

*cross-tree constraints* enable more complex configuration restrictions. A cross-tree constraint is a propositional formula over features, expressing dependencies and relationships between (multiple) features [21]. Notice that our feature model depicted in Figure 2.3 contains three cross-tree constraints, each of which uses an implies-operator ( $\Rightarrow$ ). For instance, selecting the feature *Search* implies that the feature *Iterable* must be selected as well, which is indicated by the constraint  $Search \Rightarrow Iterable$ . The third constraint is also simple. It says, if all three features *Test*, *List* and *Synchronization* are selected, then *Clear* must also be selected.

In summary, a feature model allows expressing complex feature relationships. Moreover, it serves as input for automatically deriving configuration menus and the algorithms handling dependencies.

## 2.4 IMPLEMENTING VARIABILITY

Variability modeling defines features as domain abstractions that describe the functional and non-functional characteristics of an SPL. To realize the desired characteristics, each feature is typically implemented in (multiple) feature artifacts. To implement a feature artifact, developers can choose from various SPL variability implementation approaches. Since each approach represents feature artifacts differently, we start with a brief overview on the different variability representations, followed by a brief description how variants are derived from the respective representations. Then, we take a closer look at annotative, modular and variant representations.

### 2.4.1 Overview of Implementation Approaches

VARIABILITY REPRESENTATIONS. Features can be implemented in various ways, for instance by using *variability annotations* [94, 121, 150, 177], *templates* [94], *aspects* [2, 111], *deltas* [157, 158], or *modules* [7, 11, 22, 147]. Although features are implemented differently, many of these approaches represent a feature's artifacts either as annotations or modules [14, 159]. Some approaches also allow showing the feature artifacts conditionally—that is, representing the SPL's variability in a concrete variant based on a feature selection (cf. Fig. 2.4).

Annotative representations constitute feature artifacts as variability annotations (e.g., CPP and CIDE [102]). Each annotation marks a snippet of the codebase as variable, and thus, the implementation of a feature is scattered across the program (cf. Fig. 2.4a).

Variant representations are typically realized on top of an annotative representation (e.g., version editor [16] and CIDE [104]). They enable developers to edit SPL variants in isolation—that is, editing only the feature artifacts relevant for the current task (cf. Fig. 2.4b). Thus, the SPL's variability is shown in a less complex manner.

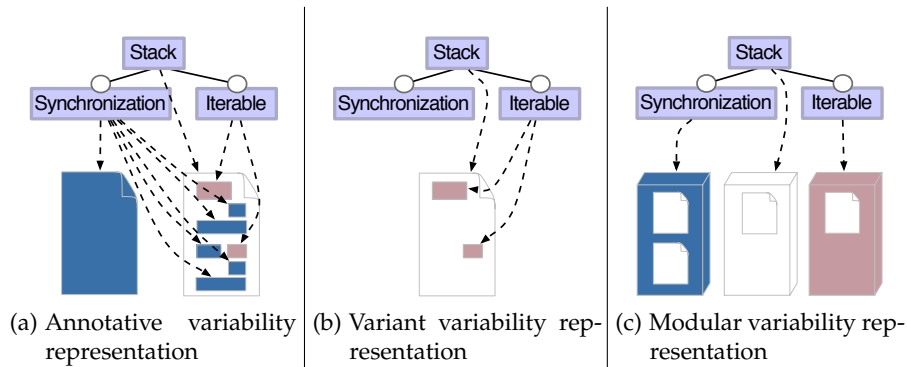


Figure 2.4: Implementing features using different variability representations: features in the variability model (upper halves) and feature artifacts in the variability representation (lower halves)

Modular representations aggregate all feature artifacts of a feature in a distinct cohesive unit (cf. Fig. 2.4c), which is also called *composition unit* [7], *feature module* [99, 147], *delta module* [157, 158], or just *module* [22]. For brevity, we mainly use the term module throughout this dissertation. The key idea is to start with a common base module and use other modules to step-wise introduce new structural elements, and refine or override existing elements [22, 56] (e.g., AHEAD or FH), or even remove existing elements [158] (e.g., *delta modeling approaches* [115, 157, 158]).

On a final note, some approaches represent variability neither pure modular nor pure annotative [7, 97], for instance, *Aspect* [110], *invasive software composition* [15], *explicit programming* [41], *metaprogramming with traits* [149], *FeatureCoPP* [118], and the *compositional choice calculus* [204]. These hybrid approaches, typically combine annotations with feature modules.

**VARIANT DERIVATION.** The core essence of all SPL implementation approaches is the ability to provide a variant from the variability representation. We distinguish compile-time and runtime variability [14]. Approaches supporting the latter skip the content not contained in the variant during runtime. In contrast, approaches supporting compile-time variability physically transform the feature artifacts into a variant. Annotative approaches realize this transformation by removing irrelevant feature artifacts from the SPL during variant derivation (a.k.a. *negative variability* [80, 198]). In contrast, many modular approaches [7, 22] are composition-based. They synthesize feature artifacts to derive a variant (a.k.a. *positive variability* [80, 198]). Some approaches, such as delta modeling, support negative and positive variability, and thus, allow a complex variant restructuring.

On a final note, there are approaches that only support a manual derivation of variants, yet facilitate reuse [97]. Components and services are the most prominent modular examples for non-automated SPL development [20, 122, 146]. The problem with these approaches

is that it is time-consuming to deploy a variant by integrating different components/services manually. In this dissertation, we focus on an automated derivation of variants from a feature selection, and thus omit a detailed discussion of non-automated approaches.

#### 2.4.2 Annotative Representations

As exemplified in Figure 2.4a, annotative approaches implement features using variability markers embedded into a program. These markers are either textual or visual. Next, we discuss a concrete approach of either category, and provide an overview on related work.

##### 2.4.2.1 Textual Annotations: The C Preprocessor

The CPP is the most frequently used and most contested approach to implement textual annotations and variability [65, 66, 121, 177]. It has been used in various software projects [126] with the *Linux kernel* being the most prominent large-scale example. In fact, kernel release 4.8.6<sup>1</sup> comprises over 11,000,000 LOC and around 16,000 features (a.k.a. *configuration options*), most of which can be selected to derive a kernel variant [1, 29], for instance for a router, server, or PC.

The CPP has been originally written for C, but it is a language-independent text-processor. Its textual annotations are called *preprocessor directives*. The directives `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, and `#endif` mark the beginning and the end of a feature artifact. Each directive has a presence condition—typically a Boolean expression over features. Based on a feature selection, the CPP evaluates each presence condition to a Boolean value in a preprocessing step. The content of directives whose presence condition evaluates to *true* is part of the selected variant, content of the ones evaluating to *false* not.

Figure 2.5 shows a CPP implementation excerpt of our Stack SPL. Each concrete feature—declared in the feature model in Figure 2.3—is implemented, using a set of textual annotations. For instance, feature artifacts of the feature *List* appear in two different program locations (Lines 11–14 and 45–46). The corresponding code is only included in the final variant if the feature *List* has been selected, otherwise removed.

Notice that developers use Boolean operators in presence conditions to express the combination of features in various situations [126], for instance, to achieve compliance with the feature model [155], to avoid code duplication, or to implement feature interactions explicitly. Feature interactions are of particular importance in SPL development since they are a cause for subtle errors. Two features interact if they behave well in isolation, but not in a setting with both being se-

---

<sup>1</sup> <http://kernel.org/>

```

1  #ifndef Stack
2  public class Stack
3  #ifndef Iterable
4  implements Iterable<Object>
5  #endif
6  {
7      private int top = 0;
8      #ifndef Array
9      private Object[] dataArray;
10     #elif List
11     private Stack.Node topNode = null;
12     private class Node { ...
13         private Node(Object element, Stack.Node next) { ... }
14     }
15     #endif
16     #ifndef Synchronization
17     private final ReentrantLock lock = new ReentrantLock();
18     private final Condition notEmpty = lock.newCondition();
19     #endif
20     #ifndef Synchronization && Array
21     private final Condition notFull = lock.newCondition();
22     #endif
23     public void push(Object data)
24     #ifndef Array || Synchronization
25     throws
26     #ifndef Array
27     ArrayIndexOutOfBoundsException
28     #endif
29     #ifndef Synchronization
30     , InterruptedException
31     #endif
32     #endif
33     {
34         #ifndef Synchronization
35         lock.lock();
36         try {
37             #ifndef Synchronization && Array
38             while (isFull()) notFull.await();
39             #endif
40             #ifndef Array
41             if (isFull()) throw new ArrayIndexOutOfBoundsException();
42             dataArray[top] = data;
43             #elif List
44             Stack.Node newElement = new Node(data, topNode);
45             topNode = newElement;
46             #endif
47             top++;
48             #ifndef Synchronization
49             notEmpty.signalAll();
50             } finally { lock.unlock(); }
51         #endif
52     }
53     public Object pop() ... { ... }
54     #ifndef Array
55     private boolean isFull() { return top >= dataArray.length; }
56     #endif
57     ...
58 }
59 #endif

```

Figure 2.5: Excerpt of a CPP-based Stack SPL implementation

lected [42, 108, 139]. Thus, to handle interactions, we need code that explicitly coordinates the interaction.

In our CPP example, the features *Array* and *Synchronization* interact. We use the presence condition *Synchronization && Array* to include the required coordination code when both features are selected (Fig. 2.5, Lines 20 and 38). In fact, we add overflow-avoidance to array-based, synchronized stacks—that is, if the stack is full, a thread must wait until there is space on the stack again (Line 39).

Line 24 shows another example for using Boolean operators. The keyword `throws` is only included in variants if either *Array*, *Synchronization* or both features have been selected. The concrete `throws`-items are then included if their respective features are selected (Lines 26 and 29).

Notice that in the Stack SPL's current implementation, we control the mutual exclusion of the features *Array* and *List*—as expressed in the feature model in Figure 2.3—using `#elif` directives, each of which denotes that *Array* has priority over *List* (Lines 8, 10, 41 and 44). That is, both features may be selected, but only the feature artifacts of *Array* will be included in the variant then. To conform with the feature model, we could express the mutual exclusion more explicitly. In fact, we could use `#ifdef Array && !List` instead of `#ifdef Array`, and `#elif !Array && List` instead of `#elif List`. Moreover, we could throw an exception in `#else` blocks, or the Stack's constructor, if both features have been selected: `#ifdef Array && List`.

Not only the support for complex Boolean expressions makes the CPP a very powerful tool, but also the ability to add preprocessor directives to arbitrary program locations. Depending on an annotation's location in the program, the annotation is either disciplined or undisciplined [125]. Similar to C [125], we define for Java that annotations on one or a sequence of entire classifiers (e.g., classes and interfaces), classifier members (e.g., method declarations and inner classes), and statements (e.g., try-statement and block statement) are disciplined, all other annotations are undisciplined. This differentiation is important since undisciplined annotations complicate the development of tools for managing variability [125]. Notice that in practice roughly 16% of all annotations are undisciplined [125], and thus implementation approaches should support them.

In our Stack SPL example, there are several disciplined annotations, such as the annotations of the class `Stack` (Line 1–60), the method declaration `isFull` (Lines 56), or the `while`-statement (Line 39). In contrast, the annotation of the `try`-statement—which wraps other statements—is undisciplined since the annotation does not mark the entire `try`-statement (Lines 34–37) as variable. In other words, the `try`-statement is only partially annotated, since its body is not annotated.

Notice that the discipline of annotations is closely related to the granularity of feature artifacts—that is, the structural level at which a feature artifact is implemented [14]. At the lower end there are fine-grained feature artifacts, such as an undisciplined annotation on a method parameter, an expression, or even a single character. The upper end denotes coarse-grained feature artifacts, such as an disciplined annotation on a file or a Java class. In our Stack example there are several undisciplined, fine-grained feature artifacts, such as the annotations on the `implements-list` (Line 4), and the `throws-items` (Lines 24–32).

## 2.4.2.2 Visual annotations: CIDE

The tool CIDE is a prominent academic preprocessor with support for visual annotations [102]. In CIDE, each feature (declared in a feature model) has a distinct color. The different colors are used in the background of the program’s text to visualize annotations. Figure 2.6 illustrates an excerpt of our Stack SPL, implemented with CIDE. For instance, the class `Stack` is colored gray (Lines 1–28) and thus realizes the feature `Stack`.

Notice that there are no Boolean expressions over features for controlling the SPL’s configuration in a CIDE implementation. In fact, each feature represents a Boolean value during variant derivation. If the feature is selected, the corresponding colored code is included, otherwise pruned. Thus, including a feature artifact in a variant if a feature is not selected requires workarounds in the implementation. Moreover, mutual exclusion cannot be modeled explicitly in the implementation. Instead, corresponding constraints have to be lifted up and maintained in the feature model. For instance, we could create a new feature using the desired Boolean operators in its name. Then, we could control the inclusion of this new feature using propositional cross-tree constraints reflecting the desired feature combination. To implement and-expressions, we can however use nested annotations, which are represented in the program text by blending feature colors.

In Figure 2.7, we show a short CPP example, which illustrates the semantic equivalence of an annotation with a Boolean and-expression and a nested annotation. The annotated content is only included if

```

Features  Stack  Array  List  Iterable  Synchronization
1 public class Stack implements Iterable<Object> {
2   private int top = 0;
3   private Object[] dataArray;
4   private Stack.Node topNode = null;
5   private class Node { ...
6     private Node(Object element, Stack.Node next) { ... }
7   }
8   private final ReentrantLock lock = new ReentrantLock();
9   private final Condition notEmpty = lock.newCondition();
10  private final Condition notFull = lock.newCondition();
11
12  public void push(Object data) throws ArrayIndexOutOfBoundsException,
13                                     InterruptedException {
14    lock.lock();
15    try {
16      while (isFull()) notFull.await();
17      if (isFull()) throw new ArrayIndexOutOfBoundsException();
18      dataArray[top] = data;
19      Stack.Node newElement = new Node(data, topNode);
20      topNode = newElement;
21      top++;
22      notEmpty.signalAll();
23    } finally { lock.unlock(); }
24  }
25  public Object pop() ... { ... }
26  private boolean isFull() { return top >= dataArray.length; }
27  ...
28 }

```

Figure 2.6: Excerpt of a CIDE-based Stack SPL implementation



we select both features *A* and *B*. Notice that in our CIDE example in Figure 2.6 all annotations in the class `Stack` are nested, and thus blended with the color gray.

*A note on CIDE's internals:*

CIDE persists all annotations in XML files, each of which accompanies a corresponding compilation unit, such as a Java class file. This way, the

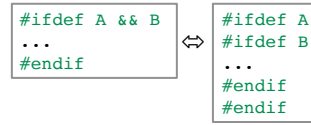


Figure 2.7: And-expression logic

program's source code remains untouched when assigning variability. In fact, all annotations are assigned to nodes of the program's AST—the underlying structural representation of the program. To ensure syntactic correct annotations, CIDE uses two simple rules [105]:

1. **OPTIONAL-ONLY RULE.** The placement of markers in the concrete syntax is limited to the program's underlying AST nodes, which are optional according to a language's syntax specification (e.g., Java [83]). These are the nodes that can be removed without invalidating the tree.
2. **SUBTREE RULE.** When an AST node is colored, all its children must be colored as well. This way, no unrelated nodes remain upon deletion.

Figure 2.8a illustrates the coloring of a `try`-statement, a node that is optional and has a subtree. If the gray feature has not been selected, all gray nodes get deleted. As a result, the method declaration which contains the subtree remains valid. Figure 2.6 shows more fine-grained examples. It is valid to remove the interface implementation (Line 1) and the throw-items (Lines 12–13), as the corresponding AST nodes are optional. In contrast, removing the class' visibility—that is, `public`—is not possible, since the respective node is non-optional.

Thus, the two rules are too strict, limiting annotations to a disciplined use—recap that around 16% of a program's annotations are undisciplined in practice [125]. To enable undisciplined annotations to align with the AST, CIDE provides *FeatureBNF* [105], a grammar specification language. Using *FeatureBNF*, language engineers can specify exceptions to the optional-only and the subtree rule in a target language's grammar, such as Java.

For non-optional language concepts, such as Java's `Type`, engineers can specify a default value in *FeatureBNF*, which enables developers to annotate corresponding non-optional nodes (i.e., the language concept instances). These nodes will not be removed during variant derivation. Instead, the default value is filled in, such as `void` for a method's non-optional return type.

To make exceptions to the subtree rule, engineers can specify wrapper language concepts—that is, language elements that wrap other concepts, such as `if`, `for`, and `try` statements. Then, developers can

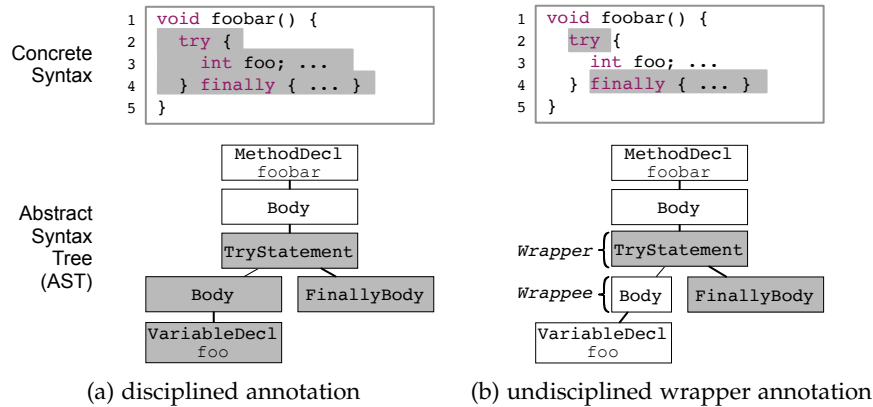


Figure 2.8: Coloring the abstract syntax tree

color the respective wrapper instances without coloring their body (the wrapped). During variant derivation, an additional transformation step is employed. This step attaches the wrapped to the wrapper’s parent (instead of removing the wrapper and its entire subtree). Figure 2.8b provides an example for such a partially annotated wrapper. In fact, the annotated try-statement and its finally-body can be safely removed without removing the try-statement’s body. It is attached to the body of the method declaration during derivation. Notice that the necessary infrastructure is generated from a FeatureBNF specification (e.g., a variability-aware parser and pretty printer).

#### 2.4.2.3 Other Textual and Visual Annotative Approaches

OTHER TEXTUAL APPROACHES. The CPP has been introduced for the C programming language. The languages *Fortran*, *Pascal* and the cross-platform toolkit *Haxe*<sup>2</sup> also come with their own preprocessors. For other languages, similar approaches have emerged. For instance, the preprocessors *Munge*<sup>3</sup> and *Antenna*<sup>4</sup>—developed with Java’s Swing library and Java ME, respectively—add conditional compilation facilities to Java. In C# and *Visual Basic* conditional compilation is even a language feature.

In the industrial sector, *Gears* [117] and *pure::variants* [35] are well-known commercial tools supporting textual annotations. In addition, several other solutions emerged supporting textual-annotation preprocessing, such as *XVCL* [94], *SPLET* [154], *General-purpose Preprocessor (GPP)*<sup>5</sup>, *M4*<sup>6</sup>, and the *choice calculus* [63].

<sup>2</sup> <https://haxe.org/>

<sup>3</sup> <https://publicobject.com/2009/02/preprocessing-java-with-munge.html>

<sup>4</sup> <http://antenna.sourceforge.net/wtkpreprocess.php>

<sup>5</sup> <https://logological.org/gpp>

<sup>6</sup> <http://www.gnu.org/software/m4/m4.html>

OTHER VISUAL APPROACHES. Similar to CIDE, the *FeatureCommander* tool uses background colors to augment CPP directives [68, 69, 71, 72]. In the same line, the *mbeddr* [197, 201] tool uses background colors to highlight feature artifacts.

Outside the SPL context, visual annotations have been used to highlight code, aggregate information and provide additional information to developers. Well-known examples are *jQuery* [92], the *Aspect Browser* [85], *concern graphs* [151], and *Spotlight* [48].

### 2.4.3 Modular Representations

Figure 2.4c illustrates that modular variability representations implement each feature in a corresponding feature module, and thus follow the principle of separation of concerns [182]. Next, we outline two important modular programming paradigms for implementing SPL variability: *feature-oriented programming (FOP)* and *delta-oriented programming (DOP)*. In fact, we discuss two concrete example approaches and conclude with an overview on related work.

#### 2.4.3.1 Feature-oriented Programming: FeatureHouse

FOP [147] approaches, such as FH [7], build upon the concepts of incrementally developing program families [142, 143]. The core idea is to start with a small base program and extend the program with functionality step by step. In fact, FOP approaches encapsulate the SPL's core functionality—the functionality that is common to each variant—in a so-called *base module* (a.k.a. *core module*). All other modules refine or extend the base module in a step-wise manner (a.k.a. *step-wise refinement* [22, 56]). In other words, the modules extend the SPL's core functionality. In FH, each module is realized as a file system directory serving as a container for a feature's artifacts. To extend or refine an existing program element (e.g., a class or a method), developers replicate the element's signature [7].

Figure 2.9 shows a FH realization of our Stack SPL. Each feature declared in the feature model (Fig. 2.3) is implemented in a respective feature module. The module *Stack* constitutes the base code of the class *Stack* (Fig. 2.9a). All other modules refine or extend the *Stack*'s core implementation (indicated by the replicated signatures). For instance, the module *Array* depicted in Figure 2.9b contributes several feature artifacts to the basic *Stack*, such as the array holding the objects (Line 2), and the method `isFull` (Line 9).

Using feature modules, FH basically supports refinement up to statement level. To refine a method declaration, developers simply replicate the target method's signature. Then, the keyword `original` allows accessing the original implementation of the method—that is, the one implemented in the feature module to be refined. Technically, the `original` keyword is simply a method call to the original decla-

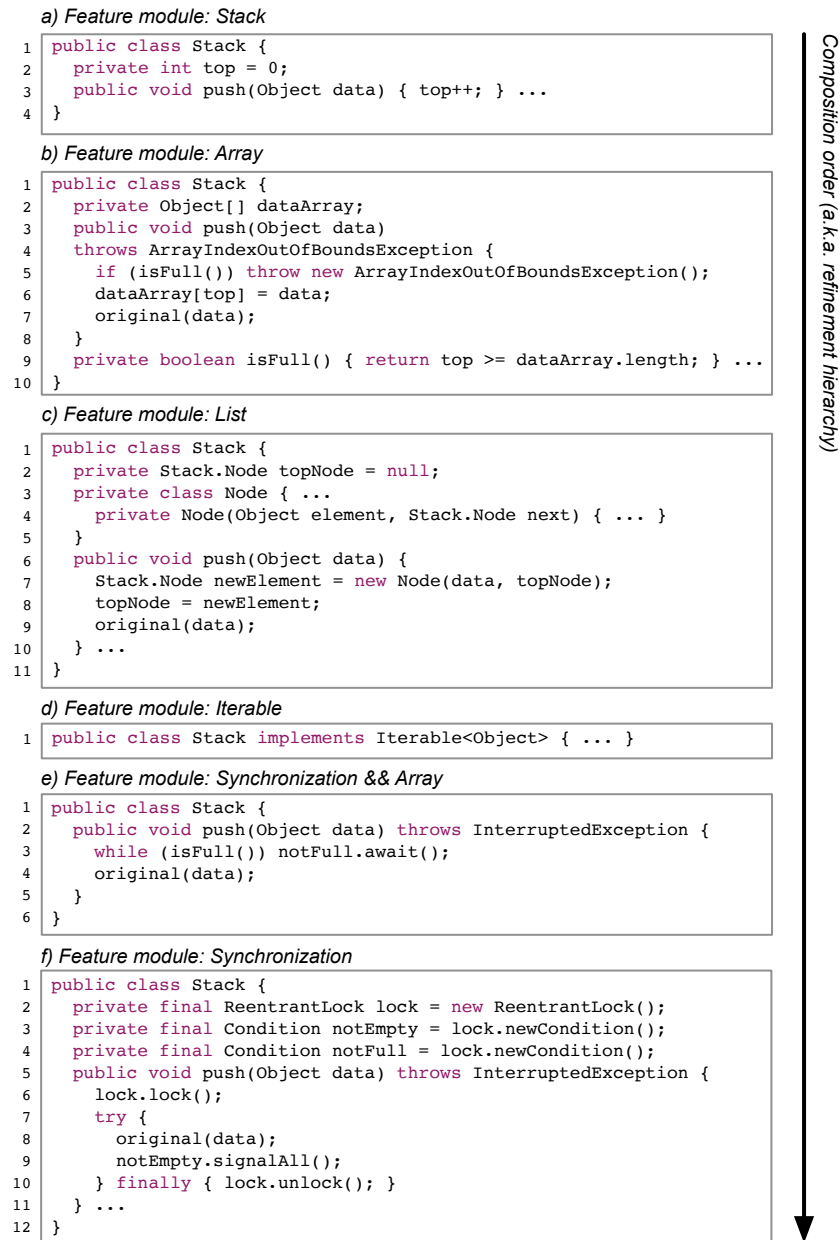


Figure 2.9: Excerpt of a FeatureHouse-based Stack SPL implementation

ration of the method. Notice that we use the `original`-keyword quite regularly in our FH implementation. For instance, the module *Array* refines the method `push`, which is indicated by the replicated signature (cf. Line 3 in Fig. 2.9a and Fig. 2.9b, respectively). In particular, the refinement introduces a `throws`-item and a sequence of statements (Fig. 2.9b, Lines 5–7). Note that FH’s keyword `original` is used to call the original implementation of `push` (Line 7), which is declared in the feature module *Stack* (Fig. 2.9a, Line 3).

Feature modules are also used to implement feature interactions. However, since there are no Boolean expressions over features in a FH implementation, we need to add a corresponding feature to the fea-

ture model as well as cross-tree constraints that select the feature automatically if all interacting features are selected. Figure 2.9e shows the feature module *Synchronization && Array*, which implements the feature interaction of the two modules *Synchronization* and *Array*. Note that the “&&”-symbols are simply tokens in the module’s name. Thus, a corresponding feature and respective cross-tree constraints must be declared in the Stack SPL’s feature model (not depicted in Fig. 2.3).

Upon variant derivation, the SPL’s feature modules serve as input. In particular, the selection of features determines the selection of modules, each of which contributes feature artifacts to the final variant. Thereby, a so-called *refinement hierarchy* denotes the ordering in which the modules are to be composed. In each composition step, the refining method’s signature remains unchanged, while there is a suffix added to the refined method’s name—that is, the name of the feature module to be refined. Then, in the refining method declaration, each occurrence of the original keyword is replaced with a method call to the refined method declaration—that is, the one with the suffix.

Figure 2.10 illustrates the example of a variant derived from selecting the feature modules *Stack*, *Array*, and *Synchronization* (Fig. 2.9a, b, f). Notice that the module *Synchronization && Array* is selected automatically to coordinate *Array* and *Synchronization* (Fig. 2.9e). According to the refinement hierarchy, the feature *Synchronization* is the last one to be composed (Fig. 2.9). Thus, the push method of *Synchronization* has not been renamed (Fig. 2.10, Line 19). To realize the refinement, the push method introduced with the module *Synchronization && Array* is called (Fig. 2.10, Line 22), which

```

1 public class Stack {
2     private int top = 0;
3     private Object[] dataArray;
4     private final ReentrantLock lock = new ReentrantLock();
5     private final Condition notEmpty = lock.newCondition();
6     private final Condition notFull = lock.newCondition();
7     private void push__wrappee__Stack (Object data) { top++; }
8     private void push__wrappee__Array (Object data)
9     throws ArrayIndexOutOfBoundsException {
10         if (isFull()) throw new ArrayIndexOutOfBoundsException();
11         dataArray[top] = data;
12         push__wrappee__Stack(data);
13     }
14     private void push__wrappee__SynchronizationAndArray (Object data)
15     throws InterruptedException {
16         while (isFull()) notFull.await();
17         push__wrappee__Array(data);
18     }
19     public void push(Object data) throws InterruptedException {
20         lock.lock();
21         try {
22             push__wrappee__SynchronizationAndArray(data);
23             notEmpty.signalAll();
24         } finally { lock.unlock(); }
25     }
26     private boolean isFull() { return top >= dataArray.length; }

```

Figure 2.10: Stack variant derived in FH by composing the feature modules *Stack*, *Array*, *Synchronization*, and *Array && Synchronization* (cf. Fig. 2.9)

adds the necessary feature details and then calls the push method of the module *Array* (Line 17), which in turn calls the push method's base code (Line 12).

Thus, the order in which modules are composed may matter. In fact, notice that the condition `notFull`—which is only required for array-based, synchronized stacks—is introduced by the module *Synchronization* (Fig. 2.9f, Line 4), although the logically correct module would be *Synchronization && Array*. However, we cannot move the condition `notFull` to this module, since an error would arise. The problem is that the field `lock`, which is introduced with the module *Synchronization*, cannot be referenced before it is defined (Fig. 2.10, Lines 4 and 6). Moreover, we cannot switch the two modules in the refinement hierarchy as we need to synchronize the call to the method `isFull` (Fig. 2.10, Line 16). As a result, the class *Stack* cannot be modularized properly using only the two modules *Synchronization* and *Synchronization && Array*.

#### 2.4.3.2 Delta-oriented programming: *DeltaJ*

DOP is a popular modular paradigm, closely related to FOP though more powerful. Features are separated into (multiple) *delta modules* (a.k.a. *deltas*) [33, 115, 157, 158]. Each delta not only supports introducing and refining feature artifacts (as known from FOP), but also removing them during variant transformation [157, 158].

*DeltaJ* [115] puts DOP into practice for Java by introducing inter alia the keywords `delta`, `adds`, `modifies`, `original`, and `removes`. Note that the latter four keywords enable manipulating a variant. In fact, similar to FH, *DeltaJ* uses a core delta—an initial variant to which all further deltas apply their transformations in a given order.

Figure 2.11 illustrates an implementation excerpt of our Stack SPL using *DeltaJ*. The delta *dStack* is the core delta. The delta *dArray* depicted in Figure 2.11b adds the field `dataArray` (Line 2) and the method `isFull` (Line 8) to the core delta. Moreover, *dArray* implements a modification of the method `push` (Lines 3–7). Notice that similar to FH access to the original code is given by the `original` keyword (Line 6). *DeltaJ* transforms each appearance of the `original` keyword into a method call to the modified original method (cf. Fig. 2.10). As a result, the same issues as in FH emerge. For instance, the order in which transformations are to be applied matters. Moreover, a technical limitation of *DeltaJ*'s current realization is that `throws-items` cannot be properly modularized. Instead, `throws-items` must be added to the core delta (cf. Fig. 2.11a, Line 6).

Notice that instead of a feature model, *DeltaJ* uses a so-called *SPL declaration* for modeling and configuring the SPL's variability. In particular, the SPL declaration allows developers to declare features and feature constraints. Moreover, deltas and the mapping of features to deltas can be declared using propositional formulas over features.

a) Delta module: *dStack*

```

1 delta dStack { adds{
2   package my;
3   public class Stack {
4     private int top = 0;
5     public void push(Object data)
6       throws ArrayIndexOutOfBoundsException, InterruptedException {
7       top++;
8     } ...
9   } }

```

b) Delta module: *dArray*

```

1 delta dArray { modifies my.Stack {
2   adds private Object[] dataArray;
3   modifies push(Object data) {
4     if (isFull()) throw new ArrayIndexOutOfBoundsException();
5     dataArray[top] = data;
6     original(data);
7   } ...
8   adds boolean isFull() { return top >= dataArray.length; }
9 } }

```

c) Delta module: *dIterable*

```

1 delta dIterable { modifies my.Stack{
2   adds interfaces Iterable<Object>; ...
3 } }

```

d) Delta module: *dSyncAndArray*

```

1 delta dSyncAndArray { modifies my.Stack {
2   modifies push(Object data) {
3     while (isFull()) notFull.await();
4     original(data);
5   }
6 } }

```

e) Delta module: *dSynchronization*

```

1 delta dSynchronization { modifies my.Stack {
2   adds private final ReentrantLock lock = new ReentrantLock();
3   adds private final Condition notEmpty = lock.newCondition();
4   adds private final Condition notFull = lock.newCondition();
5   modifies push(Object data) throws InterruptedException {
6     lock.lock();
7     try {
8       original(data);
9       notEmpty.signalAll();
10    } finally { lock.unlock(); }
11   } ...
12 } }

```

Transformation order  
↓

Figure 2.11: Excerpt of a delta-oriented Stack SPL implementation

This way, developers can specify when and in which order deltas are to be applied. Finally, developers can specify feature selections in variant declarations.

Figure 2.12 illustrates the Stack’s SPL declaration, which conforms to our feature model (Fig. 2.3). In addition to declaring features (Line 2) and deltas (Line 3), we use two constraints to specify that *Stack* and either *Array* or *List* must be selected in each product variant (Line 4). The *Partitions* section allows introducing a mapping of features to deltas (Lines 5-10). For instance, the delta *dSyncAndArray* must be applied if the features *Array* and *Synchronization* are selected (Line 9), which is the case in the product configuration called *SyncedArrayStack* (Line 13). Notice that the transformations are applied during variant derivation in the order of the partitions section from top to bottom (i.e., from *dStack* to *dSynchronization*).

```

1 SPL AnotherStackDeltaJ {
2   Features = {Stack, Array, List, Synchronization, Iterable}
3   Deltas = {dStack, dArray, dList, dSynchronization, dSyncAndArray, dIterable}
4   Constraints { Stack; Array ^ List; ... }
5   Partitions { {dStack} when (Stack);
6               {dArray} when (Array);
7               {dList} when (List);
8               {dIterable} when (Iterable);
9               {dSyncAndArray} when (Array & Synchronization);
10              {dSynchronization} when (Synchronization);
11 }
12   Products { ListStack = {Stack, List};
13              SyncedArrayStack = {Stack, Array, Synchronization};
14 }
15 }

```

Figure 2.12: Delta-oriented Stack SPL declaration

### 2.4.3.3 Other Modular Approaches

**OTHER FOP APPROACHES.** Aside from FH, other approaches, such as AHEAD and *FeatureC++* [11], also realize FOP. All approaches realize the paradigm in different, yet very similar ways, typically sharing two key commonalities. First, approaches are usually language-based, which results in the concrete syntax of a target language, such as Java or C, being enhanced with variability-related keywords (instead of managing variability on the tool level). Second, they decompose a system into cohesive feature modules, each of which is realized as a file system directory serving as the container for feature artifacts. However, FOP approaches also differ. First, the concrete syntax employed to implement variability is typically different among approaches. For instance, while FH indicates the refinement of program elements by means of replicated signatures [7], AHEAD uses the keyword *refines* to indicate refinement explicitly [22]. Second, approaches differ in the way variants are derived—that is, module composition differs. For instance, while FH language-independently merges the substructure of modules using superimposition [7, 10], AHEAD synthesizes modules using Java-specific transformations [22].

**OTHER DOP APPROACHES.** Notice that the program transformations proposed with delta modeling are independent of programming languages and artifact-types [168]. In fact, delta modeling has been manually tailored to Java in DeltaJ. Delta modeling has also been used for various other target environments and languages, such as *Matlab/Simulink* [88] and *architecture description languages (ADL)* [87].

**OTHER MODULAR APPROACHES.** Aside from FOP and DOP, there are several implementation approaches that support modularity at their core, such as *aspects* [2, 111], *collaboration-based design* [190], *mix-ins* [40, 176], and *traits* [32, 34, 58]. They all share the idea of encapsulating variability in modules.

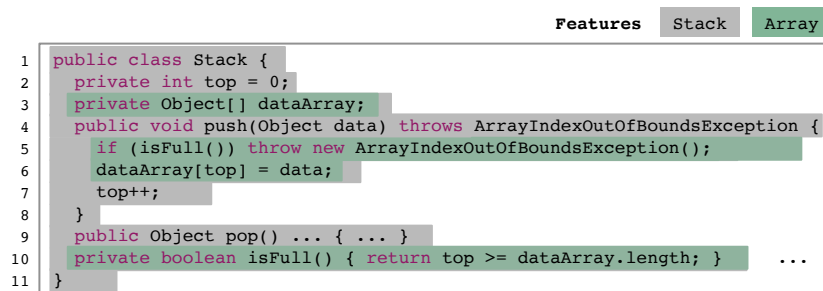


## 2.4.4 Variant Representations

Typically realized on top of an annotative representation, variant representations allow editing SPL variants in isolation. Several approaches for editing SPL variants have been proposed, such as the *version editor* [16], the *variation-control system (VCS)* [178], CIDE’s variant editor [104], and others [119, 166, 206]. They are influenced by the idea of bidirectional transformations [19, 52, 54, 76, 196]. In fact, some approaches fold [120] or hide [102, 175] feature code. Others, such as VCS, mimic the workflow of version-control systems by allowing to checkout variants, edit them, and commit the edited variants [178, 205, 206].

In this dissertation, we distinguish two different types of variant editors: proactive and reactive ones. Proactive variant editors show all feature information while editing the variant, for instance as annotations [104]. This way, feature-related decisions can be made before editing and code can be unambiguously assigned to features. Reactive approaches typically do not show feature information. Any feature-related decisions are made (semiautomatically) after editing. For instance, VCS requires to assign the implemented artifacts to features upon pushing the changes back into the SPL [178].

Notice that CIDE provides facilities to hide feature artifacts based on a feature selection. Figure 2.13a shows the proactive, colored variant editor provided by CIDE. It shows the Stack SPL with the features

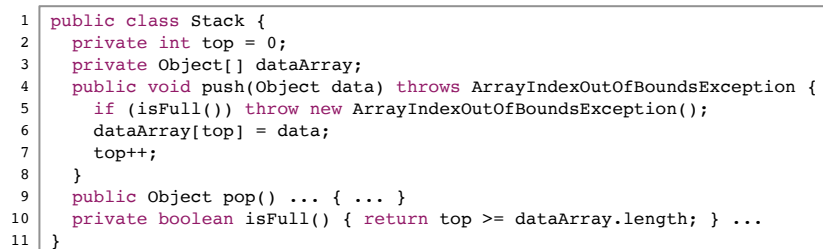


```

1 public class Stack {
2     private int top = 0;
3     private Object[] dataArray;
4     public void push(Object data) throws ArrayIndexOutOfBoundsException {
5         if (isFull()) throw new ArrayIndexOutOfBoundsException();
6         dataArray[top] = data;
7         top++;
8     }
9     public Object pop() ... { ... }
10    private boolean isFull() { return top >= dataArray.length; } ...
11 }

```

(a) CIDE’s proactive, colored variant editor for the selection of the features *Stack* and *Array* in our Stack SPL



```

1 public class Stack {
2     private int top = 0;
3     private Object[] dataArray;
4     public void push(Object data) throws ArrayIndexOutOfBoundsException {
5         if (isFull()) throw new ArrayIndexOutOfBoundsException();
6         dataArray[top] = data;
7         top++;
8     }
9     public Object pop() ... { ... }
10    private boolean isFull() { return top >= dataArray.length; } ...
11 }

```

(b) A reactive variant editor for the selection of the features *Stack* and *Array* in our Stack SPL

Figure 2.13: Editable variant representations

*Stack* and *Array* being selected. Figure 2.13b in turn shows a reactive variant editor, such as VCS, for the same features being selected.

## 2.5 CHALLENGES CAUSED BY VARIABILITY

We now discuss three challenges caused by SPL variability. We start with problems that emerge while implementing variability. Then, we review the different variability representations with regard to potential comprehension, maintenance and evolution issues. Notice that in Section 5.1, we use—complementary to this section—different quality criteria to assess and categorize different concrete SPL implementation approaches and our approach.

### 2.5.1 Challenge I: Implementation

Implementing SPL variability is the process of creating feature artifacts. The efficiency to create these artifacts differs among the implementation approaches.

**TEXTUAL AND VISUAL ANNOTATIONS.** In annotative approaches, developers simply mark the beginning and the end of feature artifacts in the codebase. To create annotations (i) using the CPP, we wrap the code to be annotated with `#ifdef`-directives; (ii) using CIDE, we select the code to be annotated and choose the desired feature(s) via a popup menu. Thus, implementing new, even fine-grained feature artifacts from scratch, and transitioning an existing non-variational codebase into an SPL is easy.

Figures 2.14a and 2.14b underline that it is simple to implement fine-grained feature artifacts using the CPP and CIDE, respectively. In fact, the two semantically equivalent implementations of the Stack SPL's push-method are more fine-grained than the one depicted in Figure 2.5. Next, we briefly discuss the CIDE version—whose annotations are added to the underlying AST and visualized in the implementation (Fig. 2.14b, lower and upper halves). To synchronize the access to the Stack's resources in the push-method, we pass a `ReentrantLock` as a method parameter and assign it to the feature *Synchronization* by coloring it blue (Line 3). Moreover, we verify validity of both method parameters in an `if`-statement (Line 7), and also color the reference to the parameter `ReentrantLock` blue. Otherwise, variants without *Synchronization* selected are type-incorrect—that is, a dangling reference would appear, since the parameter `ReentrantLock` would be removed (Line 3), but not the reference to it (Line 7).

**VARIANTS.** Variant editors, which typically use an annotative representation internally, also enable developers to efficiently implement feature artifacts. In fact, proactive variant editors show annotations, and thus provide the same straightforward workflow as annotative

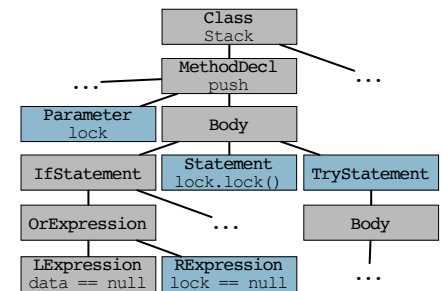
```

1 #ifndef Stack
2 public class Stack { //(1) more code
3     public void push(Object data
4     #ifndef Synchronization
5     , ReentrantLock lock
6     #endif
7     )
8     throws NullPointerException
9     #ifndef Array
10    , ArrayIndexOutOfBoundsException
11    #endif
12    #ifndef Synchronization
13    , InterruptedException
14    #endif
15    {
16        if(data == null
17        #ifndef Synchronization
18        || lock == null
19        #endif
20        )throw new NullPointerException();
21        #ifndef Synchronization
22        lock.lock();
23        try {
24        #endif
25        //(2) more code
26        top++;
27        //(3) more code
28        #ifndef Synchronization
29        notEmpty.signalAll();
30        } finally {
31        lock.unlock();
32        }
33        #endif
34    }
35 }
36 #endif
    
```

(a) CPP implementation

```

Features Stack Array Synchronization
1 public class Stack { //(1) more code
2     public void push(Object data,
3     ReentrantLock lock)
4     throws NullPointerException,
5     ArrayIndexOutOfBoundsException,
6     InterruptedException {
7     if(data == null || lock == null)
8     throw new NullPointerException();
9     lock.lock();
10    try {
11    //(2) more code
12    top++;
13    //(3) more code
14    notEmpty.signalAll();
15    } finally {
16    lock.unlock();
17    }
18 }
19 }
    
```



(b) CIDE implementation (upper half) and underlying AST (lower half)

```

Feature module: Stack
1 public class Stack {
2     public void push(Object data) {
3         if(data == null || push_sync_hook1())
4             throw new NullPointerException();
5         push_sync_hook2(data)
6     }
7
8     public void push_sync_hook1() { return false; }
9     public void push_sync_hook2(Object data) { top++; }
10 }

Feature module: Synchronization
1 public class Stack {
2     ThreadLocal<ReentrantLock> currentLock = new ThreadLocal<ReentrantLock>();
3     public void initLock(ReentrantLock lock) { currentLock.set(lock); }
4
5     public void push_sync_hook1() { return currentLock.get() == null; }
6     public void push_sync_hook2(Object data) {
7         currentLock.get().lock();
8         try {
9             original(data);
10            notEmpty.signalAll();
11        } finally {
12            currentLock.get().unlock();
13        }
14    }
15 }
    
```

(c) FH implementation

Figure 2.14: Implementation of fine-grained feature artifacts using the CPP, CIDE and FH (inspired by [102, 162])

approaches. In contrast, reactive variant editors, such as VCS, impose an editing overhead—that is, the checkout-edit-commit workflow to implement feature artifacts.

**MODULES.** Using a modular approach, it is also efficient to implement feature artifacts from scratch. Developers simply introduce new artifacts, refine, override, or even remove existing ones in the respective feature module. Yet, decomposing an existing non-variational program into modules is challenging since developers need to find the right decomposition strategy and restructure the codebase manually. Moreover, modules impose an overhead for fine-grained feature artifacts requiring workarounds, such as hook methods [102] and code clones [162]—that is, boilerplate code. In fact, we found that a large number of boilerplate code is necessary in real world SPLs. For instance, in our case study subject Berkeley DB with 70kLOC, 78% off all methods refined only once require boilerplates in pure modular approaches (cf. Sec. 5.2.3.3).

Figure 2.14c shows the FH-version of the fine-grained Stack SPL example, we introduced above. The main problem is that it is not possible to properly modularize the feature *Synchronization* in FH. First, the method parameter variability cannot be realized (as in Fig. 2.14b, Line 3). Instead, each thread has its own thread-safe ReentrantLock (Fig. 2.14c, *Synchronization*, Line 2), which must be initialized before calling the push method (Line 3). Second, to assign the variable reference `lock` to the feature *Synchronization* (as realized in Fig. 2.14b, Line 7), we need to prepare the base module *Stack* by adding a hook method (Fig. 2.14c, Lines 3 and 8). The hook's declaration is then overridden in the feature module *Synchronization* by a check whether the lock is valid (Fig. 2.14c, Line 5). Third, locking the codebase just partially—where not all statements of the codebase are synchronized—is not possible without adding yet another hook method (Line 6).

We conclude that assigning annotations is straightforward, while implementing modules can be more difficult. With regard to variant representations, assigning features during development (proactive variant editor) might be more efficient than making all feature decisions after a commit (reactive variant editor).

### 2.5.2 Challenge II: Comprehension

Program comprehension is *'the process of understanding program code unfamiliar to the programmer'* [114]. As developers spent a majority of their time with comprehending programs [131, 179, 188], it is one of the major cost drivers in software development [38]. The different variability representations influence program comprehension very differently. Although general measures exist [59, 169] and (pilot) stud-

ies have been conducted [8, 171, 172], providing a concise ranking for variability representations is difficult [67, 170].

Based on the literature, our experience from implementing SPLs (Sec. 5.2.2), and our observations made in two pilot user studies (Sec. 5.3), we investigate program comprehension from two different perspectives. First, we discuss rather general comprehension issues induced by different variability representations. Second, we refine these issues according to their cause. That is, we discuss *variability-aware code smells* [73], and how they lead to incomprehension across different variability representations.

### 2.5.2.1 Representation-related Comprehension Issues

When developers try to comprehend programs, each variability representation has its own individual strengths and weaknesses.

**TEXTUAL ANNOTATIONS.** Textual annotations are well known and well adopted in practice. Yet, several studies have shown that textual annotations have a negative impact on program comprehension [62, 65, 102, 116, 126, 177]. Textual annotations are embedded into the program text, and thus intermingle with the program’s functionality, making a program’s control flow difficult to follow—especially if fine-grained annotations and complex presence conditions appear. Understanding whether the variational code is reachable [183] and if so under which circumstances is tedious and error-prone [80, 89, 135]. Our CPP-based Stack SPL implementation depicted in Figure 2.14a illustrates the source code obfuscation. In fact, reasoning about the Stack’s functionality is difficult. Unfortunately, there are even more extreme examples in every day practice [90, 125].

**VISUAL ANNOTATIONS.** To improve the comprehensibility of textual annotations, previous work proposed to augment CPP directives with background colors [69, 72]. Tools, such as mbeddr [201] and CIDE [102], also use background colors to visualize annotations aiming at an improved program readability. In fact, our Stack SPL excerpts illustrate that using background colors instead of `#ifdef` directives substantially improves the program’s readability (cf. Fig. 2.14a and 2.14b).

However, relying solely on background colors can be problematic. First, for nested feature artifacts, background colors are blended. Thus, distinguishing colors and relating them to features can be challenging [138], or even impossible (e.g., color-blind developers). Second, recap that colored non-optional AST nodes, such as expressions, cannot be removed from the program’s AST during variant derivation since we would invalidate the AST (cf. Sec. 2.4.2.2). For instance, a right-expression of an or-expression cannot be removed (cf. Fig. 2.14b lower half, and upper half, Line 7). For such non-optional AST nodes, CIDE allows language engineers—the developer who

tailors CIDE to a programming language to declare a default value in the target-programming-language’s grammar (e.g., “0” or “1”), which is then used during derivation. Yet, which value is used by default is opaque to the SPL developer—a subtle cause of errors.

All in all, background colors are discussed controversially. For example, in an early study of the *mylyn* tool, Kersten and Murphy received negative feedback for *mylyn*’s fixed background colors [109]. In particular, two developers indicated that they “*prefer their IDE to be less colorful*” [109]. In another study conducted by Melo et al., a participant who tried to understand a program colored with CIDE explained [135]: “*I tried to keep all different paths in mind, but it was especially difficult with multiple colors.*” In the same study, another participant claimed “*with more variability you need to build up exponentially more traces in your head*” [135].

**VARIANTS.** Variant editors—which are typically realized upon an annotative representation—reduce the variability-related complexity, since they show a smaller set of feature artifacts. Optimally, depending on the current comprehension task, they either show or hide the underlying annotations in the variant (cf. Sec. 2.4.4). If annotations are shown, developers can comprehend which features are involved. If annotations are hidden, the code is less cluttered—looking like the generated variant—which makes control-flows easier to understand. Thus, both variant representations are valuable, but often only either way is supported in respective tooling [102, 175].

Notice that there are also comprehension tasks where a variant representation is not the perfect choice. On the one hand, code that is relevant for understanding the SPL may be hidden. On the other hand, the code of a variant might still be too cluttered when containing irrelevant feature artifacts, which challenges comprehension. In fact, a variant representation does not provide true modularity, and thus a feature cannot be explored in isolation.

**MODULES.** Modularity enables developers to “*identify, encapsulate, and manipulate only those parts of a software that are relevant to a particular concept, goal, or purpose*” [141]. In fact, already in the early days of software engineering, modularity has been associated with an improved program comprehension [56, 142, 144]. In modular SPL representations, each feature is clearly separated into a respective feature module, and thus the information overhead is smaller. As a result, the developer’s working memory might be less stressed [17, 171].

Yet, sometimes there is too little context information to understand a feature module’s implementation in isolation. In the worst case, developers must search manually for related, external feature artifacts, which are declared in other modules. Our fine-grained FH Stack implementation illustrates the issue (Fig. 2.14c): the code of the feature module *Synchronization* cannot be understood without reasoning

about the feature module *Stack*. That is, we must look up the positions of the hook method calls to understand the decomposition (Fig. 2.14c, *Stack*, Lines 3 and 5).

To mitigate such issues, feature-context interfaces emerged for FH recently [160]. Using static analysis, an algorithm collects context information about external feature artifacts automatically. The gathered information is then presented to the developer in an additional IDE window and the code completion menu. Yet, optimally, the information could be directly integrated into the source code. Unfortunately, several other modular approaches, such as AHEAD and DeltaJ, do not support such context interfaces at all. This lack poses an issue in implementations using DeltaJ (and other delta modeling solutions), since it is challenging to understand without guidance whether a feature artifact of a delta will be deleted by another delta. Type-checking the SPL mitigates this issue to a certain degree [33, 53, 156], but a mechanism for showing context information and giving hints—optimally, directly in the codebase—is still desirable.

### 2.5.2.2 Representation-unrelated Comprehension Issues

Comprehension issues not only appear because of variability representation weaknesses. In some cases, the functionally same implementation can be difficult to comprehend across different representations. This issue is typically caused by so-called *code smells*—flaws in a software’s design and implementation [77]. Based on an existing code smell catalog [77], Fenske and Schulze derive four code smells that appear in the context of (textual) annotations and feature modules [73]. Most important, they identify that methods containing many feature artifacts of different features are difficult to understand irrespective of the representation. Then, for a more thorough investigation, they distinguish the smell into `LONG REFINEMENT CHAIN` (feature modules) and `ANNOTATION BUNDLE` (annotations) [73].

`ANNOTATION BUNDLE` describes a method that contains many, potentially nested annotations [73]. For instance, the `Stack SPL`’s `push`-method is rather short since it comprises only twelve statements. Yet, these statements are obscured by an `ANNOTATION BUNDLE` of five annotations making the control flow difficult to follow (Fig. 2.5). `LONG REFINEMENT CHAIN` is the modular counterpart, describing a method that is often refined [73]. For example, the `Stack SPL`’s FH implementation includes four separate modules, each refining the `push`-method which already makes it difficult to understand how a possible variant looks like and how the features interact (Fig. 2.9).

Altogether, annotations and modules are both not perfectly suited for understanding methods that contain a lot of variability. Thus, Fenske and Schulze propose to correct these variability-aware smells by reorganizing the implementation in the course of future work [73],

but ideally developers could cope with `ANNOTATION BUNDLE` and `LONG REFINEMENT CHAIN` more flexibly.

### 2.5.3 Challenge III: Maintenance and Evolution

Maintenance tasks, such as implementing minor changes, fixing bugs, and improving the overall software quality are common engineering activities which consume on average half of a developer's programming time [82]. Evolving software is another common activity, which comprises major modifications to the implementation, such as adding entirely new functionality. Both activities involve comprehension tasks, since developers must understand the relevant program parts before they can maintain or evolve them. In addition to the previously discussed comprehension issues, there are several maintenance and evolution issues that arise when using a particular variability representation in isolation. Next, we discuss these issues.

**TEXTUAL AND VISUAL ANNOTATIONS.** Several studies explored the impact of annotation-based variability on maintenance and evolution tasks [1, 73, 133, 135, 150, 164]. In fact, many developers report that they regularly deal with bugs related to annotations [133]. All in all, maintaining and evolving code that is enriched with annotations can be cumbersome and error-prone [133, 164]. For instance, changing code in an `ANNOTATION BUNDLE` requires great caution as inadvertencies may lead to bugs, such as broken presence conditions or dangling references [73]. Thus, while trying to fix a variability-related bug one may introduce another bug, which leads to a vicious circle.

However, there is also good news. In an annotative representation, the *'bug-finding time appears to increase only linearly with the degree of variability'* [135]. Thus, although time-consuming, finding bugs in annotated code seems to be practical.

**VARIANTS.** Yet, if the exact variant configuration in which the bug appears is known, it is easier to explore the respective variant than the full variational codebase (e.g., a customer, whose feature selection is available, reported an error). This way, irrelevant code must not be explored. In contrast, if the variant subset is unknown, finding a bug is difficult. In fact, if developers observe all variants manually (in the hope to detect the bug at some point), there would be an exponential explosion in the bug finding time [135].

All in all, as obscuring elements are hidden, making minor or major changes is typically more efficient in a variant than in the full SPL. In fact, a study of Atkins et al. has shown that the productivity of developers can be increased by 40% when editing a variant instead of the entire SPL [16]. However, developers must take special care while changing methods which contain a lot of variability. Code alignment issues (with respect to the hidden code) and unexpected feature inter-



actions could appear. Such unexpected feature interactions are hard to detect without contextual information. Unfortunately, there seems to be no variant editor available that informs the developer about an `ANNOTATION BUNDLE` while editing.

**MODULES.** Modularity promises an improved customizability [86] and maintenance [140, 189], as well as a better traceability [4] and reusability of feature artifacts [6, 36]. Feature modules give a clear structure of the SPL and offer the benefit of editing individual features without being distracted by irrelevant ones. Thus, maintaining and evolving a single feature in a dedicated module is typically straightforward. A `LONG REFINEMENT CHAIN`, however, may cause unexpected problems. A developer may accidentally neglect the refinement hierarchy, which results in a so-called *ordering issue* [7]. For instance, moving the feature *Synchronization* up in the refinement hierarchy invalidates thread-safety (Fig. 2.9). In fact, the Stack's resources would be unsynchronized, which is a cause of subtle errors. Such issues are difficult to detect in isolation. Yet, a modular approach that informs the developer about a `LONG REFINEMENT CHAIN` seems to be missing. Moreover, similar to variant editing, it is time-consuming and difficult to find a bug across a set of modules if the exact feature in which it appears is unknown. Then, developers must explore all feature modules in the hope of finding the bug at some point.

## 2.6 CONCLUDING REMARKS

Notice that this dissertation does not intend to give a recommendation, which variability representation is the best for engineering an SPL. Instead, the discussion of SPL implementation challenges has shown that all common variability representations share distinct advantages, and thus are complementary [100, 103, 171]. Unfortunately, existing SPL implementation approaches typically focus on one representation. Most importantly, these approaches force developers to choose one representation for developing a feature artifact and to adhere to it for evolving and maintaining this artifact. While refactorings were proposed for switching between annotative and modular representations [103], such refactorings are heavyweight and do not allow to quickly switch the representation for a feature artifact. Ideally, developers could exploit the benefits of different representations on-demand and always choose the one that suits the current engineering activity.

Part II

ENGINEERING AND MANAGING  
VARIABILITY IN PEOPLE

## IMPLEMENTING VARIABILITY: PEOPL'S FLEXIBILITY

---

We present how developers implement, comprehend, maintain, and evolve SPLs in PEOPL, a novel approach and IDE. PEOPL allows developers for a given feature artifact and task to flexibly and fluently choose the best-suited one among very different variability representations. We start with an overview (Sec. 3.1) and PEOPL's key benefits, which help to address the variability engineering challenges (Sec. 3.2). Using the example of variational Java code, we illustrate how developers use PEOPL's diverse variability representations, even for the very same feature artifact (Sec. 3.3). Then, we underline PEOPL's language and artifact-type independency by showing how developers use it for variational mathematics and fault trees (Sec. 3.4). Finally, we discuss how developers browse the SPL using variability-aware file explorers (Sec. 3.5). In summary, we focus in this chapter on the SPL developers' perspective and how developers use PEOPL. We discuss conceptual details and how the approach is realized in Part [iii](#).

### 3.1 PEOPL'S CORE IDEA: AN OVERVIEW

PEOPL is a general approach combining very different variability representations in a flexible environment—conceived as an IDE. The core idea of PEOPL is to establish a single, common internal variability representation of the SPL and separate it from the external variability representations that developers use. Figure 3.1 illustrates these ideas using a real-life scenario taken from our Berkeley DB case study. The lower half of the figure shows the internal representation, where feature artifacts are uniformly persisted in a variational AST. The upper part illustrates PEOPL's basic external representations, realized as so-called *projections* (cf. [203]). In fact, we conceive, realize, and evaluate projections—many of which are closely related to the variability representations we have seen thus far—showing feature artifacts

- as textual annotations (`#ifdef` directives),
- as visual annotations (colored bars),
- in feature modules (similar to AHEAD and FH modules),
- as annotations blended into feature modules,
- in proactive and reactive variant representations, which hide artifacts related to non-selected features,
- in fade-in feature modules (adopting some DeltaJ keywords),
- as reused code elements (instead of cloning an artifact).

Developers use the projections—as we will explain in this chapter—to engineer feature artifacts. Any of their editing activities directly change the underlying variational AST, which immediately updates all projections. Thus, developers see the concrete syntax of the variational program (upper half of Fig. 3.1), but directly interact with the underlying AST (lower half of Fig. 3.1). Notice that this way of editing is fundamentally different to parser-based text editing, where developers see and edit the program's concrete syntax. Such a projectional editing has long been seen as problematic, hindering efficient code-editing. Yet, improvements [195]—realized in modern language workbenches [60, 61], such as MPS [61, 145, 201]—and recent studies [31, 203] have shown that

- editing efficiency is quickly achievable by user training,
- projectional editing even leads to fewer typing mistakes, and
- recent, novel editing support facilities—especially for editing expressions—substantially improve projectional editing.

Our own experience and our pilot user studies confirm these findings. In fact, we found that projectional editing—as employed for implementing SPL variability in PEOPL—is practical.

### 3.2 PEOPPL'S BENEFITS: ADDRESSING THE CHALLENGES

The key benefits of our approach can be summarized from the SPL developer's perspective as follows.

- B1: PEOPL has a *uniform internal representation*, designed to support diverse external variability representations. Uniformity allows persisting variability in a consistent manner. Using different (internal) representations would break uniformity—for instance, when adding `#ifdef` directives into FH modules. As a result, using PEOPL, developers need not to combine and coordinate different tools to implement feature artifacts in diverse ways. Instead, they simply choose the best technique for a given task.
- B2: Developers can *switch the external representation* of a feature artifact (e.g., class) on demand. In fact, PEOPL allows a fluent movement between the external representations of a feature artifact—for instance of `DatabaseImpl` in the upper half of Figure 3.1—to enable developers to exploit the distinct advantages of different techniques for a given feature artifact and task.
- B3: Developers can observe and edit the same feature artifact *using different external representations in parallel* (by showing them side-by-side), which enables an even faster movement between different representations. For instance, the artifact `DatabaseImpl` in

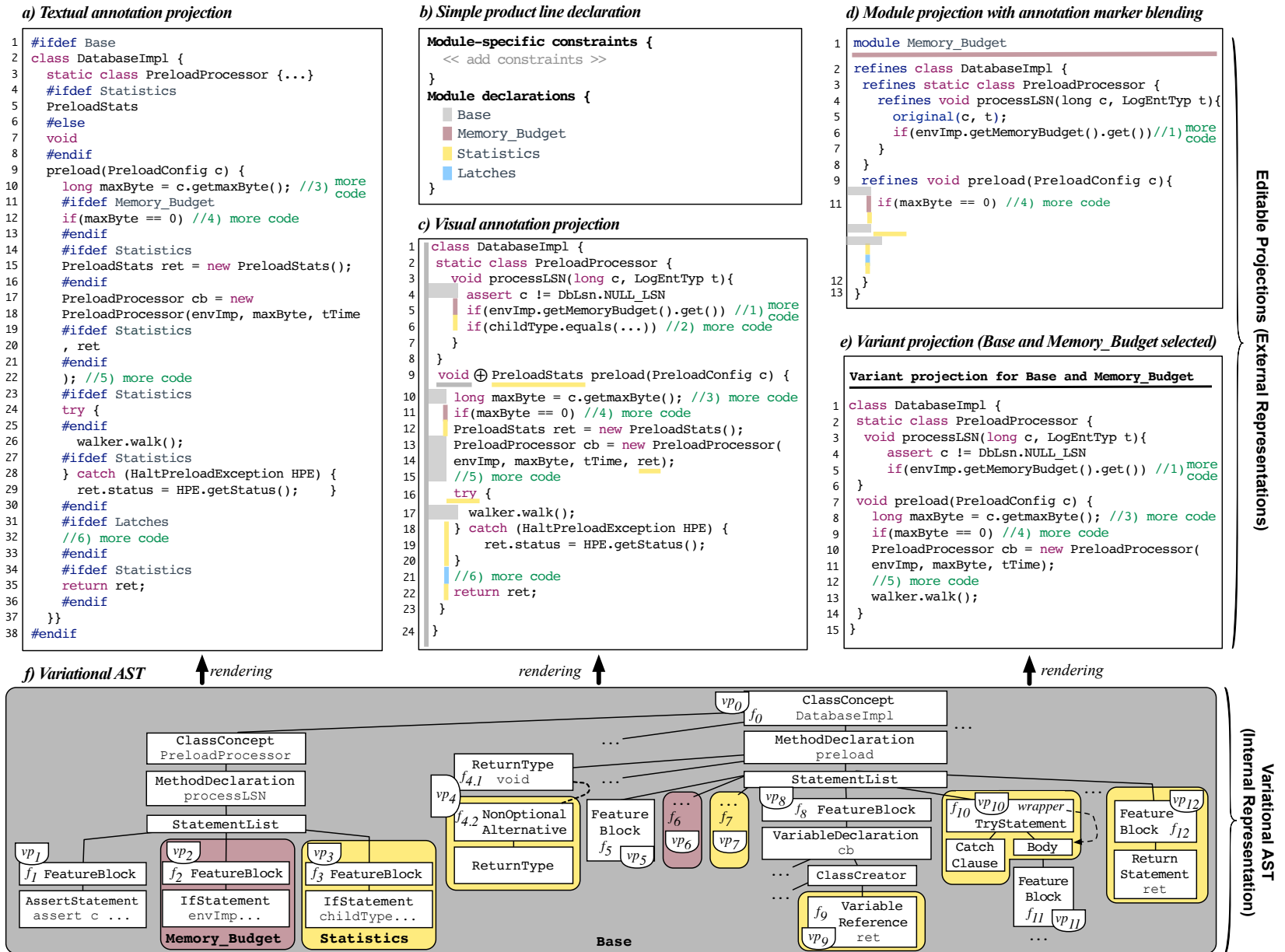


Figure 3.1: An excerpt of Berkeley DB implemented in PEOPL. The upper half shows the projections used by developers (external representation). The lower half shows the variational AST (internal representation).

the upper half of Figure 3.1 can be indeed edited in its diverse representations in parallel. Any edit in one representation immediately updates all other representations. Thus, PEOPL helps observing the impact of changes made from one representation to another in real-time (e.g., editing a feature module and a variant in parallel).

- B4: PEOPL *mitigates typical shortcomings of modular representations*, imposed by granularity problems, and the lack of context information (as they are contained in external modules). In fact, PEOPL allows developers to blend annotations into a module’s implementation on demand, and fade-in external module code. This way, developers can implement fine-grained feature artifacts in modules (without breaking modularity and uniformity) and integrate context information from other modules directly into the implementation—that is, it is possible to show all accessible field and method declarations in a modular implementation.
- B5: Our uniform internal representation enables *plugging new external representations into PEOPL* on demand. In fact, developers can draw on an extensible set of variability representations, which enables them to change the appearance of their SPL implementation on demand. This way, PEOPL can serve as a framework for the evaluation of diverse variability representations from the developer’s perspective.

These benefits enable a flexible and novel implementation of SPL variability. In fact, each variability representation has its own advantages and disadvantages with regard to different implementation, comprehension, maintenance and evolution tasks (cf. Challenges I–III in Sections 2.5.1, 2.5.2, and 2.5.3), and thus the ability to switch between PEOPL’s projections and to leverage them is indeed valuable. For instance, assume that a developer faces an `ANNOTATION BUNDLE`—a method that contains a lot of annotations in relation to its statements (cf. Sec. 2.5.2.2). If the developer just wants to reason about a single feature’s control flow or the control flow of a variant, switching to the respective projection reduces the complexity, and thus is a valuable option. Likewise, if a developer faces a `LONG REFINEMENT CHAIN`—a method that is refined multiple times (cf. Sec. 2.5.2.2)—integrating the statements of external feature modules facilitates understanding the refinement hierarchy.

In the remainder of this chapter, we discuss more concrete examples and novelties underlining PEOPL’s flexibility (B1–B5).

### 3.3 PROJECTIONS I: VARIABILITY REPRESENTATIONS

We conceive, realize, and evaluate seven external representations (projections), each of which represents feature artifacts differently. In the

following, we discuss the different projections, their conceptual details, the concrete syntax, and the variability-related editing operations available in the respective projection. For explaining syntactical details, we use our Berkeley DB example depicted in Figure 3.1, and the fine-grained Stack SPL shown in Figure 2.14. The examples are written in Java, but the different projections are independent of a specific programming language (e.g., C is supported as well [74]), and even the artifact-type of structural elements—that is, graphical representations are also supported (cf. Sec. 3.4).

### 3.3.1 Textual Annotation Projection

Reflecting the CPP’s popularity, we provide a projection with textual CPP annotations. Since most developers are familiar with the CPP, yet not with PEOPL’s other external representations, starting in a CPP projection is a useful option.

**CONCEPTUAL DETAILS AND SYNTAX.** Figure 3.1a shows our textual annotation projection. Notice that PEOPL allows developers to implement various undisciplined annotations—that is, annotations that do not mark one or a sequence of entire classifiers, classifier members, and statements as variable (cf. Sec. 2.4.2.1). Lines 4–8, 20, 24 show such undisciplined annotations on types, method parameters, and wrappers. Recap that the latter are program elements wrapping a block of code in a body (wrappee). Such wrappers can be either annotated completely or partially (without annotating the wrappee).

**EDITING OPERATIONS.** So developers see and work with `#ifdef` directives that are embedded into the program text. To annotate a code snippet, a developer simply types `#ifdef`, `#elif` or `#endif`—which directly creates feature artifacts in the underlying variational AST (as we explain in Ch. 7). Then, the developer chooses a feature or a combination of features, declared before (Fig. 3.1b) as the presence condition. In fact, the feature’s name can be either typed or selected from a code completion menu.

### 3.3.2 Visual Annotation Projection

When `#ifdef` directives clutter code and challenge comprehension, we can switch to visual annotations. The learning curve is low, since `#ifdef` directives and visual annotations can be explored for the same feature artifact in parallel.

**CONCEPTUAL DETAILS AND SYNTAX.** Figure 3.1c shows a projection of the class `DatabaseImpl` with annotations represented as colored bars, each of which relates to a feature (combination) declared in Figure 3.1b. For instance, light-gray bars relate to the feature *Base*. We

distinguish vertical and horizontal bars. Vertical bars are shown at the left of the program code and align with its indentation. Notice that each code snippet is colored by at least one vertical bar (i.e., PEOPL enforces that each code snippet is contained by at least one feature artifact). Wide vertical bars mark statements that relate to the code-base's feature artifact (e.g., Lines 4 and 10). This way, it is easy to identify the base implementation. Horizontal bars in turn underline fine-grained feature artifacts within a line of code (e.g., the method call parameter in Line 14) and partially annotated wrappers (e.g., the try/catch statement in Line 16). Moreover, to make alternatives explicit, we use a  $\oplus$ -sign (Line 9). Notice that using such graphical elements in the concrete syntax is a direct benefit of projectional editing, which allows mixing textual and visual syntax elements.

Notice that we decided against using background colors for visualizing annotations in PEOPL, because of their problems (cf. Sec. 2.5.2), and our own (negative) experiences of using them. In fact, we argue that subtle, colored vertical and horizontal bars—embedded into the source code—are more suitable to visualize annotations:

1. Bars are more elegant than background colors and less prone to obscuring the annotated source code. In fact, horizontal bars integrate well with the surrounding source code (not overlapping with text) and vertical bars align automatically with the indentation. Moreover, feature artifacts can be unambiguously nested—that is, showing multiple bars in parallel, next to each other. As a result, the features involved are easy to distinguish. Background colors are less concise and hamper comprehension as they are typically blended to visualize nested annotations.
2. Regardless of a bar's coloring, the source code remains readable. For instance, dark and saturated bars do not pose any problems, but dark and saturated background colors make the code impossible to read.
3. To ease distinguishing features, all bars can be easily augmented with additional feature information on demand—that is, the corresponding feature's name is shown in the code (cf. Fig. 3.2, Lines 1, 14, 21, and 27). Distinguishing the different features using only background colors is more difficult. Although CIDE mitigates this issue by showing a tooltip with feature information on a mouse-over, it is still time-consuming and cumbersome to grasp feature relations this way.
4. We augment colored bars with the  $\oplus$ -sign to make alternatives and their default values explicit. For instance, in Figure 3.2, Line 12, `void` is the `preload-method`'s default return type. Such default values must be assigned by the developer. In contrast, solely relying on background colors is ambiguous and neither



```

1  ▶ Base
2  class DatabaseImpl {
3      static class PreloadProcessor {
4          void processLSN(long c, LogEntTyp t){
5              assert c != DbLsn.NULL_LSN
6              ▶ Memory_Budget
7              if(envImp.getMemoryBudget().get()) //1) more code
8              ▶ Statistics
9              if(childType.equals(...)) //2) more code
10             }
11         }
12         void ⊕ PreloadStats (▶ Statistics) preload(PreloadConfig c) {
13             long maxByte = c.getMaxByte(); //3) more code
14             ▶ Memory_Budget
15             if(maxByte == 0) //4) more code
16             ▶ Statistics
17             PreloadStats ret = new PreloadStats();
18             PreloadProcessor cb = new PreloadProcessor(
19                 envImp, maxByte, tTime, ▶ Statistics);
20             //5) more code
21             try (▶ Statistics) {
22                 walker.walk();
23             } catch (HaltPreloadException HPE) {
24                 ret.status = HPE.getStatus();
25             }
26             // some more code in the DatabaseImpl |
27             ▶ Latches
28             //6) more code
29             ▶ Statistics
30             return ret;
31         }
32     }

```

Editing Operations Menu  
> Assign Variability to Comment  
> Assign Alternative to Comment

Figure 3.2: Visual annotation projection with support for toggling feature information

suitable for comprehending alternative code pieces nor default values.

On a final note on the concrete syntax, the Spotlight tool—which is outside the SPL context—also uses vertical bars to visualize annotations [47, 48]. However, the bars are placed in the IDE’s left margin and thus do not align with the source code. Thus, it is questionable whether the approach scales for a larger number of nested annotations. Moreover, there is no support for horizontal bars, and thus inter-line annotations are not supported.

**EDITING OPERATIONS.** To (partially) annotate a code snippet or assign an alternative to it, developers use a so-called *editing operations menu* (a.k.a. *intentions menu*)—which is a pop-up over a code snippet, available in all projections. Figure 3.2 shows such a menu over a comment in our Berkeley DB example (Line 26).

The figure also shows that developers can toggle feature information into the source code, which allows them to easily assign a feature artifact to another feature. We simply select the desired feature’s name from a code completion menu, or type its name. Notice that toggling feature information into the source code is not restricted to the visual annotative projection, but also available in all other projections that are enriched with colored bars.

### 3.3.3 Feature Module Projection

Now, imagine we want to evolve the *Memory\_Budget* feature or fix a bug in it, without being distracted by irrelevant code. Obviously, it is beneficial to edit the class *DatabaseImpl* in isolation and therefore switch from the annotative projection to the feature module implementing *Memory\_Budget*.

**CONCEPTUAL DETAILS AND SYNTAX.** Figure 3.1d shows our feature module projection. As in AHEAD, the *refines* keywords indicate that the module *Memory\_Budget* modifies the class *DatabaseImpl*, the inner class *PreloadProcessor*, and its method *processLSN* (Lines 2–8). As in FH, the *original-keyword* is used to refer to the refined method declaration (cf. Sec. 2.4.3). Moreover, we propose to show complexity indicators, such as a warning sign, in modules to make developers aware of potential ordering issues (not depicted). In fact, methods that are refined multiple times—that is, a *LONG REFINEMENT CHAIN* (cf. Sec. 2.5.2.2)—are prone to ordering issues, and thus a complexity indicator might help.

**EDITING OPERATIONS.** Using the modular projection, developers introduce feature artifacts by simply typing the desired code elements. In fact, PEOPL automatically assigns the typed code to the respective feature in the internal representation. Refining existing code that is introduced by an external feature module is also easy. A developer simply selects the accessible, external class, or method to be refined from a popup-menu. Then, the keyword *original* can be typed to refer to the original implementation, which internally restructures the underlying AST.

### 3.3.4 Blending Annotation and Module Projections

Notice that Figure 3.1c shows several fine-grained feature artifacts, such as scattered base code (Lines 10, 13–15, and 17), alternative return types (Line 9), and parameter variability (Line 14). These cannot be implemented without workarounds in classical modular approaches (cf. Sec. 2.4.3 and [102]). Although we could explore the annotative and the modular projection in parallel, it might be beneficial to allow integrating annotation markers into modules.

**CONCEPTUAL DETAILS AND SYNTAX.** Figure 3.1d shows a blending projection, which enriches a feature module with statement-level markers (Lines 9–12). To avoid obfuscation, only *Memory\_Budget* code is shown. All other code is hidden—that is, code of external modules is collapsed into colored bars. This way, the granularity trade-off of modular approaches can be addressed and fine-grained changes im-

plemented. Since the surrounding, external code of other modules might be important for comprehension, we allow developers to

- expand a selected or all markers showing the content of annotations, and
- project accessible fields and forward method declarations directly into the module on demand.

This way, developers get context-information that may help to better understand the source code, how the feature interacts with its surroundings, and the possible implementation options of a module. For instance, the variable `maxByte` in Line 11 of Figure 3.1d is declared in the hidden base code—as shown in Line 10 of Figure 3.1c. To ease comprehension, expanding the respective marker is possible.

**EDITING OPERATIONS.** Editing is as simple as in the pure feature module projection. In addition, the editing operations menu enables to show markers, and expand them on demand.

### 3.3.5 Variant Projection

Now, imagine we want to evolve the features *Base* and *Memory\_Budget* or fix a bug that occurs when both features are enabled. We could show all feature artifacts of *Base* by expanding the respective annotation markers in the module *Memory\_Budget* (Fig. 3.1d), or switch to a corresponding variant editor.

**CONCEPTUAL DETAILS AND SYNTAX.** Figure 3.1e shows our reactive variant projection, which allows exploring variant-specific code and control flows in isolation. To understand which code artifact implements which feature, we can show colored bars in the variant as depicted in Figure 3.3. This way, code can be unambiguously assigned to features in a proactive fashion.

**EDITING OPERATIONS.** Edits in the reactive variant editor are currently assigned to features without the developers awareness—that

```

Variant projection for Base and Memory_Budget
1  class DatabaseImpl {
2      static class PreloadProcessor {
3          void processLSN(long c, LogEntTyp t){
4              assert c != DbLsn.NULL_LSN
5              if(envImp.getMemoryBudget().get()) //1) more code
6          }
7          void preload(PreloadConfig c) {
8              long maxByte = c.getMaxByte(); //3) more code
9              if(maxByte == 0) //4) more code
10             PreloadProcessor cb = new PreloadProcessor(
11                 envImp, maxByte, tTime);
12                 //5) more code
13             walker.walk();
14         } }

```

Figure 3.3: Proactive variant editor with colored bars

is, developers do not see to which feature an edit belongs. Thus, the proactive variant representation should be used for editing instead. In fact, it provides the same variability-aware editing operations as the annotative editor.

### 3.3.6 *Fade-in Feature Module Projection*

Aside from realizing the aforementioned “standard projections”, we also experimented with the concrete syntax of projections (i.e., an inherent advantage of separating the internal variability representation from the external representations). According to our own experience, we found that blending annotations into feature modules sometimes obscures the code too much. In fact, it might be irrelevant for the developer to which feature an annotated code snippet (or the respective marker) belongs. For instance, in the Stack SPL example in Figure 3.4a (cf. Fig. 2.14), it might be irrelevant that the `if`-statement belongs to the codebase (Line 8), or that there are four different markers (Lines 11–14). Thus, we conceive a projection fading modules—that is, integrating code of external modules (whenever necessary) into a module, while hiding external feature relationships.

**CONCEPTUAL DETAILS AND SYNTAX.** Figure 3.4b shows a fade-in feature module for our fine-grained Stack SPL example (cf. Fig. 3.4a and Fig. 2.14). The module provides a coherent view on all content belonging to the feature *Synchronization*, while supporting to visualize fine-grained feature artifacts. The concrete syntax is similar to what we know from DeltaJ (cf. Sec. 2.4.3.2). In particular, we adopted the keywords `adds` and `modifies` for clarity reasons—that is, making introductions and modifications explicit. Notice that PEOPL also supports the removal of feature artifacts, but currently not via a respective keyword (as we will discuss in Ch. 4). To ease the identification of code that belongs to the current module, any code snippet that is introduced by an external feature module is represented light gray. Moreover, coherent external feature code is collapsed into a textual marker on the statement level (i.e., `<ExternalFeatureCode>`, Line 10). Thus, multiple markers may appear. To ease comprehension, sub-statement feature artifacts are always shown in their context. For instance, it is straightforward to comprehend that the module *Synchronization* adds the method parameter, the `throws-item`, and the expression (Lines 6 and 7).

**EDITING OPERATIONS.** Modifying an existing external class or method corresponds to the process provided with the standard feature module—that is, a pop-up menu allows to select classes and methods introduced externally. On the statement-level, the markers can be expanded on demand using the editing operations menu, for instance to add a new (fine-grained) feature artifact.

```

1 module Synchronization
2 refines class Stack {
3   private final ReentrantLock lock = new ReentrantLock();
4   private final Condition notEmpty = lock.newCondition();
5
6   refines public void push(Object data, ReentrantLock lock)
7   throws InterruptedException {
8     if(data == null || lock == null) throw new NullPointerException();
9     lock.lock();
10    try {
11      |
12      |
13      |
14      |
15      notEmpty.signalAll();
16    } finally { lock.unlock(); }
17  }
18 }

```

(a) Feature module with annotations to mark fine-grained feature artifacts

```

1 module Synchronization
2 modifies class Stack
3   adds private final ReentrantLock lock = new ReentrantLock();
4   adds private final Condition notEmpty = lock.newCondition();
5
6   modifies push(Object data, ReentrantLock lock) throws InterruptedException
7   if(data == null || lock == null) throw new NullPointerException();
8   lock.lock();
9   try {
10    <ExternalFeatureCode>
11    notEmpty.signalAll();
12  } finally { lock.unlock(); }

```

(b) Fade-in module with fine-grained feature artifacts (code introduced by external feature modules is gray)

Figure 3.4: Implementing fine-grained feature artifacts by blending annotations (markers) into feature modules (a), or using an fade-in feature module (b)

### 3.3.7 Reuse Projection

There is still one use case that is not properly supported by solely using the aforementioned projections. Assume that we want to reuse a concrete feature artifact (multiple times). If we copy and paste the artifact, we introduce a code clone, which hampers maintenance activities, such as detecting [124] and fixing bugs [96]—that is, developers may fix a problem in one clone instance, but miss another instance. Thus, it would be beneficial to reuse a feature artifact by letting it “appear” in other program locations. This way, a change to the feature artifact itself or one of its appearances is consistent, since any edit directly changes the sole physical artifact in the internal representation. We discuss the necessary fundamentals, and our projection, which enables the desired reuse next.

**FUNDAMENTALS.** Basically, implementing a feature means implementing program extensions that either add new structural elements or replace existing elements [14]—as opposed to a program change, which also enables removing elements. As such, an extension comprises a set of feature artifacts, for instance new classes, methods, and statements; or even replacements for other classes and methods.

The literature often distinguishes program extensions into heterogeneous and homogeneous extensions [9, 49]. Figure 3.5 illustrates this differentiation. A heterogeneous extension adds different feature artifacts to different “extension points” in the SPL’s codebase (contained by other feature artifacts). Such extension points are typically not specified explicitly, but either from within the extension itself (feature modules) or by annotating the codebase. In contrast, a homogeneous extension uses the very same feature artifact to extend various positions in the codebase. A prominent approach focussing on homogeneous extensions is *aspect-oriented programming (AOP)* [111]. There, *aspects* have *advice*, which can be applied to multiple *join points*.

In real SPLs, heterogeneous extensions amount to over 90% [126] to 98% [5] of the codebase. As such, SPL engineering typically focusses on heterogeneous extensions—which are not supported well by AOP [101]. All in all, support for rare homogeneous extensions is desirable to a certain degree.

**CONCEPTUAL DETAILS AND SYNTAX.** Our reuse projection adds homogeneous extension support to PEoPL. Figure 3.6 gives an example of a feature *Trace*, whose feature artifact prints a method’s name in different program locations. In fact, we reused the print-statement, instead of copying and pasting it. A reused (original) feature artifact is highlighted by a colored opening square bracket (Line 3), its appearance by a colored closing square bracket (Lines 7 and 11). Changes to the reused feature artifact appear directly in all reuse-positions (e.g., Lines 7 and 11). Likewise, editing an appearance directly changes the feature artifact. Notice that the reuse projection can be used within all projections we have discussed thus far. The design of more advanced visualization and editing facilities is subject of future work. For instance, we could allow developers to specify a homogeneous extension from within a module in an AOP fashion, or make changes to feature artifact appearances without changing the reused artifact.

**EDITING OPERATIONS.** Employing the reuse projection is simple. Similar to copy and paste, we provide the operations *pick and appear*, which are available via a menu or respective keyboard shortcuts.

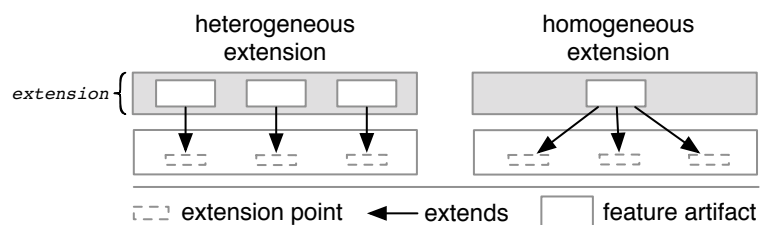


Figure 3.5: Feature artifacts can realize heterogeneous and homogeneous extensions (adapted from [14])

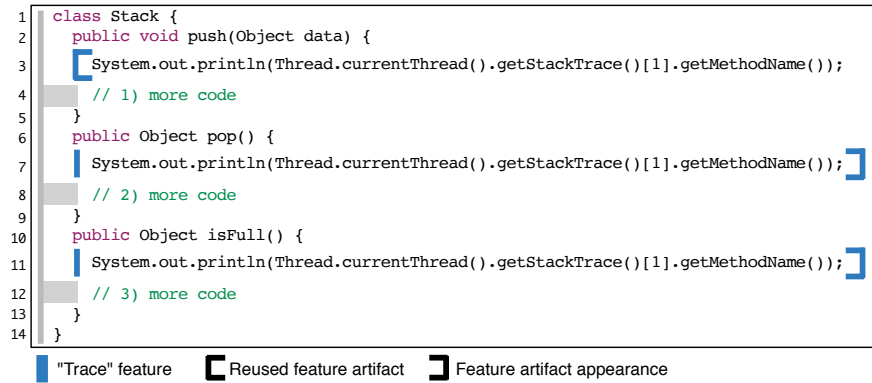


Figure 3.6: Feature artifact reuse instead of cloning

### 3.4 PROJECTIONS II: NON-CODE FEATURE ARTIFACTS

Projectional editing enables developers to use and combine diverse notations, for instance, program code can be enriched with graphical elements such as diagrams [199]. As such, PEOPL, which realizes a projectional approach, is not bound to text. In fact, it inherently allows assigning software artifacts to features irrespective of their type. Next, we demonstrate how we use PEOPL in the context of mathematical formulas [199] that are embedded into program text, and fault trees.

#### 3.4.1 Variational Mathematical Formulas

Figure 3.7 shows a simple example of using math in C code. The number  $\pi$  is calculated in three different ways using *Gregory and Leibniz's* formula, the *Madhava-Leibniz* series, and the *Monte Carlo Method*. In Figure 3.7b, the code is difficult to read, especially for inexperienced programmers, since the math is transformed into text. Using mathematical symbols [199] instead of program text substantial improves the comprehensibility of formulas as illustrated in Figure 3.7a. In fact, mathematical symbols, for instance,  $\sqrt{\quad}$  and  $\sum$  as used in  $\sqrt{12} * \sum_{k=0}^n \frac{(-3)^{-k}}{2^{*k+1}}$  (Line 16), can be simply selected from a code completion menu (after typing the symbol's name, e.g., sum).

Using PEOPL, we make the  $\pi$  calculation variational (Fig. 3.7a). Each formula relates to a feature named after the corresponding algorithm, while the three features are alternative to each other—that is, only one of them can be selected at a time, which is indicated in the code by the  $\oplus$ -sign (e.g., Line 13). Thus, we can generate C code calculating  $\pi$  using exactly one of these methods (depending on the algorithm choice).

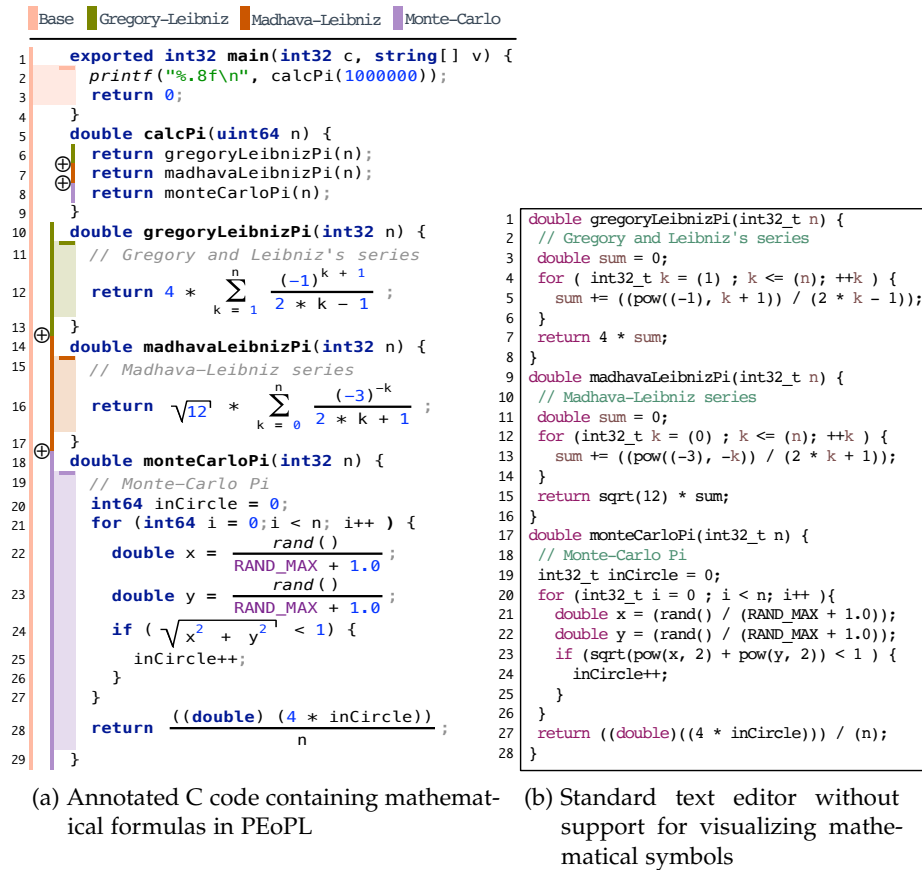


Figure 3.7: Annotated C code containing mathematical formulas

On a final note, projects, such as ExaStencils<sup>1</sup>, highlight that assigning features to mathematical algorithms is indeed desirable, for instance, to avoid wasting memory or to increase performance [84, 123]. Additionally, visualizing mathematical symbols using projectional editing positively affects the comprehensibility of complex mathematical formulas.

### 3.4.2 Variational Fault Trees

On the example of fault trees, we now show how PEOPL is used to assign more complex non-code artifacts to features. We give a quick overview on *fault tree analysis (FTA)*, outline the textual and graphical representations we use for constructing fault trees, and show how we deal with variability in fault trees using annotations, modules, and variants.

<sup>1</sup> <http://www.exastencils.org/>



### 3.4.2.1 *Fault Tree Analysis: A Quick Overview*

Safety-critical systems perform control-tasks that in case of failure lead to catastrophic consequences, such as serious harm to humans or the environment. Such systems must be certified by authorities before the safety-critical system can be used. Typically, for a system to be certified, it requires a rigorous assessment of possible risks. FTA is one way to assess risks and, in fact, the causes of hazards. FTA is a deductive technique to identify and analyze failure paths of systems [191]. Constructing a fault tree amounts to identifying and expressing the root causes for a given hazardous event utilizing Boolean logic. In fact, a fault tree is a logic equation of causes (a.k.a. *basic events*), where each cause has a probability to occur, and thus the probability of such a hazardous event can be calculated.

Moreover, safety critical systems typically contain a lot of variability, since different customers have different requirements. Yet, certification authorities deal with single systems, and thus it is desirable to cope with variability in fault trees as well. In fact, it is worthwhile to derive a product-specific fault tree automatically instead of constructing fault trees for each product from scratch. To enable variability in fault trees, previous work proposed to use *annotations* [55, 161, 180], *components* [98], *meta-modeling* [81], and *deltas* [167]. Yet, ideally developers could move fluently and transparently between different variability-aware fault tree representations (i.e., annotations, modules, and variant-specific). To address this desire, we use PEOPL, and thus underline its independence of a specific programming-language and artifact-type.

### 3.4.2.2 *Textual and Visual Fault Tree Representations*

To construct fault trees in PEOPL, we conceive a flexible fault tree notation, which is based on our previous work [26, 132]. We differentiate a fault tree into an internal and two editable external parts (projections) that developers use. In fact, developers can work with and fluently switch between a textual fault tree representation (program text) and a graphical representation (diagrams). While the former allows making complex changes in an efficient manner, the latter typically eases comprehension. We discuss both projections next.

**TEXTUAL FAULT TREE PROJECTION.** The textual notation allows creating fault trees in a programmatic fashion. In general, a textually specified fault tree has the following shape:

```

TopEvent [<top event description>]
  OrGate [<gate description>] [ with { <list of event ids> } ]
    AndGate [<gate description>] [ with { <list of event ids> } ]
      NotGate [<gate description>] [ with { <event id> } ]

```

---

**BasicEvent** <event id> = <event description> **has** <probability>

We use these language constructs to create fault trees. Developers start with a `TopEvent`—the hazard—to which different gates, such as `OrGates` and `AndGates` can be attached. A gate is a Boolean operator, representing an intermediate event (i.e., an intermediate cause of the hazard to occur). In fact, the gates are nested using indentation to structure the tree (hierarchy). Moreover, each fault tree has a set of `BasicEvents`, each of which represents a root cause of the hazard that occurs with a certain probability. Notice that events have a unique ID, which is used to attach the event to a gate (or multiple gates). In fact, different gates can refer to the same basic event, and thus a directed acyclic graph may be constructed. Moreover, fault tree elements may be unassigned—that is, not every element needs to be a descendant of the top event.

As an example scenario, we adopt a fault tree for the personal robot `TurtleBot` used in previous research [26, 167]. In a possible hardware configuration, the robot has actuators for moving and bumpers to detect collisions. Figure 3.8 shows a corresponding fault tree using the data from the `TurtleBot` example [167]. The hazardous top event is a *catastrophic collision* (Line 1) caused by the two intermediate events *breaking fails* and *bumped into obstacle* (Lines 2–4). In fact, breaking may fail if the *robot is moving* on a *low friction surface*, such as ceramic tiles or a wet floor (Lines 6–7). Moreover, the robot bumped into an obstacle if (i) the robot is moving, (ii) there is an *obstacle in its way*, and (iii) we indeed have *detected a bump* (Line 8–9).

Notice that the four basic events have a probability assigned that enables calculating the probability of a catastrophic collision. To calculate the probability, we first calculate the so-called *minimum cut set* [191]—the smallest conjunctions of basic events causing the top event. In fact, the fault tree description of the `TurtleBot` can be translated into Boolean logic as follows:

$$\text{collision} := (\text{lfs} \wedge \text{rim}) \vee (\text{rim} \wedge \text{bum} \wedge \text{oiw})$$

This equation is already our minimum cut set, since there is no smaller conjunction of basic events causing the top event. Notice that we would use a top-down approach using the rules of Boolean

```

1 TopEvent "Catastrophic collision"
2 OrGate
3   AndGate "Breaking fails" with { rim, lfs }
4   AndGate "Bumped into obstacle" with { rim, bum, oiw }
5
6 BasicEvent rim = "Robot is moving" has 0.8
7 BasicEvent lfs = "Low friction surface" has 0.02
8 BasicEvent bum = "Bump detect" has 0.95
9 BasicEvent oiw = "Obstacle in way" has 0.15

```

Figure 3.8: Example textual fault tree for a robot with actuators and bumpers

algebra to determine the cut set otherwise. Then, the probability of the top event is the union of the cut sets [191]:

$$collision = (0.02 * 0.8) + (0.8 * 0.95 * 0.15) = 0.13$$

GRAPHICAL FAULT TREE PROJECTION. Aside from implementing the tree textually, we can use a graphical representation. In fact, the graphical notation allows creating fault trees as diagrams. In general, a graphically specified fault tree has the following shape:

Type	Graphical Element	In	Out
TopEvent	□ [<top event description>]	1	0
OrGate	≥1 [<gate description>]	n	1
AndGate	& [<gate description>]	n	1
NotGate	¬ [<gate description>]	1	1
BasicEvent	○ [<event id> [<event description>] [<probability>]	0	n
Reused BasicEvent	⊖ [<event id> [<event description>] [<probability>]	0	1
Connection	→	-	-

Notice that the graphical elements match the textual description. Developers start with a TopEvent and then add new gates and basic events in a step-wise manner. Figure 3.9 shows the graphical representation of the textual TurtleBot example (Fig. 3.8). Notice that connections between fault tree elements point towards the root. This way, we clarify that basic and intermediate events are the cause for other intermediate events (or the top event).

On a final note, other approaches to construct fault trees typically either allow a textual or a graphical editing of the data structure (not side-by-side in parallel). Textual approaches, such as the Haskell

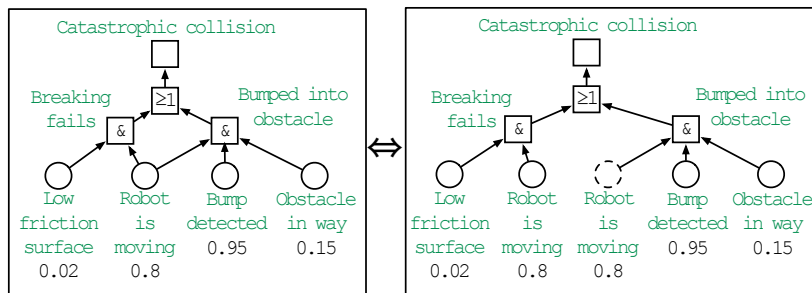


Figure 3.9: Graphical fault tree for a robot with actuators and bumpers. On the left, the reuse of the basic event *Robot is moving* is represented by multiple output edges. On the right, the reuse is made explicit by a dotted circle which eases comprehension in complex trees (or more precise graphs). Both representations are logically equivalent.

fault tree package<sup>2</sup>, only support rendering the textual notation into a graphical notation that cannot be edited (e.g., using Graphviz<sup>3</sup>). Visual fault tree representations—provided by tools such as OpenFTA<sup>4</sup>, FaultTree+<sup>5</sup>, or the RAM Commander<sup>6</sup>—are typically realized using a rendering engine, which is tailored to graphical editing. Consequently, developers cannot move fluently between the different representations.

### 3.4.2.3 Projecting Variability into Fault Trees

Now imagine that we construct three different hardware variants of the TurtleBot (cf. [167]), each of which has actuators to move, but a different sensor setting:

V1: Bumpers (to detect collisions)

V2: A laser scanner (to detect obstacles in the robot's range)

V3: Bumpers and a laser scanner (to detect obstacles and collisions)

Obviously, we need a respective fault tree for each variant (for certification). Instead of creating each variant from scratch, potentially cloning fault tree elements, we create an SPL, where the two sensors are represented by the features *Bumper* and *LaserScanner*, respectively. Then, we can select the features and create the variant's fault tree automatically. We now discuss how developers can work with variability-aware fault trees using the annotative, modular and variant projections we conceived in PEOPL.

**ANNOTATION PROJECTION.** Using PEOPL, variational fault trees can be constructed by annotating the fault tree's external textual and graphical representation. Figures 3.10a/b show textual and graphical fault trees for the TurtleBot SPL. Annotations are represented as colored bars. Notice that we only color gates. The rationale is that if a gate is removed during variant derivation, the references to the basic events are removed as well—which typically is sufficient. For instance, if the feature *Bumper* is not selected, then the and-gate *Bumped into obstacle* is together with the references to the basic events removed.

**VARIANT PROJECTION.** Now imagine that we want to understand what a specific fault tree variant looks like. We simply switch to the textual (e.g., Fig. 3.10c) or graphical (e.g., Fig. 3.10d) variant projection. In fact, comparing the graphical annotation projection

<sup>2</sup> <https://hackage.haskell.org/package/faulttree>

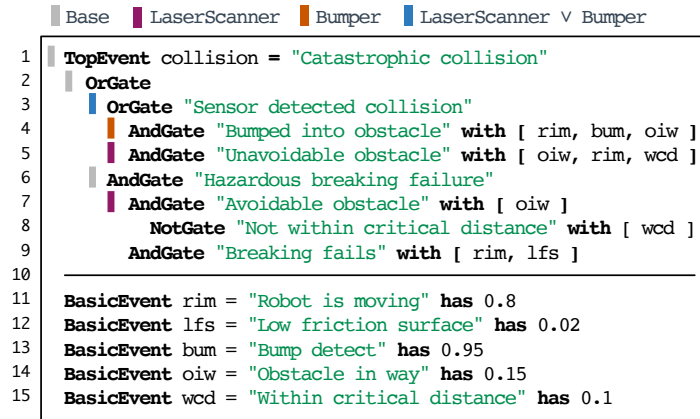
<sup>3</sup> <http://www.graphviz.org/>

<sup>4</sup> <http://www.openfta.com/>

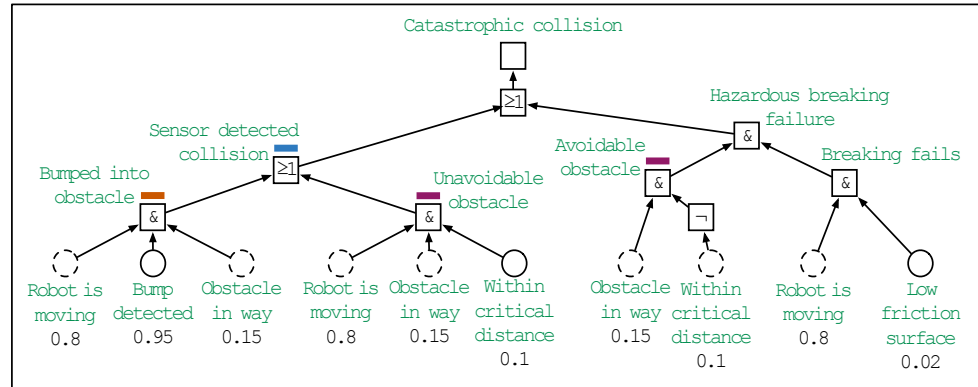
<sup>5</sup> <https://www.isograph.com/software/reliability-workbench/faulttree-analysis/>

<sup>6</sup> <http://aldservice.com/Fault-Tree-Analysis-FTA-Software.html>

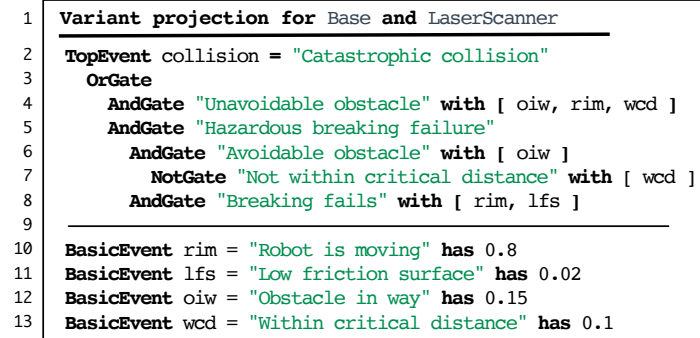
**a) Textual annotation projection**



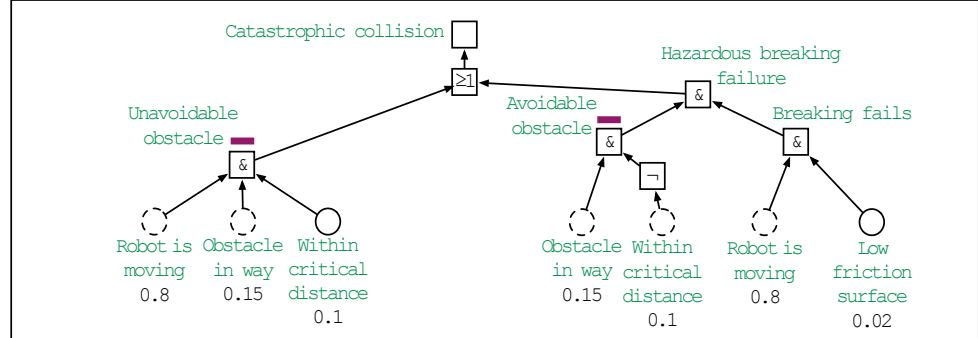
**b) Graphical annotation projection**



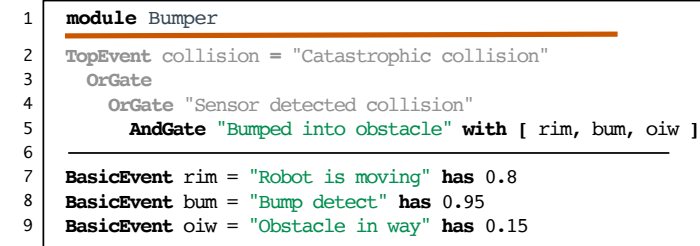
**c) Textual uncolored variant projection (reactive)**



**d) Graphical colored variant projection (proactive)**



**e) Textual module projection**



**f) Graphical module projection**

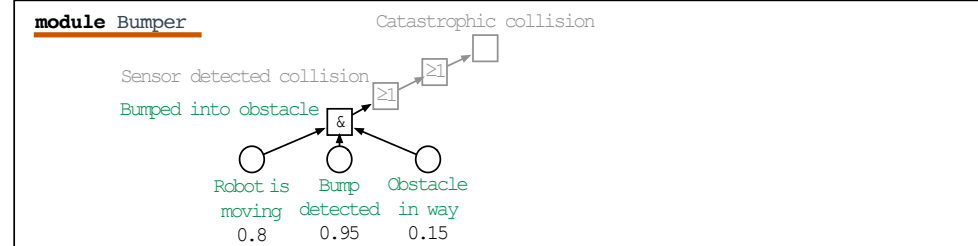


Figure 3.10: Variational fault tree for a robot rendered into different external variability representations

(Fig. 3.10b) and the graphical variant projection (Fig. 3.10d) side-by-side—which is possible in PEOPL—eases comprehension. Notice that the gate *Sensor detected collision* is not rendered, since it is obsolete for the variant—that is, *Unavoidable obstacle* is the only input gate.

**MODULE PROJECTION.** When developers want to understand and edit a fault tree of a specific module in isolation, they simply switch to the textual or graphical modular projections. For instance, in our TurtleBot example, we can observe the fault tree of the module *Bumper* in isolation as illustrated in Figures 3.10e/f. To understand how the module contributes to the hazard the path to the top event is shown (in grey). Moreover, notice that in both, the textual and the graphical projection, only the referenced basic events are visible. To refer to another basic event, developers can blend available events into the module.

### 3.5 PROJECTIONS III: VARIABILITY-AWARE FILE EXPLORERS

So far, we only discussed variability within files (i.e., on source code snippets). However, developers need to be able to explore an SPL's files in a variability-aware manner as well. To deal with annotative, variant, and modular representations, we conceive respective file explorers, as discussed next.

**ANNOTATIVE REPRESENTATIONS.** For exploring annotated files, a standard file explorer is sufficient, since it inherently shows all files of the SPL. Figure 3.11a shows an example file explorer for Berkeley DB (cf. Fig. 3.1), where all source files (e.g., our example class *DatabaseImpl*) are organized in respective packages. The only addition we make to the standard explorer is that the respective top level feature is shown after each file's name—which is possible, since each code snippet is contained by at least one feature artifact. Thus, it is easy to identify that, for instance, the class *BtreeStats* is introduced by the *Statistics* feature and the class *DatabaseImpl* by the *Base* feature (cf. Fig. 3.1).

**VARIANT REPRESENTATIONS.** For exploring the files of a specific variant, we use a standard explorer, which hides the files unrelated to a feature selection. For instance, the variant file explorer depicted in Figure 3.11b shows the files for selecting only the *Base* feature. Since the *Statistics* feature is not selected, the class *BtreeStats* is hidden.

**MODULAR REPRESENTATIONS.** Exploring the files introduced or modified by a feature is also straightforward. In fact, our modular file explorer organizes the SPL's implementation into feature modules, each of which shows the files it introduces and modifies. For instance, Figure 3.11c shows the four features *Base*, *Memory\_Budget*, *Statistics*, and *Latches* of our Berkeley DB example. To distinguish in-

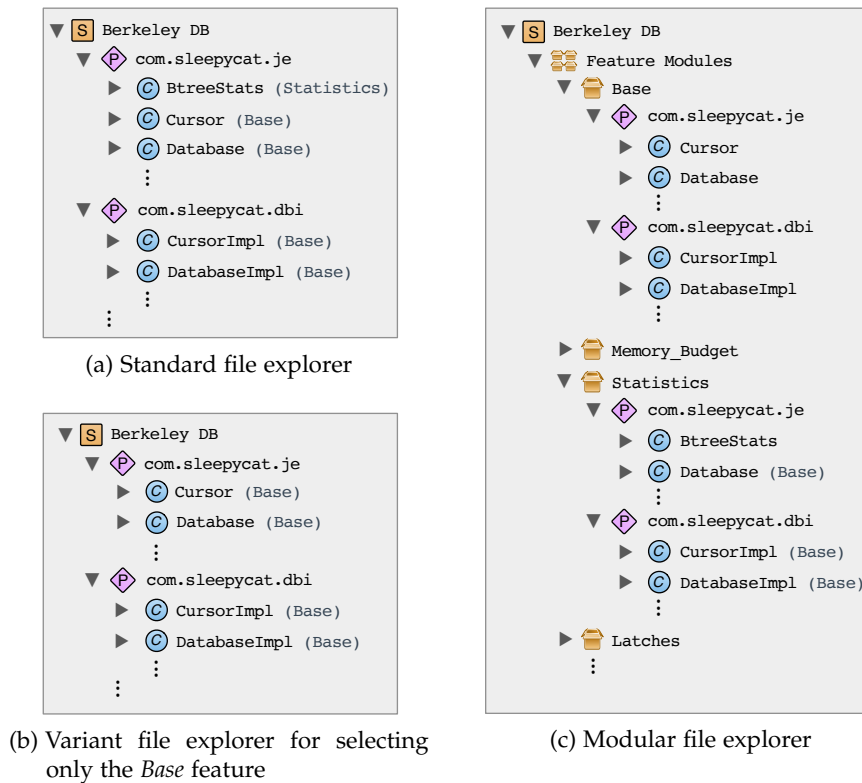


Figure 3.11: Different variability-aware file explorers supported by PEOPL

roduction and refinement, we extend the modified file’s name with the external module’s name—that is, the module, which introduced the file. In our example, it is, in fact, easy to distinguish introductions and modifications. For instance, the module *Base* introduces the file `DatabaseImpl`, while *Statistics* modifies it (indicated by the file name’s suffix).

### 3.6 CONCLUDING REMARKS

We have seen that PEOPL enables developers to implement, comprehend, maintain, and evolve SPLs in diverse ways. Establishing an internal variability representation and separating it from the external representations that developers use, enables a very flexible and novel way of editing SPLs. In fact, PEOPL allows developers to leverage the advantages of annotations, modules, and variant-specific SPL editing, and thus helps addressing common variability engineering challenges. We discussed PEOPL’s diverse variability projections, their individual strength and weaknesses, and possible usage scenarios. We have shown how developers can browse the SPL using variability-aware file explorers, and that PEOPL is not restricted to code artifacts, but enables adding variability to arbitrary textual and graphical elements. Aside from implementing variability, PEOPL supports modeling and managing variability as we will see in the following chapter.

MODELING AND MANAGING VARIABILITY

---

We now discuss how developers use PEOPL to setup a product line and derive variants (Sec. 4.1). Moreover, we discuss how PEOPL helps developers to deal with an SPL's inherent complexity and the correctness of variants (Sec. 4.2).

## 4.1 PRODUCT LINE SETUP AND VARIANT DERIVATION

To model an SPL, PEOPL provides DSLs for a simple and an advanced SPL setup. Both ways for configuring an SPL are projections of a common, internal configuration representation, which can be directly edited by experts (cf. Sec. 8.2.1). Notice that all projections enable declaring feature modules to which developers refer from the SPL's implementation, but internally only one module declaration per module exists. Thus, developers can switch from a simple declaration to an advanced declaration, if necessary without redeclaring the SPL's feature modules and changing the implementation. In the following, we discuss the simple and advanced ways for modeling an SPL.

4.1.1 *Simple Product Line Declaration*

A simple setup of an SPL has only two types of declarations: feature modules and constraints over feature modules.

**FEATURE MODULE DECLARATIONS.** Each feature (and feature interaction) maps to exactly one feature module declaration as illustrated in our Berkeley DB example in Figure 3.1b. Subsequent to its declaration, a feature module is implemented using a set of concrete feature artifacts (each of which refers to the module's declaration). For instance, the *Base* feature module comprises five feature artifacts (Fig. 3.1c, Lines 4, 9, 10, 13–15, and 17).

Since the feature artifacts of different modules can be alternative to each other (e.g., the return types `void` and `PreloadStats` in Fig. 3.1c, Line 9), developers need to be able to specify which module's artifacts shall be included in a variant. To decide between such mutually exclusive artifacts, developers declare feature modules in a certain ordering—that is, developers implicitly assign a priority to each feature module. Then, the module with the higher priority overrides the conflicting artifacts of the lower prioritized module. According to the ordering in our example, *Latches* has the highest priority and *Base* the lowest (Figure 3.1b). Consequently, the module *Statistics* will *override*



conflicting artifacts defined in *Base*. In fact, *PreloadStats*, and not *void* will be included in a variant with both feature modules being selected. Yet, if we changed the priority such that *Statistics* had the lowest priority (i.e., move the declaration to the top), the return type *void* would be included in respective variants.

During variant derivation, all modules are composed while resolving conflicts according to their priority (i.e., bottom-up). In fact, composing modules results in a greater set of feature artifacts—that is, the artifacts included in the target variant. For instance, if all four feature modules of our example are selected (using checkboxes), then *Latches* is composed with *Statistics*, the resulting larger module with *Memory\_Budget* and so on, and thus all artifacts (except *void* in Line 9) are included in the variant.

**CONSTRAINT DECLARATIONS.** To enforce correctness of the SPL, developers can constrain the selection of features. A constraint is a propositional formula over feature modules (cf. Sec. 2.3) which enables developers to express dependencies and relationships between (multiple) features. In PEOPL, constraints can be built using the following operators:

- ! (not)
- && (conditional and)
- || (conditional or)
- \*|| (one-out-of-many)
- $\Rightarrow$  (implies)

For instance in our Berkeley DB example (Figure 3.1b), we could use the constraint *Base && Memory\_Budget* to define that any variant requires the feature modules *Base* and *Memory\_Budget* to be selected.

#### 4.1.2 Advanced Product Line Declaration

If the simple product line declaration does not suffice, developers can switch to an advanced declaration, which has been inspired by delta-oriented product line declarations [115, 157], and thus has a similar shape (cf. Sec. 2.4.3.2). Figure 4.1 shows the structure of an advanced declaration, which comprises the following elements:

- a set of features characterizing the SPL (Line 2)
- a set of feature modules (Line 3) which are referenced by developers from the implementation to mark feature artifacts
- a set of constraints over features using propositional logic to declare a feature model and enforce behavioral correctness of the SPL (Line 4)

```

1 SPL <SPL name> {
2   Features { <list of features> }
3   Modules { <list of feature modules> }
4   Constraints { <list of feature constraints using propositional logic> }
5   Mappings {
6     <list of modules> when (<Boolean expression over features>);
7     ...
8   [ <list of modules> removal of artifacts when (<Boolean expression over features>); ]
9     ...
10  }
11  Variants {
12    <variant name> = { <list of feature selections> }
13    ...
14  }
15 }

```

Figure 4.1: The shape of PEOPL’s advanced product line declaration

- an ordered set of mappings from features to feature modules (Lines 6 and 8) with each individual mapping comprising a list of modules that add, override, or remove feature artifacts if the corresponding expression over features evaluates to true
- a set of valid variant configurations, each of which comprises a selection of features (Line 12)

To derive a variant, all modules—whose Boolean expression over features evaluate to true for a given feature selection (Line 6)—are applied according to their priority (i.e., bottom up). The result is a transient variant set of feature artifacts. Then, in a second step, the modules that remove feature artifacts are applied (if their Boolean expression over features evaluates to true). The result is a final variant set of feature artifacts. In a final step, to transform the SPL into the variant, feature artifacts that are not in the variant set are removed.

Figure 4.2 shows an advanced SPL declaration for our Berkeley DB example (cf. Figure 3.1). We declare a set of features (Line 3) and a set of modules (Lines 6–7). Notice that there is no ambiguity between feature and feature module declarations. In fact, we can even use the same names, since either features or feature modules, but not both, can be referred from a clause. For instance, a constraint can only be declared, using a set of features (e.g. Line 11). Most features in the example are mapped to a corresponding module (with the same name)—that is, a simple one-to-one mapping (Lines 14–18). Moreover, to add coordination code for the features *Statistics* and *Memory\_Budget*, we include the module *StatisticsAndMemory\_Budget* when both features are selected (Line 19).

Now assume that stakeholders can select a feature *Optimization* (Line 3), which improves the performance of the database. In fact, imagine that selecting the feature triggers the removal of some feature artifacts of the module *Statistics*. To enable removal, developers mark the implementation artifacts to be removed using a module. Notice that we internally use an alternative to realize removal (as we will

```

1 SPL BerkeleyDB {
2   Features {
3     Base, Statistics, Memory_Budget, Evictor, Latches, Optimization, ...
4   }
5   Modules {
6     Base, Statistics, Memory_Budget, Evictor, Latches,
7     RemStatistics, StatisticsAndMemory_Budget, ...
8   }
9   Constraints = {
10    Base && Evictor ;
11    Memory_Budget => Evictor && Latches
12  }
13  Mappings {
14    Base when (Base);
15    Memory_Budget when (Memory_Budget);
16    Statistics when (Statistics);
17    Latches when (Latches);
18    Evictor when (Evictor);
19    StatisticsAndMemory_Budget when (Statistics && Memory_Budget);
20    RemStatistics removal of artifacts when (Optimization);
21  }
22  Variants {
23    Basic = { Base, Memory_Budget };
24    OptimizedBasic = { Base, Memory_Budget, Optimization };
25  }
26 }

```

Figure 4.2: Advanced product line declaration for the Berkeley DB example (cf. Sec. 3.1)

discuss in Ch. 7). To visualize removal in the code, we show a removal marker on the feature artifact in our visual annotative projection as illustrated in Figure 4.3. Notice that we did not use removal in our case studies (cf. Sec. 5.2.2), and thus only realized the projection as a proof of concept. Consequently, integrating a `removes` keyword in the modular projections (similar to the delta-oriented `removes-keyword` [115, 157]) is subject of future work.

On a final note, expert users can configure variants on an even more fine-grained level—for instance, removing feature artifacts at any composition state—using PEOPL’s underlying algebra as we will detail during the discussion of PEOPL’s internal representation (Ch. 7).

```

1  ▶ Base
2  class DatabaseImpl {
3    static class PreloadProcessor {
4      void processLSN(long c, LogEntTyp t){
5        assert c != DbLsn.NULL_LSN
6        ▶ Memory_Budget
7        if(envImp.getMemoryBudget().get())
8        ▶ Statistics (remove artifact if ▶RemStatistics)
9        if(childType.equals(...))
10       }
11     }
12   }

```

Figure 4.3: Enhanced visual annotation projection with a removal marker

## 4.2 PRODUCT LINE CORRECTNESS

A product line is a complex beast to deal with. In fact, unmet dependencies between features are a common cause for subtle errors [137, 152]. In order to minimize issues and avoid problems before deployment (i.e., while editing the SPL), PEOPL automatically detects artifact-related feature dependencies in the SPL’s entire implementation. Moreover, PEOPL allows analyzing the data flow of a variant to be deployed or all valid variants of the SPL. We discuss both next.

### 4.2.1 Artifact-related Feature Dependency Analysis

Prior work has shown that unresolved artifact-related feature dependencies can lead to syntactical errors and to type errors during product generation [12, 13, 106, 127, 147, 152, 184]. Recap that such a feature artifact dependency appears if a feature’s artifact depends on another feature’s artifact. There are multiple strategies to detect this issue, for instance, type checking and static analysis [186]. In our case studies (cf. Sec. 5.2.2), we also encounter a great need for resolving feature artifact dependencies. In particular, we adopt product lines by re-assigning features manually to the codebase in order to learn more about PEOPL and possible usability issues. This approach is error prone, and thus we need support for calculating and handling such dependencies in PEOPL. For instance, to detect, when we forgot to annotate a code snippet.

Based on our internal representation, we implement a set of simple checking rules that run with the type checker (as detailed in Sec. 9.1). To enable developers to handle dependencies, we add IDE support for navigating to dependent nodes and highlighting dependencies in the code. Figure 4.4 exemplifies the latter using a code snippet from our *graph-product-line (GPL)* case study. There, the variable reference `weight` is not annotated, and thus a feature artifact dependency between the features `GN_OnlyNeighbors` and `Weighted` emerges. In fact, if `GN_OnlyNeighbors` is selected, but not `Weighted`, a dangling reference appears. Notice that such missing annotations are difficult to detect without tool support, especially if a dependency spreads across methods and classes (a.k.a. *inter-procedural dependencies* [152]).

To enforce correctness of the SPL in the presence of intra- and inter-procedural code-level dependencies [152], classical approaches require developers to lift each dependency up into the variability model. PEOPL in turn extracts such dependencies automatically and informs developers about unresolved dependencies in the product line declaration. This way, variability model and variability implementation are clearly separated.

A quick note on internals. Relying on the concept of projectional editing has the potential of leveraging static analyses that are usually

```

1  ▶ Base
2  public class Graph { ...
3      EdgeIfc addEdge(Vertex s, Vertex end, int weight(▶ Weighted)) {
4          ...
5          ▶ GN_OnlyNeighbors
6          Neighbor e = new Neighbor(end, weight);
7          ...
8      }
9  }

```

Info: reference to 'weight' causes a feature artifact dependency: GN\_OnlyNeighbors => Weighted

Figure 4.4: Example of an artifact-related feature dependency

expensive in parser-based systems. Since projectional editors operate on an AST (which is directly modified by the user’s editing gestures), references between AST nodes (e.g., method call to method declaration) are actively maintained. PEOPL analyzes the AST for extracting feature constraints. In contrast to expensive static analyses required for parser-based systems [137], our analysis is quick (<1,8s on average for Berkeley DB). We discuss the concrete realization of the dependency checker in Section 9.1.

#### 4.2.2 Variant-based Data-Flow Analysis

Another challenge is to achieve SPL correctness. Figure 4.5 illustrates the issue. The variable `entryType` is not initialized if the feature *Transactions* has not been selected (cf. assignments in Lines 4, 9, 12 and the reference in Line 16). In fact, a variability-unaware data-flow analysis would not indicate an error, since the variable `entryType` is initialized in the 150% model (Lines 9 and 12). PEOPL mitigates this issue, since it enables developers to analyze the data flow of a target variant as well as all valid variants (each reflecting a different feature selection) of the SPL in a brute-force fashion.

Notice that analyzing the codebase for different feature selections in a brute-force fashion is a non-optimal solution. The computational cost is high, since the number of variants might be exponential (worst case). Thus, if the number of features is too large, the variant-based analysis might be too expensive or actually infeasible. To address this issue, previous work proposed, for instance, to increase the abstraction of the SPL to approximate the analysis—that is, trading in precision for speed, which makes the analysis of SPLs with a large number of variants feasible [57].

Another option is a so-called *family-based data-flow analysis* (a.k.a. *lifted data-flow analysis* or *feature-sensitive data-flow analysis*) in which data-flow graphs of variants are not generated, but a variational data-flow graph is analyzed (cf. [37, 39, 57]). This is helpful to find problems, such as unreachable code or uninitialized variables, in all variants before compiling the code (as illustrated in Fig. 4.5, Line 16). So creating variational data-flows (from the variational AST) is a valuable approach to be investigated for PEOPL in the future.

```

1  ▶ Base
2  public class LN {
3      private long log(..., Locker locker) throws DatabaseException {
4          LogEntryType entryType;
5          // 1) more code
6          Txn logTxn;
7          ▶ Transactions
8          if (locker != null && locker.isTransactional()) {
9              entryType = getTransactionalLogType();
10             // 2) more code
11         } else {
12             entryType = getLogType();
13             // 3) more code
14         }
15         // 4) more code
16         LNLogEntry logEntry = new LNLogEntry(entryType, this, ...);
17         // 5) more code
18     }
19 }

```

**Error:** Variable 'entryType' is not initialized if the feature 'Transactions' is not selected

Figure 4.5: Checking the data flow of a variant where the Berkeley DB feature *Transactions* is not selected

All in all, it might be valuable to exploit different analysis techniques for PEOPL in the future. For more information, we refer interested readers to a survey on available analysis strategies for software product lines [186].

On a final note, we discuss the concrete realization of the data-flow analyzer in Section 9.2.

#### 4.3 CONCLUDING REMARKS

Notice that, for now, PEOPL does not provide full-fledged variability-aware type checking support (cf. [107]). Thus, PEOPL currently cannot detect all problems in advance (before variant derivation). For instance, duplicated method signatures that occur due to the removal of method parameters cannot be detected. Instead, such errors are detected while transforming the SPL into a variant. Yet, detecting feature artifact dependencies and analyzing data flows of variants already helps resolving many issues before transformation. In summary, we plan to increase the validity of SPLs developed in PEOPL by leveraging techniques, such as family-based type checking [107] or model checking [44].

## EVALUATION I: THE PRODUCT LINE ENGINEER'S VIEW

---

To validate PEOPL from the SPL developer's perspective, we rely on three ingredients.

- 5.1: We use and extend existing classification frameworks [80, 100, 118] to compare PEOPL with other SPL implementation techniques. This way, we quantify PEOPL's practicality on a set of well defined quality criteria and investigate whether the approach is indeed desirable.
- 5.2: We implement a series of Java SPLs in PEOPL. These case studies enable us to understand the practicality of our approach and to investigate its scalability. Moreover, we approximate the number of boilerplates necessary in pure modular approaches to underline the usefulness of PEOPL, and its ability to switch between and blend annotative and modular representations.
- 5.3: We conduct two pilot user studies to learn more about PEOPL's usability. We identify possible usage scenarios and lay ground for future studies.

### 5.1 CLASSIFICATION I: COMPARING TECHNIQUES AND TOOLS

We have seen in Section 2.5 that different variability representations have distinct advantages [100, 103, 171]. We now complement this discussion by evaluating and comparing PEOPL with other concrete SPL engineering techniques and tools. To study and discuss different quality criteria, we use a classification framework which has been similarly proposed and used by others [14, 80, 100, 118, 171]. In contrast to previous work [100, 118], we quantify a broader set of techniques and tools. That is, we not only discuss CIDE (visual annotations), FH (feature modules), and a refactoring from CIDE into FH and vice versa on a set of quality criteria, but we also consider the CPP (textual annotative), DeltaJ (delta modules), and the VCS (reactive variant editing). Moreover, we expand the previously proposed set of quality criteria [14, 118]. In addition to *preplanning effort*, *adoption*, *uniformity*, *granularity*, *modularity*, *traceability*, and *information hiding* [14, 118], we also quantify the criteria *variability implementation flexibility*, *expressive power*, *homogeneous extension*, and *variant editing*. Table 5.1 shows our results, which we discuss in detail next.

	Variability Implementation Approach						
	CPP	CIDE	FH	DeltaJ	VCS	Refac- toring <sup>1</sup>	PEoPL
Preplanning effort ↗	low	low	high	mid	low	low	low
Adoption ↗	●●	●●	●○	●○	●○	●●	●●
Variability implementation flexibility ↗	○○	●○	○○	○○	○○	●○	●●
Uniformity ↗	✓	✓	✓	✓	✓	✗	✓
Granularity ↗	●●	●●	●○	●○	●●	●●	●●
Modularity ↗	○○	○○	●●	●●	○○	●○	●●
Feature traceability ↗	○○	●●	●●	●●	●○	●●	●●
Information hiding ↗	○○	○○	●○	●○	○○	●○	●○
Expressive power ↗	●●	●○	●○	●●	●●	●○	●●
Homogeneous extensions ↗	○○	○○	●●	○○	○○	○○	●○ <sup>2</sup>
Variant editing ↗	○○	●●	○○	○○	●●	●●	●●

●● very good, ●● good, ●○ medium, ●○ poor, ○○ no support

<sup>1</sup>Refactoring engine to transform CIDE into FH implementations and vice versa [103].

<sup>2</sup>The full strength of the underlying concepts are yet to be exploited.

Table 5.1: Classification of variability implementation techniques from the SPL developer’s perspective (extended/adapted [100, 118])

### 5.1.1 Preplanning Effort

Developers need to preplan an SPL’s realization irrespective of the concrete technique or tool used [14]. What are the features to be realized (a.k.a. *scoping decisions*) and how will they interact? Some approaches facilitate systematic reuse, while others induce a higher effort.

In approaches that support annotations such as the CPP, CIDE, and PEoPL the preplanning effort is low. That is, annotations can be added in an ad hoc manner at any time in the development life cycle [14, 100]. Thus, there is no upfront investment for separating commonalities and variabilities in the implementation. Approaches that support refactoring from and into annotations benefit from this advantage as well.

In VCS, the preplanning effort is also low. A developer just starts editing a certain product variant in an ad hoc manner. During a commit which requires a developer to assign features, all changes are merged into the underlying database—the SPL’s 150% model.

Modular approaches such as FH require a higher preplanning effort as developers need to find the right decomposition strategy. In particular, developers need to properly identify the SPL’s commonalities and variabilities encapsulating them in a base module and several



additional (optional) modules. If unidentified variabilities appear in later project phases, the base module must be restructured, and the variable code must be moved to a respective optional module in a cut & paste manner. Thus, pure FOP approaches make it rather expensive to fix unidentified variabilities in evolved structures.

DOP approaches mitigate this and other FOP issues [158]. Developers start the modular implementation from any product expressed in a core delta. If hitherto undiscovered variabilities are identified in the core delta, the developer can introduce a new delta, which (partially) removes the content of the initial product. This way, the core delta remains untouched and the program's evolution is better manageable. For instance, if a scoping decision is to be reverted the respective delta is simply removed (instead of refactoring the code again as in FH). Still, DOP approaches induce a medium preplanning effort as decomposition into deltas must be planned to a certain degree to obtain a well structured system.

### 5.1.2 Adoption

[jump to overview ↑](#)

*Adoption* mainly describes the (potential) interest of industrial software engineers and developers to use a certain variability representation in isolation [100]. Consequently, adoption is closely coupled with the (preplanning) effort to use a certain representation.

Practitioners have already widely adopted the CPP (●●) to implement variability (e.g., Linux) [90]. The main reason for the success of annotations is that they are easy to use, allowing developers to quickly produce results [45, 100]. Thus, the initial risk for companies is very low [45]. CIDE (●●) poses a similar line of work, and thus skills for visually annotating the source code can be quickly acquired. Although the CIDE prototype itself has not been adopted in practice, it has inspired tools such as mbeddr [193, 197], which is actually used by industrial developers [194].

Clone-and-own is also common industrial practice [3, 64, 75, 136]. To create a new variant, developers copy or branch the code to be reused and customized. However, this strategy is problematic, especially when changes need to be merged back into multiple variants. A VCS (●○) could improve this process, and thus the SPL's maintainability. Instead of manually applying changes to each variant, a developer can specify which variants are affected by a change [178]. Thus, although contemporary approaches are academic [178], a full-fledged VCS could have the potential to be adopted in practice. On the other hand, the acceptance of a reactive variant-editor is yet to show. One could argue that it is error-prone and intellectually challenging to assign a multitude of features after implementing all the functionality, and to decide additionally which variants should be changed.

Unfortunately, FOP (●○) and DOP (●○) approaches are rarely adopted in industrial practice despite their benefits [14, 100, 102]. The main issue is that the short-term risk for companies is high. Quick results cannot be expected as developers must learn a new technique from scratch. Moreover, developers must find the right decomposition strategy for implementing the SPL's modules, and potentially deal with boilerplates. Thus, the benefits of modular solutions remain unexploited in practice. Refactoring engines (●●) promise to reduce this adoption-barrier. Developers can start with visual annotations and transform them into modules and vice versa. Yet, there is currently no advanced refactoring engine allowing to refactor diverse representations—that is, only CIDE to FH and vice versa is available.

In contrast, PEOPL (●●) allows developers to start with a CPP-based projection and to use it with several other variability representations in parallel. Based on this flexibility, we argue that the potential interest of using PEOPL in industrial practice is the highest among its academic companions. Since projectional editing is substantially different to classical text editing, one could argue that developers need to learn new concepts, weakening adoption. However, recent work has shown that projectional editing efficiency is quickly achievable by short user trainings [31, 203]. In fact, a 45-minute training suffices to achieve editing skills comparable to classical text editing [31].

### 5.1.3 Variability Implementation Flexibility

[jump to overview ↑](#)

To quantify *variability implementation flexibility*, we investigate techniques and tools in the light of two key questions:

RQ1: *How fluently can developers switch between the available variability representations?*

A fluent movement enables using the best variability representation for a given feature artifact and task on demand.

RQ2: *Can developers observe and edit a feature artifact (e.g., class) using different variability representations in parallel (side-by-side)?*

Parallel editing enables an even faster movement between two techniques. Moreover, it helps observing the impact of changes, for instance to a variant.

The CPP, FH, DeltaJ and VCS do not support switching variability representations out of the box (○○).

CIDE (●○) only supports visual annotations (background colors) and a simple variant editor, which hides feature artifacts. True modularity is not supported. To edit a variant, developers unselect features. Yet, moving between the annotated codebase and different configurations is cumbersome, since for each configuration the correct features

need to be selected manually and from scratch—that is, configurations cannot be predefined (RQ<sub>1</sub>). Moreover, developers cannot see and edit the full annotated codebase and a specific variant side-by-side in parallel (RQ<sub>2</sub>).

Figure 5.1a illustrates the editing possibilities in a refactoring environment (●○). First, refactoring is time-consuming as a variability representation must be transformed into another representation. Thus, moving efficiently between different variability representations is not possible (RQ<sub>1</sub>). Second, refactoring does not support parallel editing properly, since only one particular snapshot of the SPL can be edited consistently at a time. In fact, parallel editing of different refactored versions causes inconsistencies, and demands developers to resolve conflicts (RQ<sub>2</sub>).

Figure 5.1b illustrates the projectional environment of PEOPL (●●), which seems to be the most flexible approach thus far. PEOPL allows developers to move fluently between its different variability representations for a given feature artifact (RQ<sub>1</sub>). Moreover, developers can edit the code using different techniques side-by-side in parallel (RQ<sub>2</sub>). In fact, any editing activity directly manipulates the internal representation of the SPL, while the variational concrete syntax (e.g., annotated program) is just a rendering. This way, inconsistencies are avoided by design.

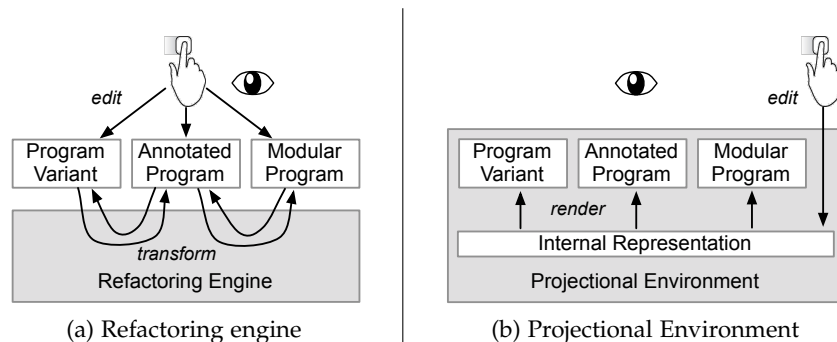


Figure 5.1: Editing activities compared: refactoring vs. rendering

#### 5.1.4 Uniformity

[jump to overview ↑](#)

*Uniformity* describes a consistent, homogeneous variability encoding, regardless of the feature artifact type (e.g., text or diagrams) [14, 22]. If different (internal) representations are intertwined, variability is persisted in an inconsistent manner. Then, developers need to integrate and coordinate different tools, and thus tracing a feature to all its artifacts becomes more difficult.

The CPP, CIDE, FH, DeltaJ, and VCS have their own variability representation, and thus persist variability in a uniform manner (✓). Notice that PEOPL (✓) allows developers to use diverse external vari-

ability representations, but is build upon a common, internal variability representation. Thus, uniformity is achieved as well. In contrast, using partial refactoring (✗), variability is persisted inconsistently. For instance, CIDE annotations can appear within FH modules, which requires developers to integrate and coordinate the two tools.

### 5.1.5 Granularity

[jump to overview ↑](#)

Feature artifacts implement features and their functionality. Recap that *granularity* describes the structural level a feature artifact possesses in a program [14] (cf. Sec. 2.4.2.1). The CPP (●●) and VCS (●●) are text-based. As such, they support the addition of very fine-grained annotations to arbitrary positions in the program’s concrete syntax. Yet, a drawback is that developers can accidentally invalidate the program’s syntactic correctness. For instance, annotating a statement while missing the semicolon.

To enforce only syntactically correct annotations [105], CIDE (●●) and PEOPL (●●) allow an addition of annotations only to AST nodes. Still, both approaches support fine-grained variability. In fact, they allow developers to realize the most important undisciplined annotations, for instance adding an annotation to a method parameter, or a wrapper [125]. As a result, they are powerful enough to implement real-world SPLs such as Oracle’s Berkeley DB [27, 102].

Modular approaches such as FH (●○) and DeltaJ (●○) are known to support variability only on a medium to coarse-grained level. In fact, fine-grained extensions require workarounds such as hook methods, code clones, or additional fields [102, 162]. Refactoring engines (●●) improve granularity of modular approaches, since they allow an integration of annotations into modules.

### 5.1.6 Modularity

[jump to overview ↑](#)

*Modularity* enables developers to clearly separate an SPL’s concerns. It is “the ability to identify, encapsulate, and manipulate only those parts of a software that are relevant to a particular concept, goal, or purpose” [141]. Optimally, modular approaches even enable the separate compilation of individual modules [43]. This way, content-sensitive modules can be compiled before handed over to customers.

Typically, modularization is not supported by approaches that have a 150% model at their core. In fact, the CPP, CIDE and VCS do not support true modular reasoning (○○). FH (●●) and DeltaJ (●●) are better in that concern, since features are mapped to individual modules or deltas, respectively. Yet, neither FH nor DeltaJ supports separate compilation of modules.

Using a refactoring engine (●○), developers can start with annotations and refactor them into separate modules. Yet, refactoring may

introduce unforeseen hook methods in modules refactored from fine-grained annotations [103]. Alternatively, partial refactoring is possible, which would leave some annotations unmodularized. Consequently, either way weakens modularity.

Compared to refactoring, PEOPL (●●) offers a more powerful and flexible solution. Fine-grained modular reasoning is available in annotated and fade-in feature modules. Exploring a feature in isolation is possible as PEOPL employs a modularity-aware 150% model, which treats feature modules as first-class citizens (explained shortly in Ch. 7). However, the drawback of employing a 150% model is that separate compilation is not available.

### 5.1.7 Feature Traceability

[jump to overview ↑](#)

*Feature traceability* describes the ability to trace a feature from its declaration to its feature artifacts [4, 14, 50, 100]. This traceability is important, for instance, when exploring a feature's implementation, or when searching for bugs that are known to be introduced with a certain feature [100].

Using the CPP (○○), each feature's implementation is scattered across multiple annotations. Thus, without additional tooling, the CPP does not support feature traceability [14]. The VCS (●○) supports to check out a variant with only the relevant features selected. However, it is likely that selected feature code is nested within unselected feature code. In this scenario feature artifacts are missing, and thus the trace results are incomplete. CIDE (●●) performs better, although building upon a 150% model. In fact, CIDE allows hiding all irrelevant features by unselecting them. If the code of a selected feature is nested within the code of an unselected feature, then the code of the unselected features is shown. Thus, it is likely that the selected feature's code is cluttered by irrelevant feature code, which in turn weakens the traceability.

The traceability support of modular approaches is better, since they separate features into distinct modules [4, 14]. In FH (●●), a feature can be easily traced to a corresponding module. In DeltaJ (●●), mappings from features to deltas can be expressed in an SPL declaration explicitly. Partial refactoring (●●) weakens traceability [100]. For instance, if a feature module contains annotations, then the underlying variational models are intertwined. In fact, we would need to additionally hide all code that is irrelevant for the selected feature within a module (e.g., using CIDE).

In contrast to partial refactoring, PEOPL (●●) uses a single internal variability representation, and thus even fine-grained feature artifacts can still be related to their respective feature module. All in all, PEOPL is on a par with other modular approaches, and thus supports traceability well.

### 5.1.8 Information Hiding

[jump to overview ↑](#)

The key idea of *information hiding* is to divide a module into an internal part and an external part [14]. The internal part is only visible within the module, whereas the external part is publicly available via an interface—a contract, which specifies how the module communicates with all other modules [14]. On this basis, a developer can implement and understand a module by reasoning about the module’s internal part as well as all imported external parts of other modules [14]. Note that information hiding is strongly linked to modularity, and thus non-modular variability representations, such as the CPP, CIDE and VCS, do not support information hiding at all (○○)

Without explicit external parts (i.e., interfaces), feature internals cannot be hidden. Moreover, a developer needs to find all code accessible in other modules manually—a tedious and error-prone endeavour. For instance, to find all methods that are available in a given module, a developer must browse through all external modules, identifying the public methods declared and accessible in all configurations. Unfortunately, FH (●○), DeltaJ (●○) and other modular approaches handle information hiding this way without additional tooling [14]. To deal with this issue in an automated fashion, *feature context interfaces* recently emerged, showing an outline of accessible fields and methods for a given module and enriching code completion [160].

PEoPL (●○) uses a similar but more flexible approach. In fact, it does not require developers to search for all the accessible declarations. Instead all available declarations can be temporarily included in the current feature module (by projecting them in). Moreover, code completion shows accessible declarations as well. Yet, unsound results such as declarations, which are not available in the configuration set, are not filtered out currently. In addition, interfaces are not explicit, and thus we count PEoPL’s information-hiding support only as mediocre.

### 5.1.9 Expressive Power

[jump to overview ↑](#)

*Expressive power* (a.k.a. *expressiveness*) describes the ability of a technique to modify the SPL’s implementation (i.e, which kinds of changes are applicable by a feature artifact).

The CPP (●●) is clearly the most expressive approach. It allows developers to conditionally include a feature artifact, using a presence condition, which declares a possibly complex expression over properties (typically features). Even equations, inequations, and arithmetic operations are possible. Thus, feature artifacts can be included or removed if a feature is not selected or a certain threshold is met. In contrast, the current VCS (●●) version supports only Boolean pres-

ence conditions over features [178]. Thus, it is slightly less expressive than the powerful CPP.

DeltaJ (●●) and other delta-oriented approaches are highly expressive. They allow the addition, modification and removal of feature artifacts through deltas. In fact, using DeltaJ's SPL declaration, developers can control the application of a delta by means of a propositional formula over features. More recently, *Parametric DeltaJ* (●●) emerged, which enables developers to propagate feature attributes to deltas, and to use equations, inequations, and arithmetic operations within feature constraints [207]. As a result, Parametric DeltaJ is even more expressive.

PEoPL (●●) provides the same expressiveness as standard DeltaJ [25], since feature modules enable the addition, modification and removal of feature artifacts. Moreover, PEoPL's SPL declaration allows developers to conditionally apply a set of modules using a propositional formula over features.

In contrast, CIDE (●○) represents annotations as simple presence conditions over features. Thus, if a feature is selected, the corresponding code is included, otherwise not. As a result, CIDE is less expressive. For example, it is not supported to include a feature artifact if the corresponding feature is not selected. One possible workaround on the statement-level is to exclude the corresponding code at runtime, using the target language. For instance in Java, a developer colors only an `if(false)` statement, but not its body. If the feature is not selected, the wrapper is removed, otherwise the `if`-statement prohibits execution at runtime. Yet, there is no workaround above statement-level, and uniformity is weakened this way, since variability is encoded using CIDE and the target language (e.g., Java).

Similar problems arise in FH (●○), which only allows the addition, refinement, and replacement of feature artifacts. In fact, removal of feature artifacts is not supported. As CIDE and FH only provide a mediocre expressiveness, a refactoring (●○) between the two cannot perform better.

#### 5.1.10 Homogeneous Extensions

[jump to overview](#) ↑

Homogeneous and heterogeneous extensions are the two fundamental ingredients of a feature's implementation (regardless of the variability representation employed) [9, 49]. Recap that a heterogeneous extension adds different feature artifacts to different program locations, while a homogeneous extension adds the same feature artifact to multiple program locations (cf. Sec. 3.3.7).

Variability mechanisms support homogeneous extension very differently [7, 130]. For instance, aspect languages, such as *AspectJ*<sup>1</sup> (●●), provide full-fledged support, since aspects have advice applied to po-

<sup>1</sup> <http://www.eclipse.org/aspectj/>

tentially multiple join points. Similarly, FH (●●) provides support for homogeneous extensions on its underlying structural level—that is, the program’s FST, which is a simplified stripped down AST [7]. In fact, via a query, FH supports the specification of positions in the program’s FST to which a set of feature artifacts are to be applied. However, extensions in the control-flow such as in AspectJ are not possible.

PEoPL (●○) supports homogeneous extensions in general (cf. reuse projection in Sec. 3.3.7). In fact, a feature artifact’s multiple appearance can be persisted in the internal representation (explained shortly in Ch. 7). Yet, there is currently no support for specifying a set of join points in a projection, for instance, via a query. Instead, developers add and maintain homogeneous extensions manually, using a *pick & appear* process (cf. Sec. 3.3.7). In fact, a developer simply selects a piece of code (similar to copy) and then selects positions across the program where the code should appear (paste). Notice that internally the variational AST is enriched with cross-tree references for each appearance pointing to the original piece of code (i.e., the feature artifact). This way, in contrast to FH, even fine-grained homogeneous extensions are possible, and changes made in one appearance are automatically visible in all other appearances. Adding an advanced query mechanism is subject to future work.

All other approaches in our classification (○○) do not properly support homogeneous extensions. In fact, developers use workarounds, such as copy & paste, which leads to code clones, making bugs difficult to detect [124] and fix [96]. As a possible workaround, developers can implement homogeneous extensions using an AOP approach on top of the variability representation (e.g., CPP) which breaks uniformity.

On a final note, a study of forty C-based product lines has shown that up to 10% of the extensions are homogeneous [126]. Interestingly, a study of eleven AspectJ programs indicates that developers seem not to leverage the expressive power of AspectJ as only about 2% of the extensions are homogeneous [5]. Consequently, support for homogeneous extensions seems not to be a crucial feature.

#### 5.1.11 Variant Editing

[jump to overview ↑](#)

Most modular approaches such as FH (○○) and DeltaJ (○○) do not support editing concrete variants. An exception is AHEAD (●○), which is not classified here. It provides the *jampack* composer [22], which composes Jak language files—that is Java files enriched with variability—and transforms the composite file into Java code. Changes made in the Java code can be merged back into the composite Jak file. However, it is unclear how to propagate the changes back into the source Jak files.



In the VCS (●●), developers can pull a certain variant from a common database. After implementing the desired functionality, the corresponding changes can be commit into the database. Refactoring (●○) a FH SPL into CIDE enables variant editing, since CIDE (●○) emulates a variant editor by hiding feature-related code. Yet, it is not possible in CIDE to switch fluently between variants as detailed in Section 5.1.3. Moreover, it is not clear whether the selected variant is valid or not (i.e., conforms to the feature model). This issue arises as CIDE’s hiding facilities are not connected to CIDE’s satisfiability solver, which ensures validity of a certain configuration. This lack however constitutes a major issue as *‘many developers fail to exactly identify the set of erroneous configurations, already for a low degree of variability’* [135].

In contrast, PEOPL (●●) provides full-fledged proactive variant editing support, which is based on a valid feature configuration—that is, PEOPL checks the propositional formula of a given configuration. In fact, valid configurations can be preconfigured, and thus it is possible to fluently switch between different variants on demand.

## 5.2 CASE STUDIES

We now evaluate PEOPL with three objectives on a set of case studies.

### 5.2.1 Objectives

- O1: *Analyze practicality*: We show that PEOPL can realize SPLs by writing them from scratch or migrating from common annotative or modular variability representations.
- O2: *Analyze scalability*: We investigate latencies for creating variant projections of a specific file and for deriving full variants, together with qualitatively assessing the editing efficiency.
- O3: *Assess the benefit of multiple projections*: We study this benefit by analyzing the overhead of a classical pure modular approach by approximating the boilerplate code it would require to write.

### 5.2.2 Case Study Subjects

Table 5.2 shows our SPLs. We migrate seven SPLs used in previous research [7, 101, 102, 115, 128, 163], and implement one (Jest) from scratch. All cover different domains and scales. Most migrations are CIDE projects, for two reasons. First, it is easy to migrate annotative SPLs to PEOPL. We import the codebase and manually re-implement annotations. Second, we aim at using annotative SPLs to evaluate the potential overhead in a pure modular approach. We also migrate three projects from DeltaJ and FeatureHouse respectively. They both

use modular representations. We import each module as a Java package into PEOPL and use our modular projection for migrating the code. The adoption effort for all subjects is moderate. Creating the subjects takes seven days for Berkeley DB, three days for Jest, and just a few hours each for the others (including comprehending the SPLs).

### 5.2.3 *Investigation*

We now investigate our three objectives by discussing metrics, methodologies and results.

#### 5.2.3.1 *Practicality (O1)*

Although time-consuming and error-prone, and although an analytical approach could have sufficed to evaluate PEOPL, the manual adoption helps us understand the usability of our projections. No subject requires specific workarounds. We conclude that PEOPL indeed enables developers to realize annotative and modular SPLs. To reduce adoption effort, we plan to write custom importers.

#### 5.2.3.2 *Scalability & Latencies (O2)*

**METRICS.** We use the following three metrics (all in milliseconds) to evaluate scalability. TCF (time to compose file) is the time to compose the variant's set of feature artifacts of a single file. TCV (time to compose variant) is the time to compose the variant set of all files (i.e., the complete set of feature artifacts of a variant). TGV (time to generate variant) includes TCF plus the time to write all Java classes of the variant to disk. We use TGV to compare PEOPL to composition times of other SPL tools. Finally, we also measure the time it takes to rebuild the feature artifact dependencies across all feature modules for the entire SPL (TMD) using our dependency checker.

**METHODOLOGY.** TCV measures the editing latencies of variant projections, since we compose a full product to update the variant editor and explorer (i.e., the tree view on a product's files). PEOPL caches the current variant's set of feature artifacts until variability-related operations (e.g., adding a new feature artifact or changing a module assignment) invalidate the cache. So we turn off caching to avoid confounding. To measure TCV, we compose all modules included in the current configuration. TCV is most important, as it excludes the confounding model-to-text transformation introduced with TGV. We compare TCV to TCF to determine whether the reduced set of feature artifacts of TCF improves composition performance and yields a better efficiency. A drawback of TCF may be that we need to populate all feature artifacts in a file that belong to a module before composition, since the reduced set is not persisted.

SPL	Size & Complexity						Scalability & Latency				Source Tool	Description
	LOC	CLA	MET	F	FM	FA	$\overline{TGV}$	$\overline{TCV}$	$\overline{TCF}$	$\overline{TMD}$		
Jest	19k	144	1105	22	22	205	2535ms	9ms	<1ms	240ms	From scratch	Java Elasticsearch client*
Berkeley DB	70k	218	3433	42	83	1373	5153ms	45ms	<1ms	1753ms	CIDE	Embedded database <sup>¶</sup> [102]
GPL	1k	15	125	21	26	105	248ms	2ms	<1ms	54ms	CIDE	Graph product line <sup>¶</sup> [128]
Java-Chat	0,6k	8	58	9	9	33	260ms	<1ms	<1ms	22ms	CIDE	Chat client <sup>¶</sup>
Lampiro	45k	140	1693	19	19	181	4234ms	7ms	<1ms	1883ms	CIDE	Instant-messaging client <sup>†¶</sup>
Prop4J	2k	6	174	14	14	192	249ms	1ms	<1ms	98ms	FeatureHouse	Propositional formula library <sup>§</sup>
Vistex	2k	9	99	16	16	37	287ms	<1ms	<1ms	62ms	FeatureHouse	Graph visualization and text editor <sup>§</sup>
STE	1k	9	128	10	10	38	259ms	<1ms	<1ms	50ms	DeltaJ	Simple text editor <sup>‡</sup> [79, 115]

LOC: lines of code (source) | CLA: classes | MET: method declarations | F : features | FM : feature modules  
 FA: feature artifacts (sanitized) |  $\overline{TGV}$ : time to generate/derive a variant |  $\overline{TCV}$ : time to compose a full variant  
 $\overline{TCF}$ : time to compose a file variant |  $\overline{TMD}$ : time to rebuild feature artifact dependencies

\*<https://github.com/searchbox-io/Jest> †<http://lampiro.blundo.com/>

‡<https://www.tu-braunschweig.de/isf/research/deltas/#fulljava> §<http://spl2go.cs.ovgu.de/> ¶<http://ckaestne.github.io/CIDE/>

Table 5.2: Case study subjects: Java-based product lines realized in PEoPL

We conduct all measurements on a standard 2011 iMac (3,1GHz Intel i5, 16GB, Radeon HD 6970M, OS X 10.10.5, MPS 3.3.6, Java 1.8) with randomly generated distinct variants that conform to the feature artifact dependencies. We then compare the composed variant to all variants previously generated. If sets of feature artifacts are equal, we skip the variant, otherwise we save it for future comparison.

**RESULTS.** Table 5.2 shows all results and Figure 5.2 the distribution of TCV values for Berkeley DB (for the others, the values are too low to be meaningful). Generating and writing 2000 Berkeley DB variants to disk is below 5.2 sec. on average ( $\overline{TGV}$ ). Using equivalent product configurations, we compose and write the same Berkeley DB variant to disk using PEOPL (around 6 sec.), FeatureHouse (around 18 sec.), and CIDE (around 7 sec.). Composing a full variant is below 45 ms on average ( $\overline{TCV}$ ) and just a single document below 1 ms ( $\overline{TCF}$ ). Calculating the feature artifact dependencies is quick as well (for Berkeley DB <1,8 sec. on average,  $\overline{TMD}$  in Table 5.2).

In summary, the PEOPL prototype scales well to SPLs of Berkeley DB size. Latencies to calculate the feature artifacts for a variant projection are efficient according to the TCF and TCV measures. PEOPL does not introduce any significant overhead, and thus developers can experience a smooth editing.

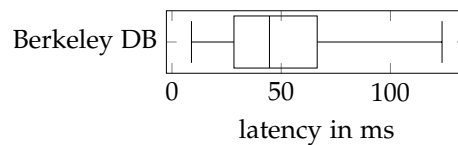


Figure 5.2: Calculation times for a full variant (TCV)

### 5.2.3.3 Boilerplates in Modules ( $O_3$ )

**METRICS.** The basic idea of  $O_3$  is that a developer uses feature modules for their advantages. Yet, classical modular approaches require boilerplate code for fine-grained feature artifacts, such as hook methods (cf. Sec. 2.5.1). We aim to show the need for annotations, providing indirect evidence that PEOPL—which allows blending annotations into modules and switching to annotations on demand—is useful. To approximate the potential interest of these two PEOPL facilities, we measure the number of methods that would require boilerplate code in a pure modular projection in correlation to the involved modules. This shows whether variability in a method’s body introduced by different modules impacts the number of boilerplates.

**METHODOLOGY.** To determine whether a method requires boilerplate code, we search for variability in the middle of a method, and fine-grained variability (e.g., annotated method-call parameters). A method requires boilerplates if such variability is found. Notice that

we allow variational method parameters and return-types in our modular projection. Thus, we do not mark such methods as demanding boilerplates—although they would require boilerplates in some classical modular approaches.

**RESULTS.** It is not surprising that all the methods adopted from the modular FeatureHouse and DeltaJ examples do not require additional boilerplates in a pure modular projection. Thus, we concentrate on the CIDE examples. With total numbers of 13% (Berkeley DB), 1% (Lampiro), 6% (GPL), and 15% (Java-Chat), all tested annotative SPLs contain a relatively small number of methods requiring boilerplates. However, as soon as investigating variability in the method’s body, the need for blended projections and fluent movement between projections becomes obvious.

Table 5.3 shows our boilerplate test results. The majority of methods not requiring boilerplates in pure modular approaches are simple introductions (i.e., only one module is involved). The picture changes as soon as a method gets refined (i.e., at least two modules are involved). Especially, the Berkeley DB methods require a large number of boilerplates.

FM	Berkeley DB		Lampiro		GPL		Java-Chat	
	MET	BOIL	MET	BOIL	MET	BOIL	MET	BOIL
1	2813	0%	1567	0%	77	0%	40	0%
2	398	78%	21	86%	12	25%	6	83%
3	82	86%	8	88%	5	40%	5	60%
4	35	91%	2	100%	3	0%	1	0%
5	21	95%	1	100%	–	–	–	–
6	5	100%	1	100%	1	100%	–	–
7	3	100%	–	–	–	–	–	–
8	1	100%	–	–	2	100%	–	–

FM: feature modules involved in method body

MET: method declarations

BOIL: method declarations that would require boilerplate code

Table 5.3: Method declarations requiring boilerplates in pure modular approaches

#### 5.2.4 Threats to Validity

**INTERNAL VALIDITY.** To mitigate the threat that our SPLs are incorrectly implemented, we cross-checked their implementations and carefully specified and reviewed our variant transformation rules. Moreover, in the final Java code-generation, MPS—the base platform

we used to implement PEOPL—would have detected invalid ASTs. Furthermore, PEOPL relies heavily on cross-tree references in the internal representation. We carefully designed our internal representation to maintain these references throughout the AST editing, as broken references can invalidate a program’s variability.

To enhance the validity of our scalability evaluation ( $O_2$ ), we randomly created variants and implemented a checking rule to detect duplicated ones. We double-checked that tested configurations are not biased (i.e., too few or many feature modules over all configurations).

Finally, for the modularity overhead ( $O_3$ ), we inspected a sample of files to verify that the boilerplates are actually necessary.

**EXTERNAL VALIDITY.** To increase external validity, seven of our eight subjects are publicly available SPLs of different size and complexity previously used as SPL benchmarks. Among them, there is Berkeley DB, a substantial embedded database that has been decomposed using different techniques before [7, 101, 102]. Furthermore, although we extensively tested PEOPL only with Java, it is a mainstream language. Java not only benefits from PEOPL’s projections, but also from variability support in general. Still, evaluating PEOPL in the context of other languages—especially C—and larger SPLs is valuable future work.

### 5.2.5 *Lessons Learned*

We found switching representations and using them in parallel useful when adopting our subject SPLs. For instance, we switched the modular to the annotative (or blended) projection when contextual information was required for comprehension. We also switched for implementing fine-grained variability or exploring feature interactions. Using the annotative projection, we found a behavior-related issue in the STE SPL that was neither easy to identify in the original DeltaJ implementation, nor in our modular projection [25]. We typically used either the modular and product projection, when annotative code was too complex, or we searched for bugs known to occur in a feature. We also leveraged the locality of the modular projection to identify how a single feature was implemented.

We learned that using our dependency checker and variant-based data-flow analysis is straightforward and valuable. We actually found 57 implementation-specific dependencies which were not declared in the feature model of the CIDE version of Berkeley DB. So incorrect variants could have been generated.

### 5.3 PILOT USER STUDIES

We now discuss our two PEOPL pilot user studies. Our aim is to validate the study setup and prepare a larger (longitudinal) study and a controlled experiment. We start with a discussion of our objectives defining two variability-related research questions and one research question related to projectional editing in general (Sec. 5.3.1). Then, we introduce the participants and the general study format (Sec. 5.3.2). Moreover, we discuss the material and tasks for setting up the experiments (Sec. 5.3.3), and our methodology and analysis (Sec. 5.3.4). We conclude by discussing and interpreting the results providing an outlook for future studies (Sec. 5.3.5 and 5.3.7). Note that all materials and results are available online<sup>2</sup>.

#### 5.3.1 Objectives

The goal of PEOPL is to increase the flexibility of programmers when working on an SPL's implementation. For instance, facing limitations of a variability representation, developers can switch projections on demand. We are interested in how developers use PEOPL's projections when adding variability to real-world systems, implementing new features from scratch, and extending existing features.

We investigate PEOPL-specific usage scenarios and the behavior of developers using the following research questions:

**RQ1:** *Which variability projections do developers use, how frequently and in which scenarios?*

We use this question to determine the preferred variability representation of developers (i.e., the ratio of using PEOPL's annotative, modular, and variant-specific projections). We are also interested in analyzing the decision making of developers (i.e., when to choose which projection) to identify pitfalls and give more solid future guidelines for projection usage and the overall development process.

**RQ2:** *Do developers frequently switch projections and edit the same code, using different variability representations side-by-side?*

In our case studies, we regularly switched projections and edited the same code, using different representations side-by-side. For instance, we use full annotative and variant projections side-by-side to determine the impact of editing activities on variants in real-time. We want to determine whether developers use similar strategies to be able to give thorough usage advice.

We are also interested in the general usage of PEOPL and the underlying MPS language workbench to answer the question:

<sup>2</sup> <http://peopl.de/pilot-user-studies/>

RQ3: *Which IDE elements do developers use and which shortcuts?*

We want to determine how much time developers spend on non-code editing activities. For instance, using the project explorer or finding the usage of a specific node within the project. This way, we can provide recommendations on how to use PEOPL in general, and identify possible pitfalls in the overall development process.

### 5.3.2 *Participants and Study Format*

We conducted two pilot studies. The first study was part of the *Software Product Line Engineering* course open for students of the Chalmers University of Technology and the University of Gothenburg from November 2016 to January 2017. The second study was part of the *Software Architecture* course at htw saar from February to March 2017. In the following, we refer to the first study with *Chalmers* and to the second with *htw*.

The subjects of the Chalmers group were four computer science master students, the subjects of the htw group three master students in an applied computer science study program. The Chalmers group had one, the htw group two lectures on *Variability Implementation Techniques and Concepts* in which classical annotative, modular and variant-based variability representations were taught on a comparable level of detail. All students were able to choose between different projects, of which one was an SPL implementation project using PEOPL. To solicit participation, we briefly presented the PEOPL IDE during a lecture, and taught basic skills in a small scale development task during 12 hours of hands-on lab sessions. In doing so, students had to implement an SPL from scratch, first using solely annotative and then modular projections. Students used neither PEOPL nor the language workbench MPS before.

All students who chose the PEOPL project were aware of their participation in a pilot user study. There was no compensation for participation. With regard to grading, students were informed that the plausibility of their SPL scoping decisions, the quality of their developed SPL, and the quality of a mandatory experience report were used for grading. Thus, neither their efficiency in using PEOPL nor any usage-related data was taken into account.

Students were given a project description, which included the key learning goals of the project, an example description of a subject system, our expectations for the experience report, and the evaluation criteria for their grading. Students in the Chalmers group had six weeks, students in the htw group eight weeks to meet all the requirements for course completion. All students were asked for participation in our anonymous user interaction tracking program, which automatically captures and transmits data to us. After successful project



completion, we asked them to complete a questionnaire. Both, the monitoring and the post-experiment questionnaire were voluntary.

### 5.3.3 *Material and Tasks*

We now present the subject systems used by each group. Moreover, we discuss the ingredients we used for evaluation: experience report, questionnaire, and IDE interaction monitoring.

#### 5.3.3.1 *Subject Systems*

The domain was not fixed, and thus students could propose a subject system from a domain they were familiar with. Yet, the chosen subject system had to be a feature-rich Java project (importable into PEOPL). In our project proposal, we outlined an example scenario using Jest<sup>3</sup>, a Java client for Elasticsearch available on github with 832 commits from 48 contributors. While Jest is relatively small compared to other Java enterprise software (19KLOC in 144 classes), its size is manageable within a six weeks project. In fact, we implemented a PEOPL reference SPL implementation of Jest, which we used to check whether time constraints can be fulfilled by students (cf. Sec. 5.2.2).

The Chalmers group chose the Jest Elasticsearch project from our proposal. The htw group chose a different project. They implemented the *Simple Java Mail*<sup>4</sup> client, which is available on github and has 456 commits from four contributors. It is smaller than Jest (5400 LOC in 68 classes), but also feature-rich.

#### 5.3.3.2 *Subject System Setup*

Both groups were required to analyze the domain of their chosen subject system and make scoping decisions. Based on their decisions they had to specify a feature model and appropriate domain-specific constraints. Then, each student had to progressively grow the SPL in PEOPL by implementing the identified features using the projections provided. As a bonus for grading, we proposed to implement new features from scratch, and to extend existing features. For instance, we proposed the Chalmers group to integrate a GUI into Jest for three fictional customers (e.g., user, assistant, administrator), while the htw group integrated support for emojis. In fact, our intention was to see whether they would use modular techniques for implementing features from scratch—since transitioning a single software product into an SPL is easier with annotative approaches, and thus biased.

<sup>3</sup> <https://github.com/searchbox-io/Jest>

<sup>4</sup> <https://github.com/bbottema/simple-java-mail>

#### 5.3.3.3 *Experience Report*

An integral part of the students' grading was a mandatory project report in which they had to elaborate their experiences of implementing the SPL and using PEOPL. We advised them to keep a daily diary to systematically gather data for their report. Moreover, we informed them that there should be a focus on their experience using PEOPL. For instance, we proposed to discuss how they used the different projections to adopt the identified variability, how they reasoned about the product line, and how they implemented new features. Moreover, we asked them to report whether and when they switched projections, when they edited the code using different projections side-by-side, and how they used the tool in general.

#### 5.3.3.4 *Questionnaire*

We use a post-experiment questionnaire to capture the students' perceptions. Using a five-point Likert scale—for instance, ranging from very poor to excellent—we ask closed questions such as “*How do you estimate the impact of visual annotations on the understandability, maintainability and evolvability of the implementation*”. We also ask them for a self-assessment in questions such as: “*Which projections did you use in PEOPL and how regularly?*” We use this quantitative data for comparison with our user interaction monitoring (explained shortly). In open questions, such as “*In which scenarios do you prefer to use the visual annotative projection (i.e., code enhanced with colored bars)?*” and “*Was there anything you wish you could have done with a particular projection, but could not?*”, we gather qualitative information about experiences.

Moreover, to guarantee sufficient programming experience of the students, we measure their GPL and SPL programming experience based on a previous proposal of Siegmund et al. [70]. For instance, we ask questions about their experience with different programming paradigms and industry projects.

#### 5.3.3.5 *IDE Interaction Monitoring*

To transparently and anonymously capture usage statistics in MPS, we implemented an interaction monitoring mechanism, which was inspired by the Eclipse *mylyn*<sup>5</sup> plugin [109]. The interaction monitoring allows us to augment our qualitative measures from the experience report and questionnaire with a larger set of significant data. Based on the captured data, we can recall all past contexts and quantitatively analyze the usage of PEOPL. In particular, our monitor captures keyboard and mouse events to determine active editors (projections), window and editor changes, selections, periods of inactivity, key bindings, as well as editing and navigation behavior. All interaction events

---

<sup>5</sup> <http://www.eclipse.org/mylyn/>

are stored in memory until persisted in an interaction history—that is, an XML file. We benchmarked that the interaction history for a single person day amounts to up to 300KB.

#### 5.3.4 *Method and Analysis*

For all research questions, we triangulate three sources of data: experience report, questionnaire, and IDE interactions. First, we inspect the experience report. We are particularly interested in the strategies used by developers to cope with variability and their overall experience in using PEOPL. Second, we analyze the questionnaire’s closed questions (Likert scale) using box plots, and inspect the open questions.

To analyze the interaction monitoring usage data, we implement a simple analysis tool that extracts relevant interaction information and creates usage statistics. All interaction monitoring data is automatically captured and uploaded to a Dropbox account, along with message and event logs (e.g., info, warning and error information). The entire process is anonymous (i.e., each user creates a unique anonymous identifier at the beginning of the experiment). Transmission does not require user interaction and is performed automatically.

With regard to RQ<sub>1</sub>, we determine the *ratio of variability representation usage*—the number of code editing in a projection over the number of code navigations. In other words, we capture the amount of keystrokes in a projection and compare it to the number of mouse activities (i.e., moving the mouse, scrolling, and selecting) in the projection. A higher ratio indicates a higher editing activity in the corresponding projection, while a lower ratio indicates that the projection is rather used for exploring the code. Using this data, we can generate usage statistics for each projection and compare them.

For RQ<sub>2</sub>, we chronologically analyze the history in regard to active editor windows that render the same file. To assess RQ<sub>3</sub>, we analyze the activity of mouse and cursor interactions in the entire IDE. This way, we can determine, which windows are used by developers and how regularly.

#### 5.3.5 *Results*

We now discuss the results of our pilot study. Before we proceed with details, note that our sample is too small, and the majority of students were too inexperienced in programming in general and in the different variability representations involved in the study. We also exclude the user interaction, monitoring data gathered from the Chalmers group, since we just used it as a trial run for improving the interaction monitoring and fixing bugs. Unfortunately, a student of the htw group identified issues with the interaction monitoring on Windows

when his computer switched to stand-by (i.e., *NullPointerException*), which reduces the validity of approximately two thirds of the data of the htw group. Consequently, we cannot control all answers from the questionnaire, responses using a second valid source of data. We thus focus on the experience report and the questionnaire. Moreover, two questionnaire responses from the htw group are contradictory (explained shortly), and thus we exclude them from our evaluation. Altogether, our results are not meaningful enough and only reflect tendencies. In fact, the results are only suggestions laying ground for future studies.

Nevertheless, we can confirm that PEOPL is indeed usable, since both groups successfully realized their SPLs and gave us positive feedback on PEOPL's usability. Moreover, we made several interesting observations that we discuss next.

#### 5.3.5.1 Which variability projections do developers use, how frequently and in which scenarios? (RQ1)

All students indicated in the responses to the questionnaire that they used the visual annotative projection by default, which the interaction data tendencies confirm. Yet, due to the “sleep bug” in the user interaction monitoring, the edit samples for calculating the edit ratio are too small to be meaningful. Figure 5.3 provides an overview on the variability-representation usage of participants based on questionnaire responses. Note that participants typically avoided the textual annotative projection, while the modular and variant projections have a higher usage frequency.

The questionnaire responses and experience reports also provide qualitative insights. All participants rated the comprehensibility, maintainability and evolvability of programs rather high when using

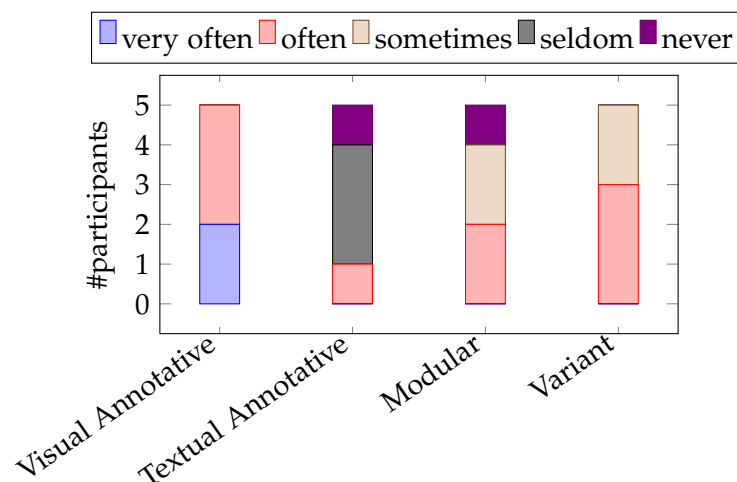


Figure 5.3: Results for the question: “Which projections did you use in PEOPL and how frequently?”

visual annotations. Textual annotations in turn are rather poorly rated. Modular and variant projections are generally seen as beneficial and helpful. The open questions provide insights why the participants focussed almost entirely on the visual annotative projection. One htw participant explained that she mostly uses visual annotations,

*"[...] because you have an overview of all features and their assignment."*

Along the same line of argumentation a Chalmers participant says:

*"Visual annotative is the simplest view for annotating features in the code. The procedure is not complicated to learn, and it is easy to see the separation of concerns. Visual annotations clearly show different features, and to which feature the code belongs to. So, I used it when making changes to the code and adding new features."*

Another Chalmers participant says that she focussed on visual annotations, because using them

*"[...] it is very clear to see the base code together with all possible variants."*

With regard to textual annotations one Chalmers participant explicitly explained:

*"These annotations are not so often used for our project. [...] #ifdef directives make the code harder to view and understand in comparison to visual annotations."*

The same participant claims:

*When creating new features, it is better to use modular projections, you can see clearly what features there are in the project. With the modular projection, you can select the desired class to refine, and it also forces you to write the code densely rather than widely scattered. The new habit makes it easier to find different features and make changes to the features later on.*

#### 5.3.5.2 Do developers frequently switch projections and edit the same code using different variability representations side-by-side? (RQ2)

Three participants claimed that they sometimes switched projections, two that they switched rather seldomly. However, a Chalmers participant says:

*"I became quite familiar with visual annotations and found it easy to use. [...] So, I didn't switch projections to be honest."*

So, we cannot count her “seldom”-response. Note that she reported that she neither used the modular nor the textual annotative projection (cf. Fig. 5.3), but the variant projection “often”, which is questionable based on her response. All (other) participants found the ability to switch projections on demand (very) useful in general. A Chalmers participant notes:

*“The annotated view is good enough for implementing the project, and sometimes when necessary other views, such as modular and variant, can be applied to assist in bug finding or adding new features.”*

One htw participant claims that she switched projections:

*“To check if my variant looks like expected.”*

A participant of the Chalmers group came to the same conclusion. She also comments:

*“There are always advantages and disadvantages in different views, and [...] the developer can make use of the view that is most appropriate to solve the problems at hand.”*

Four participants (out of five valid responses) claim that they have edited the same feature artifact using different representations side-by-side. Yet, one Chalmers participant could not remember in which situations she used side-by-side editing eventually. So, we count her “yes-I-used-it” rather weakly. The Chalmers participant who did not switch projections and focussed on visual annotations says about side-by-side editing:

*“I found it a bit confusing. I was getting lost as to what I was working on at that time.”*

The other participants found side-by-side editing helpful in general. One htw participant says that she edited the same code using visual annotative and variants side-by-side. Likewise, a Chalmers participant says that she explored visual annotations and modules side-by-side while adding and removing code.

### 5.3.5.3 Which IDE elements do developers use and which shortcuts? (RQ3)

Notice that the monitoring of IDE elements and shortcut usages was stable and independent of the MPS-specific listeners (i.e., we used an additional `AWTEventListener` to monitor IDE elements and shortcuts). Thus, we count the results valid as opposed to the general editor usage monitoring capturing the editing ratio.

According to the data our interaction monitoring captured, the top three IDE elements used by the htw students (in relation to their overall activities) are the projectional editors (60%), the project explorers

(19%) and the messages window (18%), which shows errors, warnings, and infos (e.g., compiler messages). So the students spent most of their time on development activities.

Figure 5.4 shows the accumulated monitoring results for the shortcut (a.k.a. *keybinding*) usage of the three htw students in relation to the overall shortcut usage of MPS-specific and PEOPL-specific shortcuts, respectively. The MPS-specific shortcut usage shows that they regularly used MPS' autocompletion, which is not surprising, since development in MPS heavily relies on autocompletion. They also often used the shortcut to open MPS' intentions menu, which is also used to assign a code snippet to a feature. All other MPS-specific shortcuts are quite common and not surprising (e.g., copy, cut, paste, and undo). Figure 5.4b shows the data for switching between projections, which basically confirms the htw group's questionnaire responses. They typically switched between visual annotations and a variant projection. Note that the modular projection cannot be reached via a shortcut. Instead, developers must currently use the modular project explorer, and thus the corresponding data is unfortunately not included.

Note that two participants of the Chalmers group had problems using MPS itself, which can be very efficiently commanded through keyboard shortcuts, each of which can be triggered from MPS' menu structure, too. To help developers in the editing process, we provide a

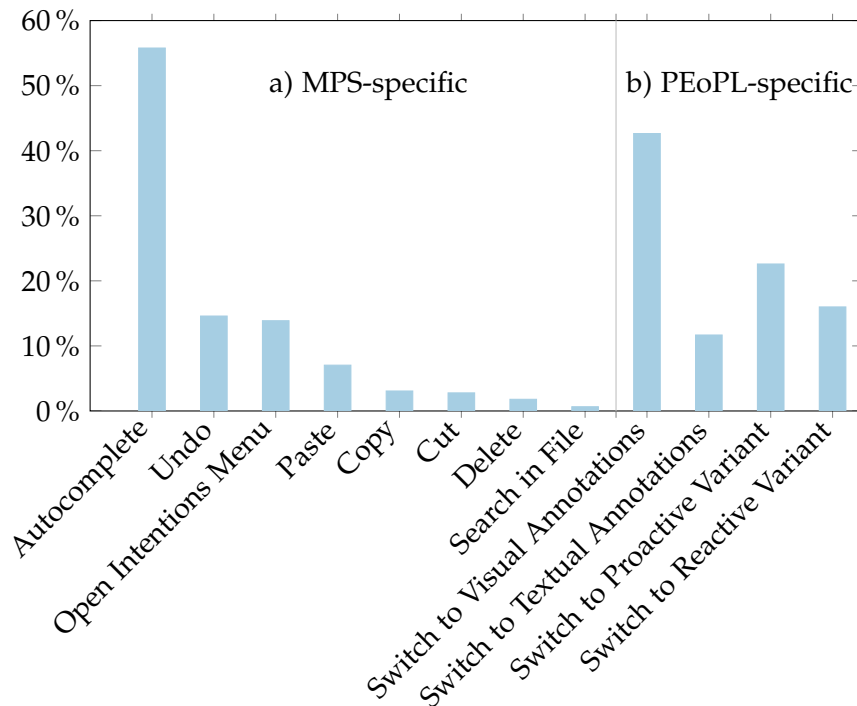


Figure 5.4: Interaction monitoring results for the usage of shortcuts. Each value is relative to the overall usage of the MPS-specific we used and PEOPL-specific shortcuts, respectively.

one page cheat sheet with the 22 most important keyboard shortcuts. Yet, one Chalmers participant says:

*“It didn’t act like most IDEs [...] a difficulty was remembering all the keyboard shortcuts (some of which I couldn’t find in the menu).”*

Along the same line of argumentation, another Chalmers student says:

*“Too much short-cuts need to be memorized. So, maybe adding some buttons to the menu will be good.”*

In contrast, all htw students claim that they like working with short-cuts, which are in fact similar to the ones in JetBrains IntelliJ IDE<sup>6</sup>.

### 5.3.6 Threats to Validity

We now discuss internal and external threats to validity, as well as conclusion and construct threats.

#### 5.3.6.1 Internal Validity

We chose Java as the target programming language, increasing the chances to attract students to the project and study. Java is typically well known. We only admit students that have a good background knowledge of Java. Yet, two participants of the Chalmers group and one participant of the htw group left the impression that their experience in programming is at a rather mediocre level in general. So, we carefully assessed whether their answers in the questionnaire are relevant.

Our interaction monitoring could be erroneous, producing false results. We systematically tested the logging mechanisms and performed a trial run with the Chalmers group. Unfortunately, a participant of the htw group reported an error—we did not find before—that appears on Windows when the systems goes to sleep. We fixed the problem and excluded the erroneous, distorted data.

We also checked participants for color blindness, since it could have biased them towards textual annotations and modular projections.

Since we graded participants for their performance, we did not expect them to deliberately perform poorly.

#### 5.3.6.2 Conclusion Validity

Since our (valid) sample is too small, we use our evaluation only for the feasibility of the study setting. Thus, we do not interpret the study data for giving ultimate recommendations on representation usage. We only describe tendencies for using them, and thus omit statistical tests—that is, we only use descriptive statistics (boxplots).

<sup>6</sup> <https://www.jetbrains.com/idea/>



### 5.3.6.3 *Construct Validity*

When comparing variability representations, we need to ensure that participants have a comparable level of knowledge of each representation. To control this threat, we taught (classical) variability representations at a comparable level of detail. However, we found that a more detailed training is required to decrease the adoption barrier of modular representations. Annotations appear more natural and have a lower learning curve.

To ensure that students know what to do, we provided a detailed project description including an example scenario and our expectation of their implementation. Moreover, the description detailed our experience report expectations and elaborated in detail how we grade the project. Moreover, we conducted 12 hours of hands on sessions to demonstrate how they can use PEOPL.

We decided to eliminate a questionnaire response if answers indicate a contradiction in terms. We eliminated two questionnaires of htw participants. One response of a Chalmers student was a marginal case, but we decided to include the data (explained shortly).

### 5.3.6.4 *External Validity*

Our study is dedicated to PEOPL and its several variability representations. Comparing PEOPL with other (textual) variability implementation techniques in a user study (e.g., controlled experiment) is valuable, but was not intended at this stage.

A threat of a longitudinal study is that we cannot control the activities of participants. So, participants could have used MPS for other tasks than the study project. To mitigate this threat, we asked them to use the PEOPL instance exclusively for their project tasks. Using the interaction data, we also checked whether the used editors render PEOPL's internal representation or just any file that does not belong to the project.

In general, our results are only relevant for student and not expert programmers. To draw sound conclusions, future studies must include industry experts.

### 5.3.7 *Interpretation and Lessons Learned*

Unfortunately, our sample is too small, and the majority of students were too inexperienced with the different variability representations involved in PEOPL (based on our impression). Thus, we cannot draw sound conclusions. Yet, we still made some interesting observations.

First, according to their feedback, all students liked working with PEOPL and found the tool usable in general. The usability problems issued by participants are mainly MPS-specific (and not PEOPL-specific). MPS does not behave like a text editor, since it directly

operates on the underlying AST. For instance, a common problem was over deletion—that is, deleting an AST node, deletes its entire subtree. For instance, deleting a closing curly brace, deletes the entire method with all its content. This and other issues have been identified before [31]. The hotspots are planned to be fixed in the future by JetBrains.

Second, one Chalmers participant found working with different projections side-by-side confusing. Although all other participants generally liked the ability to use different projections of the same feature artifact in parallel, we plan to provide indicators in the IDE that better guide side-by-side development. For instance, it could be helpful to clearly separate full annotative and variant editors, using dedicated IDE positions for the respective editor windows. Another idea would be to introduce a right hand vertical bar (or another marker) to indicate the variant view. Based on the feedback and before conducting the htw study, we introduced a variant projection indicator at the top of a projected file variant as illustrated in Figure 3.1e.

Third, the project task focussed too much on transitioning a single software product line into an SPL. Although both groups had an assignment to realize a from-scratch-implementation task, they did not implement many additional features from scratch in their projects. So the study just confirms the better adoption performance of annotative approaches (cf. Sec. 5.1.2) but lacks maintenance and more advanced evolution tasks. These should be included in future studies.

Fourth, our findings confirm the results of a previously conducted controlled experiment by Siegmund et al. [171]. They investigated the correctness, response times and search behavior of students in five bug fixing tasks. They divided the eight student participants into a CPP (five students) and a FH group (three students). Two out of three students expressed that they were unhappy to be in the FH group, since they did not like the modular paradigm. We also found that the modular variability representation challenges students. In our opinion, this reluctance appears to be a result of the need to learn a completely new development paradigm. It seems that this barrier and especially the short time frame to complete their development project led to the rather low usage of the modular projection. In fact, most students focussed on the visual annotative view, since it is the closest technique to what they already know (i.e., the program still looks like a regular program that is just enriched with annotations). Note that these results contradict our own experience made in our case studies. We found using different projections helpful, especially the modular projection (cf. Sec. 5.2.5). Consequently, expertise appears to play an important role.

Fifth, we found that one or two lectures on variability implementation is too short for students to learn the necessary development skills. Although the time frame was sufficient to enable them to understand

the concepts, they were still not proficient enough to independently use and choose between the different variability representations (i.e., beyond annotations). Two htw participants even did not know that there is a modular projection. One htw student told us that it would be nice to see a feature in isolation, which is in fact possible using our modular projection. Another htw student indicated a very high exclusive usage of the modular projection in the questionnaire, although the interaction monitoring clearly contradicts this statement. The students obviously did not implement the voluntary assignment for using the modular projection, and thus we count their questionnaire answers as not relevant and exclude their data. Altogether, we conclude that a complete course on variability implementation with more hands on sessions would be better suited for setting up a longitudinal study than just a few lectures (cf. the SPL course at the Carnegie Mellon University<sup>7</sup>). This way, the knowledge and experience of participants could be significantly improved, leading to more valid results. We would like to point out that PEOPL is still a great tool to learn the different variability representations. The code can be explored in parallel, and thus differences between techniques can be grasped easily.

Sixth, based on our experience, it might be worthwhile to create a catalogue of tasks providing hypotheses for which variability representation might leverage in which situation. To evaluate these hypotheses, we plan to conduct a controlled experiment for investigating exact usage scenarios of multiple projections, providing sound empirical results. We also plan to use this data for creating a recommender system, which supports developers to choose the best technique for a given task. Then, developers could use the validated catalogue and the recommender system, which would enable us to conduct a more solid longitudinal user study. We conclude that the catalogue, the recommender system, and eventually a better user training could lower the adoption barrier of modular approaches in the future. PEOPL's flexibility provides a valuable first step.

Finally, we summarize our observations and the pitfalls that should be avoided in another longitudinal study:

1. Teaching a comparable level of knowledge requires focussing more on the "esoteric" representations (modules, reuse, and variants), since annotations are easy to learn
2. Providing clear guidelines and a catalogue of examples when to use which technique
3. Providing a recommender system that facilitates projection usage and gives hints at development time

---

<sup>7</sup> <https://www.cs.cmu.edu/~ckaestne/17708/>

4. Using a better balanced study setting including a mixture of adoption, implementation, maintenance, and evolution tasks.

#### 5.3.8 Future Research Questions

Besides the aforementioned activities to provide a better study setting, we plan to investigate SPL complexity handling more detailed. Realizing variability is challenging: (i) product configurations need to be expressible in a structured way while providing guidance to handle and fix invalid configurations, and (ii) feature artifact dependencies need to be manageable. We want to identify possible shortcomings of PEOPL and determine whether PEOPL suffices for engineering and managing SPL variability. We propose to investigate the following research question:

*Is the IDE support for handling variability provided by PEOPL beneficial for mastering the complexity of real-world C applications? Which additional features are helpful or needed?*

In particular, we plan to analyze the usage history of our feature module dependency checker, which allows developers to conveniently resolve corresponding issues. We will determine the *ratio of dependency fixes*—that is, the number of navigations to dependent code, using the dependency checker over the number of corresponding module assignments. Likewise, we also plan to investigate the usage of the variant-based data-flow analysis and the corresponding fixes. Moreover, we plan to assess the time spent in the product line declaration, module configuration, and dependency resolution.

## 5.4 CONCLUDING REMARKS

We evaluated PEOPL from the SPL developer’s perspective using three ingredients. First, we used and extended existing classification frameworks to compare PEOPL with other SPL implementation techniques. We found that PEOPL compares well to other techniques, while enabling new interesting opportunities (e.g., lowering the adoption barrier of modular approaches). Second, we evaluated PEOPL’s practicality, scalability, and flexibility in eight Java-based product lines, finding that all can be realized, that our external representations are feasible, and that variant computation and rendering projections is quick. Third, we conducted two pilot user studies for investigating PEOPL’s usability and laying ground for more detailed future studies. We found that PEOPL is in fact usable and identified pitfalls that should be avoided in future studies.

Part III

REALIZING PEOPLE USING LANGUAGE  
ENGINEERING

Thus far, we discussed how developers use PEOPL to engineer and manage SPL variability. In the remainder of this dissertation, we focus on the internals of PEOPL, how language engineers use PEOPL to make target languages (e.g., Java, C, and fault trees) variability-aware and how new variability representations can be realized.

In this chapter, we provide an overview on tooling. We start by briefly elaborating why we use a language workbench for realizing the PEOPL approach (Sec. 6.1), and—together with conceptual details and concrete examples—why we use MPS in particular (Sec. 6.2). Moreover, we discuss the architecture of PEOPL in a feature model (Sec. 6.3), which helps understanding the general relationships.

### 6.1 WHY AND HOW PEOPL USES A LANGUAGE WORKBENCH

In 2005, Fowler [78] popularized the term language workbench [60, 61] characterizing several key concepts of modern projectional editing tools such as *Intentional Programming*<sup>1</sup> [173, 174] and MPS [145, 201]. He defines the key characteristics of a language workbench as follows [78]:

- C1: “The primary source of information is a persistent abstract representation”
- C2: “Language users manipulate a DSL through a projectional editor”
- C3: “Language designers define a DSL in three main parts: schema, editor(s), and generator(s)”
- C4: “Users can freely define new languages which are fully integrated with each other”
- C5: “A language workbench can persist incomplete or contradictory information in its abstract representation”

PEOPL leverages these characteristics. It stores variability in an internal representation—that is, feature artifacts are uniformly persisted in a single, variational AST (C1). Users manipulate this internal representation through diverse projectional editors, each representing a different external variability representation (C2). In particular, we conceive the core-variability language CoreVar, which provides the

---

<sup>1</sup> <http://www.intentsoft.com/>

structure for representing variability together with projectional editors and generators for dealing with variability (C<sub>3</sub>). Language engineers use CoreVar to add variability to arbitrary target (programming) languages (irrespective of artifact-types), which is possible since languages can be integrated into each other (C<sub>4</sub>). In fact, no parser is involved and AST nodes are always unambiguously assigned to a language.

For meaningful variational ASTs, CoreVar can easily be tailored to specific (programming) languages using CoreVar’s tailoring infrastructure and DSL. In this dissertation, we provide a tailoring to Java called *JavaVar* declaring which AST node types are annotatable, for instance, *Class* and *Statement*. Moreover, we briefly discuss a tailoring to fault trees. As part of the PEOPL project, Fey also realized a tailoring of CoreVar to the C programming language to underline the flexibility and practicality of the approach [74].

Altogether, to make a target (programming) language variability-aware in a meaningful manner, language engineers simply integrate:

1. the target language (e.g., Java),
2. the generic CoreVar language, and
3. the tailoring language (e.g., JavaVar), which uses CoreVar’s tailoring infrastructure and DSL to tailor CoreVar to the target language.

On a finale note, CoreVar is in fact black-box generic. It must not be manipulated to make a target language variability-aware. Any restrictions and extensions are defined in external tailoring languages.

## 6.2 THE MPS LANGUAGE WORKBENCH

To implement the concepts of the PEOPL approach, we use the MPS language workbench, which is available under the Apache 2.0 license. That is, we did not develop MPS, but simply use it to put PEOPL into practice. MPS evolved over the past decade to a mature tool that

1. provides the currently most advanced projectional editing facilities, which we need for rendering our external variability representations,
2. allows engineers to define how languages integrate in a sophisticated fashion, which we need for making target languages variability-aware in a meaningful way, and
3. provides all common language-design facilities of a modern language workbench [60, 61], which we need for providing an efficient and user-ready implementation with support for checking the SPL’s correctness.

We now discuss the basic language aspects used in MPS to construct a language: structure (a.k.a. *schema*), editor, and generator. Details on all language aspects available in MPS—for instance, so-called *actions* and *intentions* for efficient, on-the-fly AST editing—can be found in the literature [192, 195, 202].

### 6.2.1 Structure

In MPS, the *abstract syntax* (a.k.a. *meta model*) of a language is defined using so-called *language concepts* (cf. Fig. 6.1a-d), which declare the structure (i.e., properties, children, and references) of concept instances (AST nodes). Some instances (e.g., `ClassConcept` or `Interface` instances in Java) act as root nodes of an AST. These root nodes are contained by a so-called *model* (similar to a package in a Java program), so a model aggregates multiple ASTs, and thus we consider a program as one large AST.

If a language concept is instantiated, its children, and the children inherited from other concepts (explained shortly) are considered for instantiation as well. For instance, the `TryStatement` concept in our example declares a `body`, a `finally-body` and potentially multiple catch clauses (Fig. 6.1c). So if a `TryStatement` instance appears in the AST, it must have—according to the childrens’ cardinality “[1]”—exactly one `body`, and exactly one `finally-body`. Catch clauses in turn are optional (i.e., the cardinality is “[0..n]”). Using these rules, an AST is always constructed in compliance with the involved language(s).

To specialize a language concept, we can use language concept inheritance, which in fact corresponds to the idea of inheritance in object-oriented languages. Figure 6.1e gives an example overview of the inheritance hierarchy of language concepts. The abstract concept `BaseConcept` is the super concept of all concepts in MPS. For instance, the `TryStatement` concept inherits from `Statement`, which in turn inherits from `BaseConcept`.

Another way to extend language concepts are so-called *node attributes*, which allow the extension of language concepts without modifying the structure of the AST explicitly (i.e., they are non-invasive). In particular, any concept inherits from `BaseConcept`, which holds an `Attribute` as a child (cf. Fig. 6.1e and 6.1a). So sub-concept instances of `Attribute` (e.g., `NodeAttribute`) can be attached to any AST node. Thus, a node attribute is in fact an annotation.

In PEOPL, we leverage such annotations to embed additional (variability-related) information in a non-invasive fashion into the AST. In particular, we use a `Fragment` concept instance to mark an AST node as a feature artifact (details are discussed shortly in Ch. 7). Note that each fragment (i.e., feature artifact) knows its feature module via a cross-tree reference (cf. Fig. 6.1d). Such cross-tree references are



```

abstract concept BaseConcept
extends <default>
implements <none>

instance can be root: false
alias: <no alias>
short description: <no short description>

properties:
shortDescription : string
alias : string
virtualPackage : string

children:
smodelAttribute : Attribute[0..n]

references:
<< ... >>
    
```

(a) BaseConcept language concept in MPS core

```

concept Statement
extends BaseConcept
implements ILocalVariableElement ...

instance can be root: false
alias: <statement>
short description: <no short description>

properties:
<< ... >>

children:
<< ... >>

references:
<< ... >>
    
```

(b) Statement language concept in Java

```

concept TryStatement
extends Statement
implements IContainsStatementList ...

instance can be root: false
alias: try {...} finally
short description: <no short description>

properties:
<< ... >>

children:
body : StatementList[1]
finally : StatementList[1]
catchClause : CatchClause[0..n]

references:
<< ... >>
    
```

(c) TryStatement language concept in Java

```

@attribute info
multiple: false
role: Fragment
attributed concepts: BaseConcept
concept Fragment extends NodeAttribute

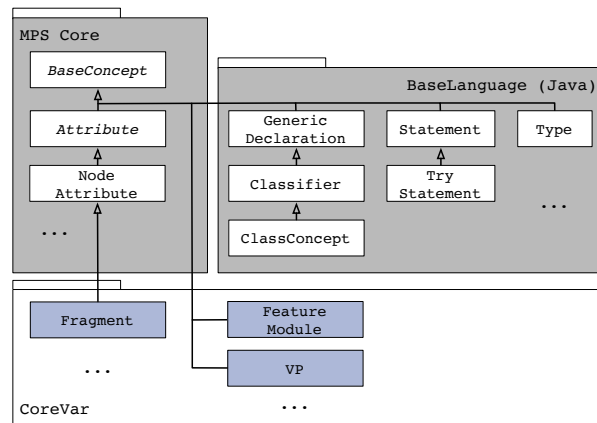
instance can be root: false
alias: <no alias>
short description: <no short description>

properties:
...

children:
<< ... >>

references:
myModule : FeatureModule[1]
...
    
```

(d) Fragment node attribute language concept in CoreVar



(e) Overview on the language concept inheritance hierarchy in MPS

Figure 6.1: Different concrete language concepts that are used to construct a language’s structure (a), and the inheritance hierarchy of language concepts (b)

actively maintained by MPS (e.g., method call to method declaration), and thus the AST is actually a directed graph.

In summary, using different languages in MPS enables constructing a concrete AST using diverse language concepts. Thus, a concrete AST may contain language concept instances from potentially multiple languages (cf. C4 in Sec. 6.1).

### 6.2.2 Editor

Projectional editors allow developers to work conveniently with the variational AST (instead of manipulating it manually). Each language concept requires a projectional editor defining the rules for rendering it into concrete syntax. An editor definition consists of so-called *editor cells* [195]. Figure 6.2 shows an example for the editor of the TryStatement language concept (Sec. 6.1c). The editor adds a constant cell with the keyword `try` and surrounds the body cell with curly braces. In fact, the body cell embeds the editor of the body's StatementList (cf. Sec. 6.1c), which in turn just embeds each statement's editor, and so on. Likewise, the `catchClause` cell embeds a CatchClause editor for each catch clause child. Then, the editor of the CatchClause concept renders the `catch` keyword and the curly braces (using constant cells). Consequently, editors also follow a hierarchy.

Moreover, editors can accommodate so-called *editor hints* defining in which context the rendering rules are to be applied. In fact, a language concept can have different editors through different (combinations of) hints. Then, in a concrete model, MPS allows developers to select (a.k.a. *push*) hints for a given editor component—that is, a concrete instance of an editor for a given AST node (typically a root node). Note that Figure 6.2 shows the default editor for the TryStatement concept. Default editors are used by MPS if no hint has been pushed by the developer or there is no editor implementing one of the hints selected.

On a final note, we leverage such projectional editors and hints for rendering the variational AST into PEOPL's diverse external representations (as we will discuss in Ch. 8).

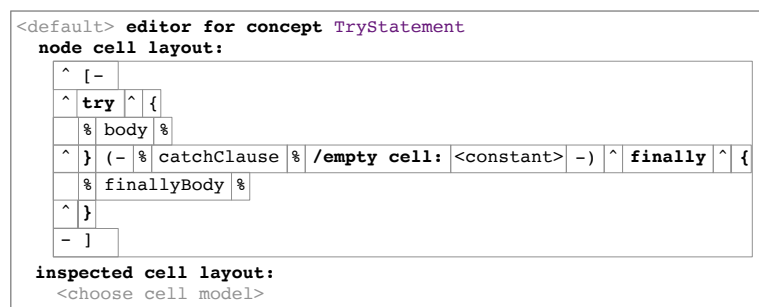


Figure 6.2: Projectional editor of the TryStatement language concept

### 6.2.3 Generator

The persisted ASTs need to be transformed into program text which can be compiled and executed. As such, MPS uses a generator to translate a language concept either into another concept or into compilable program text. In other words, language engineers can specify for a concept a tree-to-tree mapping (i.e., a *model-to-model transformation*) or a text generation (i.e., *model-to-text transformation*).

In PEOPL, we leverage transformation rules to reduce specialized, variability-related language concepts into other language concepts, and to construct variants using a preprocessing script. In a nutshell, before another model-to-model or model-to-text transformation kicks in, PEOPL simply decides whether a feature artifact must be removed or the tree must be restructured (as explained shortly in Ch. 7).

On a final note, it was not clear whether MPS scales for the PEOPL approach. With our case studies (Sec. 5.2.2), we provide evidence that it is indeed feasible to use MPS as PEOPL's base platform. In fact, we also experimented with other tool-implementation technologies. For instance, for testing PEOPL's internal representation [28], we used the academic *Snippet System* prototype [112, 113], which however did not scale. That is, generating a Berkeley DB variant and writing the source code to disk takes about 2 hours in the Snippet System, while in MPS the very same variant generation algorithm is quick (about 5 seconds). The reason is that MPS—in contrast to the Snippet System prototype—heavily relies on caching the internal data structure, which is crucial for software of Berkeley DB size (with over 192,000 AST nodes). To be fair, the snippet system was not made for such a demanding environment (i.e., handling ASTs and large numbers of nodes). It focusses on dealing with office documents, the smart reuse of elements within these documents, and more [113].

## 6.3 AN OVERVIEW ON PEOPPL'S ARCHITECTURE

Figure 6.3 shows a feature model with PEOPL's key features—that is, we package PEOPL's languages and some language aspects into features to discuss the relationships in a feature-oriented fashion. Each language is categorized into either internal variability representation, external variability representation, or variability management facilities (i.e., depending on the main contribution of the language within PEOPL). The following chapters on PEOPL's internals are organized accordingly. We now provide an overview on each category. Moreover, we briefly discuss the *extras*, such as helpful tools, developed in the course of the PEOPL project.

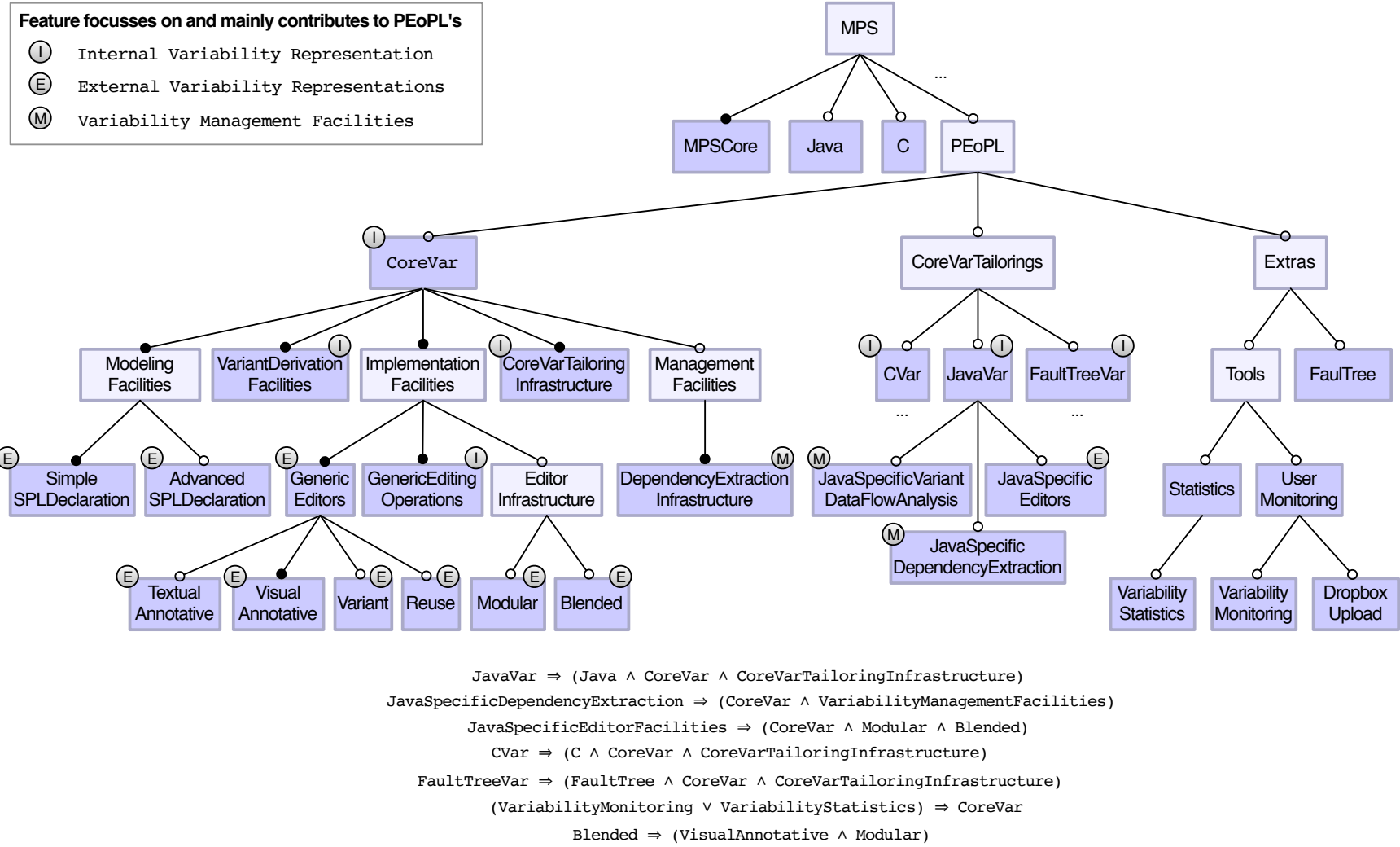


Figure 6.3: PEOPL's feature model with the most important languages and aspects being packaged as features. An indicator highlights for each language whether it focusses on PEOPL's internal variability representation, external variability representation, or its variability management facilities. Note that each language can, of course, also contribute to the other parts, but we decided to make the main contribution clear. Dependencies between PEOPL's languages are modeled as well. For instance, JavaVar depends on Java, CoreVar, and the CoreVar tailoring infrastructure, which enables tailoring CoreVar to Java. That is, JavaVar only functions correctly if the other two languages are available.

### 6.3.1 *Internal Variability Representation*

CoreVar is the most important language in PEOPL, since it implements a formalism for constructing variational ASTs. Focussing on internals, several languages (and language aspects) build upon this formalism (shown as sub-features in Fig. 6.3). In fact, CoreVar provides:

1. a set of *generic editing operations* that are used to manipulate variability information in the AST,
2. *variant derivation facilities* that enable the composition of feature modules and the transformation of the variational AST to a specific variant, and
3. a *tailoring infrastructure* that contains the definition of a DSL, which is used within specific tailorings to tailor CoreVar to a target language.

The PEOPL tool provides exemplary tailoring specifications for C, Java, and fault trees in the languages CVar, JavaVar and FaultTreeVar, respectively. More tailorings for other languages can easily be added, since PEOPL is extensible.

### 6.3.2 *External Variability Representation*

To be able to manipulate the internal variability representation, CoreVar provides languages for simple and advanced SPL declaration together with a set of generic editors (annotative, variant, and reuse) that can be directly used for target languages. Moreover, it provides the infrastructure to build modular and blended editors, which (for now) have to be implemented in target-language-specific editors (e.g., Java-specific).

### 6.3.3 *Variability Management Facilities*

CoreVar also provides a generic infrastructure for dependency extraction. Yet, the concrete references to be checked are target language dependent (e.g., method call and variable reference) and must be declared. These are realized in target-language-specific dependency extractions (e.g., Java-specific).

### 6.3.4 *Extras*

Together with PEOPL, we also package a fault tree language, tools to extract statistics from a product line (e.g., number of feature artifacts) and to monitor a user's interaction. These tools are language independent, extended with additional facilities to understand CoreVar.

## PEOPL'S INTERNAL VARIABILITY REPRESENTATION

---

In this chapter, we present PEOPL's internal variability representation. We start with CoreVar's language structure, describing variational ASTs (Sec. 7.1). For manipulating such ASTs, we conceive generic editing operations (Sec. 7.2). Then, we discuss the process of deriving individual variants, which takes a variational AST as input (Sec. 7.3). Thereafter, we present CoreVar's tailoring infrastructure and DSLs for describing variability aspects and restricting annotations to a meaningful level for a given target language (Sec. 7.4). Finally, we demonstrate the concrete tailoring of CoreVar to Java and fault trees (Sec. 7.5).

### 7.1 COREVAR'S LANGUAGE STRUCTURE

We start with an abstract formalization of variability (Sec. 7.1.1) and introduce a variational AST formalism on this basis (Sec. 7.1.2). To put the formalizations into practice, we conceive a general meta model (Sec. 7.1.3), and provide a concrete implementation in MPS (Sec. 7.1.4).

#### 7.1.1 Variability Formalism

CoreVar adopts, modifies, and extends the *structured document algebra* (SDA) [24]—an abstract formalization of feature modularity. The SDA enables variability through *feature modules*, which assign *fragments* to *variation points* (VPs). A VP represents a point of variability in a product line and a fragment marks content as a feature artifact. Let  $V = \{vp_1, vp_2, \dots\}$  be a set of VPs and  $F = \{f_1, f_2, \dots\}$  a set of fragments. A feature module  $m : V \rightsquigarrow F$  is an injective partial function assigning fragments from  $F$  to VPs from  $V$ . In contrast to SDA (not using an injective function), fragments are unique to a VP—that is, a fragment cannot be assigned to multiple VPs to reflect a homogeneous extension. Notice that this is not a limitation in practice, but just eases our implementation (explained shortly).

The domain  $dom(m)$  of a module  $m$  is the set of VPs assigned by  $m$ . The module  $m$  assigns a fragment to the VP  $vp$  if  $vp \in dom(m)$ , otherwise  $vp$  is not related to  $m$ . A feature module can assign a VP only once, but multiple modules can assign different fragments to the very same VP.

For a fragment  $f \in F$ , we define the helper function  $VP(f)$  returning the VP  $vp$  associated with  $f$  (possible as of injection). Similarly, let  $M(f)$  be the helper function returning the module  $m$  assigning  $f$  to  $VP(f)$ . Moreover, we define the helper function  $F_{vp}(vp)$ , which returns all fragments of a given VP  $vp$ . All helper functions return  $\perp$  if there is no module  $m$  assigning  $f$  to  $vp$ —that is, before a developer explicitly chooses  $m$  for assigning  $f$  to  $vp$ .

For the fragment  $f_1$  in our Berkeley DB example, the helper function  $M(f_1)$  yields the module *Base* and the helper function  $VP(f_1)$  the VP  $vp_1$  (cf. Fig. 3.1f and 7.1). All VPs in Berkeley DB

$$V_{\text{Berkeley}} = \{vp_0, vp_1, \dots, vp_{12}\}$$

are associated with fragments via modules, for instance

$$\begin{aligned} \textit{Base} &: \{vp_0 \mapsto f_0, vp_1 \mapsto f_1, vp_4 \mapsto f_{4.1}, \dots\} \\ \textit{Statistics} &: \{vp_3 \mapsto f_3, vp_4 \mapsto f_{4.2}, \dots\} \end{aligned}$$

Feature modules assign fragments to VPs in their domain, for instance

$$\begin{aligned} \textit{dom}(\textit{Base}) &= \{vp_0, vp_1, vp_4, vp_5, vp_8, vp_{11}\} \\ \textit{dom}(\textit{Statistics}) &= \{vp_3, vp_4, vp_7, vp_9, vp_{10}, vp_{12}\} \end{aligned}$$

Notice that  $\textit{dom}(\textit{Base}) \cap \textit{dom}(\textit{Statistics}) = \{vp_4\}$ , and thus *Base* and *Statistics* share  $vp_4$  (cf. Fig. 3.1f and Fig. 7.1).

### 7.1.2 Variational AST Formalism

Using this variability formalism, we now discuss how AST nodes are made variable. In the course of this, we distinguish heterogeneous and homogeneous extensions.

**HETEROGENEOUS EXTENSIONS.** AST nodes are made variable—that is, marked as a feature artifact—by annotating them with fragments from  $F$ . Let  $AST = \{n_1, n_2, \dots\}$  be a set of AST nodes. A variational AST  $vast : F \rightarrow AST$  is an injective, non-surjective function assigning AST nodes to fragments from  $F$ .

The image  $vast(F)$  of a variational AST  $vast$  is the set of AST nodes annotated with fragments. An AST node  $n$  is annotated if  $n \in vast(F)$ . The domain  $\textit{dom}(vast)$  is the set of fragments  $F$  (i.e., the AST nodes assigned to fragments by  $vast$ ). Due to injection, fragments are unique to AST nodes. Every fragment annotates exactly one node, but not every node must be annotated. The helper function  $FN(n)$  either returns the fragment annotating the node  $n \in AST$  or  $\perp$  if the node is not annotated (i.e.,  $n \notin vast(F)$ ).

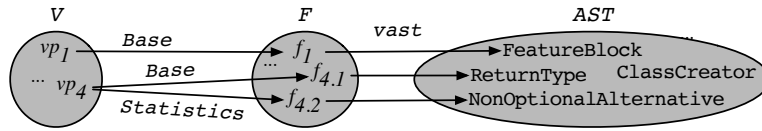


Figure 7.1: Relationships of AST nodes, fragments, and variation points

Figure 7.1 shows an excerpt of the mappings in our Berkeley DB example (Fig. 3.1f). For instance, the image of Berkeley DB’s variational AST is a set of concrete AST nodes (i.e., feature artifacts):

$$\begin{aligned} vast(F_{Berkeley}) = \{ & \text{FeatureBlock,} \\ & \text{ReturnType,} \\ & \text{NonOptionalAlternative, ...} \} \end{aligned}$$

Moreover, the helper functions yield fragments, VPs, and modules:

$$\begin{aligned} FN(\text{ReturnType}) &= f_{4.1} \\ VP(FN(\text{ReturnType})) &= vp_4 \\ M(FN(\text{ReturnType})) &= \text{Base} \end{aligned}$$

**HOMOGENEOUS EXTENSIONS.** Recap that a fragment cannot be assigned to multiple VPs, since a module is an injective partial function. Thus, we need a different way to enable homogeneous extension support (known from AOP, where advice can be applied to multiple join points). To provide support and realize our reuse projection (cf. Sec. 3.3.7), a VP can appear in the variational AST through a *placeholder* element—that is, a special AST node referring to a VP [28].

Let  $P = \{p_1, p_2, \dots\}$  be a set of placeholder members that appear in  $AST$ —that is,  $P \subset AST$ . Then, we use a non-injective total function  $appear : P \rightarrow VP$  assigning VPs to placeholders. The image  $appear(p)$  yields the VP to which  $p$  refers. Notice that  $appear$  is cycle-free, since we forbid assigning fragments to placeholders (explained shortly). Figure 7.2 gives an example where a `FeatureBlock` is reused twice through two `Placeholder` instances that appear in the  $AST$ . In fact, if  $vp_1$  is filled with content during derivation (i.e., with the `FeatureBlock`), then the `Placeholder`’s are filled with this content as well. To enable editing in our reuse projection, the `FeatureBlock` is rendered at each `Placeholder`’s  $AST$  position.

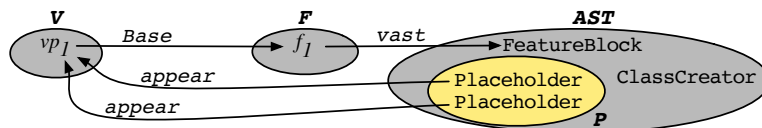


Figure 7.2: Placeholders enable developers to reuse AST nodes



### 7.1.3 Language Structure of Variational ASTs

We now propose a generic language structure (i.e., meta model) for our variational AST formalism. Figure 7.3a shows the structure and relationships of CoreVar's language concepts (upper half), and how CoreVar is applied to arbitrary trees (lower half). To discuss model details, we use a stack example which makes the assignment of fragments to VPs explicit in an internal (Fig. 7.3b) and an external representation (Fig. 7.3c).

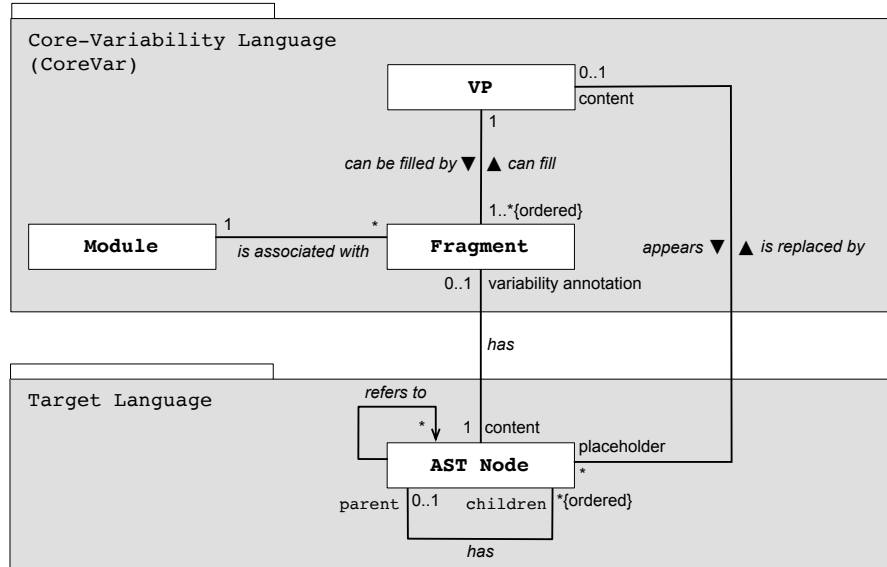
According to the formalisms, a module is associated with a set of fragments (cf. Fig. 7.3a) and each fragment is associated with exactly one module. Thus, fragments are unique to modules. Logically, a fragment acts as a container for content (i.e., represents a feature artifact). Given its content, a fragment can fill exactly one VP. Figure 7.3b illustrates the idea of containers and the relationships between fragments and modules by coloring the AST nodes that are annotated with a fragment (and all descendants of the node). For instance, the fragment  $f_1$ , which is associated with the module *Log* (light gray box), annotates the method declaration `sLog`. Notice that VPs are illustrated as elements next to the fragment (e.g.,  $vp_1$ ) to clarify which fragments can fill the VP with content.

To clarify the relationships of a variational AST, Figure 7.3c shows an external modular representation which makes the assignment of fragments to VPs through modules explicit. For instance, the *Base* module assigns the fragment  $f_0$ —which contains the class `Stack`—to  $vp_0$  (cf. root node in Fig. 7.3b). Within the class, multiple VPs appear that are external to the *Base* module—that is, the module does not assign fragments to these VPs.

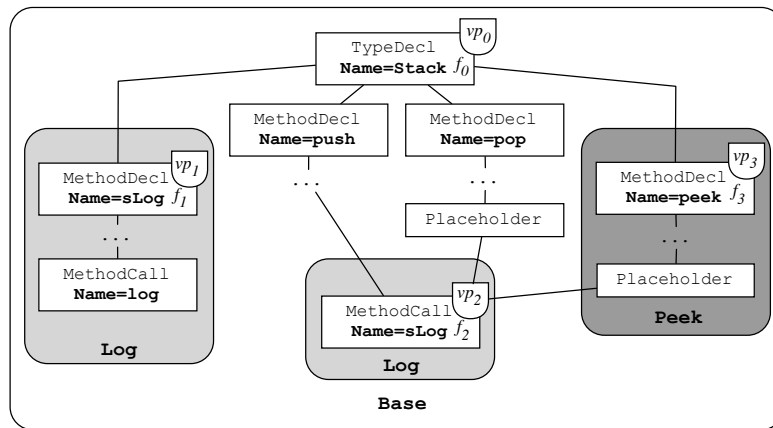
Notice that the VP  $vp_2$  appears twice in the *Base* module and once in the *Peek* module. To fill  $vp_2$  with content, the *Log* module assigns  $f_2$ —which contains the method call `sLog`—to it. In the variational AST depicted in Figure 7.3b, the method call appears also three times. Once as a child within the push method, and twice via placeholders that refer to `sLog`'s VP  $vp_2$ . Consequently, a VP can in fact appear multiple times in the AST (cf. Fig. 7.3a). In particular, according to our model, an AST node can have two different types of children: other AST nodes or a placeholder that refers to a VP.

Moreover, a VP can be filled by multiple alternative fragments—that is, a VP maintains an ordered list of fragments (cf. Figure 7.3a). In fact, a VP represents a potential point of variability that can be filled by multiple fragments with content. Yet, during product generation only one fragment can fill a VP at a time, so fragments assigned to the same VP are alternative to each other. Figure 3.1 gives an example. There, the fragments  $f_{4.1}$  and  $f_{4.2}$  are assigned to  $vp_4$ .

On a final note, to compute the domain  $dom(m)$ —the set of VPs a module  $m$  administers—we simply collect the VPs instantiated by the



(a) Core-variability language (CoreVar) and arbitrary target language



(b) Internal representation of modules, fragments and VPs

```

Base
{
  class Stack{
    int[] sData; int top;
    vp1
    void push(int data){
      vp2
      sData[++top] = data;
    }
    int pop(){
      vp2
      return sData[top--];
    }
    vp3
  }
}

Log
vp1 -> f1 {
  void sLog() {
    log(sData[top]);
  }
}
vp2 -> f2 {
  sLog();
}

Peek
vp3 -> f3 {
  int peek() {
    vp2
    return sData[top];
  }
}
    
```

(c) Making modules, fragments and VPs and explicit in an external representation

Figure 7.3: CoreVar's language structure (a) and internal and external representation of modules, fragments and VPs (b-c)

fragments associated with  $m$ . For instance, in Figure 7.3c, the module *Log* is associated with  $f_1$  and  $f_2$ , which instantiate  $vp_1$  and  $vp_2$ , respectively (i.e.,  $dom(Log) = \{vp_1, vp_2\}$ ).

#### 7.1.4 Implementation in MPS

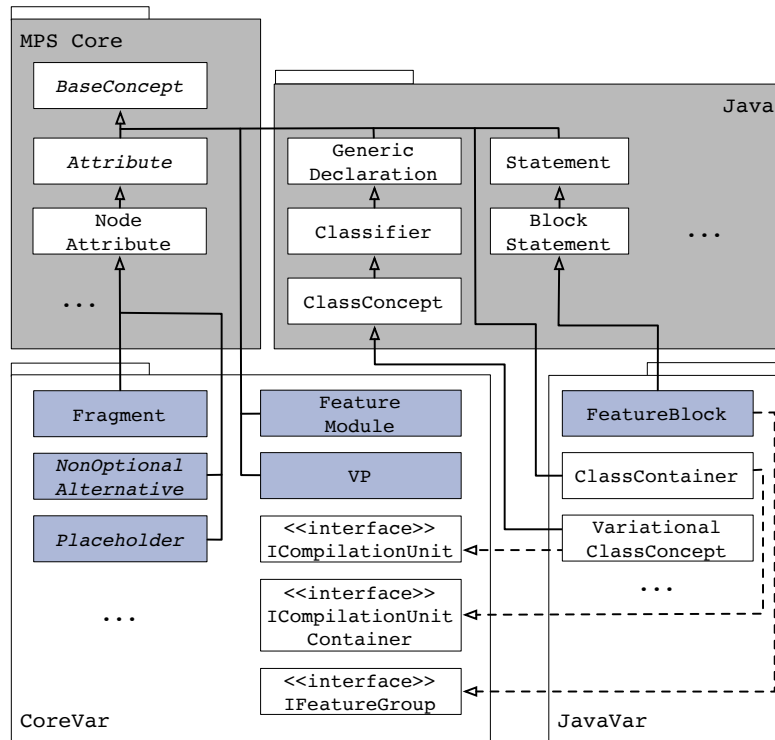
We now implement the language structure of CoreVar in MPS. We use 76 language concepts and five interface concepts—similar to a Java interface—in total. Thereof, the simple SPL declaration uses nine, the advanced SPL declaration 15 language concepts. CoreVar's tailoring infrastructure introduces 25 language concepts. The core language structure comprises five main language concepts, and 22 language concepts easing and enabling our implementation. Figure 7.4a shows an inheritance diagram for the main concepts relevant for adding and handling variability. Notice that the most important concepts are highlighted. Since CoreVar only inherits and uses concepts from MP-SCore, not MPS' model of Java, it is language-independent and can be used to make other languages variability-aware.

Based on our generic language structure (cf. Sec. 7.1.3), we define the associations of CoreVar's main language concepts. In fact, we enable making AST nodes optional in MPS. Figure 7.4b shows the most important associations of variability-related language concepts. Since the fragment concept inherits from MPS' `NodeAttribute` concept (cf. Fig. 7.4a), fragments can be directly attached to existing AST nodes (cf. the `Fragment` to `BaseConcept` relationship in Fig. 7.4b). This way, existing nodes are made optional in a non-invasive fashion. Each fragment belongs to exactly one module and to exactly one VP (cf. Fig. 7.4b). While fragments are contained by the respective AST node, modules and VPs are persisted by a dedicated container concept attached to the project root (not depicted).

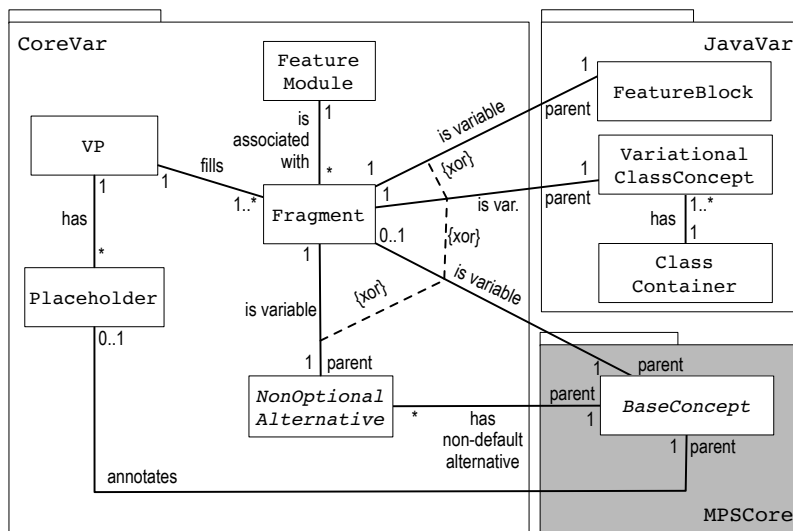
Using these concepts, the original AST structure is not changed. However, for convenience reasons when long sequences of AST siblings (i.e., `Statement` instances) belong to the same module, JavaVar (more details are explained shortly) introduces the `FeatureBlock` concept, which groups AST nodes that are mapped to fragments belonging to the same module and VP. In fact, the `FeatureBlock` concept implements CoreVar's feature group interface, which can be implemented by other languages and for other concepts as well. Notice that each `FeatureBlock` has exactly one fragment, and thus is optional by default (Fig. 7.4b).

Placeholders are also realized as node attributes (Fig. 7.4a). In fact, placeholders annotate an existing node that is to be replaced with the VP's content (i.e., the AST node annotated by the `Fragment`) in projections and during variant derivation (Fig. 7.4b). For instance, a placeholder can annotate an empty statement or an empty classifier member in Java.

We use annotations also for non-optional AST nodes that cannot have siblings in the AST (e.g., exactly one return type is required in a method declaration according to Java's syntax). For such non-optional nodes, CoreVar provides the `NonOptionalAlternative` language concept, whose instances are used to annotate a non-optional node, while holding an alternative node for it (cf. Fig. 7.4b). For



(a) Inheritance diagram of language concepts in MPS



(b) Realization of CoreVar's language structure in MPS

Figure 7.4: Realizing PEOPL's internal variability representation using the MPS language workbench

example, the return type of the method `preload` is annotated by a `NonOptionalAlternative` holding an alternative return type (cf.  $f_{4.2}$  in Fig. 3.1f).

Optimally, all AST nodes of the SPL relate to at least one module—more precisely, to a fragment that belongs to a module. In fact, nodes belonging to an SPL's commonality should be mapped to a core, top-level module (e.g., *Base* in Fig. 7.3b and 7.3c). CoreVar provides the interface concepts `ICompilationUnitContainer`, which holds multiple `ICompilationUnit` instances. In a nutshell, the container concept makes it possible to have multiple alternative compilation units (e.g., classes). These interfaces are implemented in CoreVar tailorings, for instance, JavaVar's `ClassContainer` and `VariationalClassConcept` as shown in Figure 7.4a. Notice that a `VariationalClassConcept` must have exactly one fragment, and thus the root node is assigned to a module (Fig. 7.4b).

## 7.2 COREVAR'S GENERIC EDITING OPERATIONS

To manipulate the AST, CoreVar provides five basic variability-related editing operations which can be refined by its tailoring extensions (e.g., JavaVar): *assign variability*, *assign wrapper variability*, *assign alternative*, *remove variability*, and *select and appear*—all available via a popup menu over a code snippet in the concrete syntax (cf. Fig. 3.2) or triggered automatically by an editing gesture (e.g., typing `#ifdef` or `#elif`).

### 7.2.1 Assign Variability Operation

The operation marks an AST node as optional, for instance, the class `DatabaseImpl` assigned to  $f_0$  (Fig. 3.1f). An algorithm creates a new VP  $vp_i \in V$  and fragment  $f_i \in F$ , and annotates the selected AST node  $n_j$  with the fragment such that  $vast(f_i) := n_j$ . Then, the developer selects the desired module  $m_k$  assigning  $f_i$  to  $vp_i$ .

### 7.2.2 Assign Wrapper Variability Operation

The operation marks only a wrapping node as variable, not its body (the wrappee). The idea is to annotate the wrapping node such that its wrappee is not removed during variant derivation (explained shortly). Assigning variability to a wrapping node corresponds to the assign variability operation. The only difference is that an additional annotation called *wrapper* is added to the target node which refers to the wrappee (cf. Fig. 3.1f,  $vp_{10}$ ).

### 7.2.3 Assign Alternative Operation

The operation marks an AST node  $n_a$  as an alternative to another AST node  $n_o$ . Let  $n_o$  be variational with  $FN(n_o) \neq \perp$ , and let its fragment  $f_o := FN(n_o)$  be assigned to the VP  $vp_o := VP(f_o)$  with  $VP(f_o) \neq \perp$ . If the alternative node  $n_a$  is variational with  $FN(n_a) \neq \perp$ , then an algorithm changes the alternative node's module  $M(FN(n_a))$  such that it assigns the fragment  $FN(n_a)$  to  $vp_o$ , which is then associated with  $f_o$  and  $FN(n_a)$ . If the node  $n_a$  is not variational, an algorithm creates a fragment  $f_a$  in  $F$ , assigns it to  $vp_o$  (according to the developer's module selection), and annotates  $n_a$  with the fragment such that  $vast(f_a) := n_a$ . Figure 7.1 gives an example. The fragments  $f_{4.1}$  given by  $FN(\text{ReturnType})$  and  $f_{4.2}$  by  $FN(\text{NonOptionalAlternative})$  are both assigned to  $vp_4$  by their respective modules.

Notice that  $n_o$  and  $n_a$  are typically, but not necessarily siblings in the AST (e.g., statements alternative to each other). Recap that some non-optional AST nodes cannot have siblings (e.g., return type declarations). CoreVar provides the concept `NonOptionalAlternative` for such non-optional nodes. In fact, `NonOptionalAlternative` instances are used by the assign-alternative operation to annotate  $n_o$ , while holding an alternative node  $n_a$ . For instance, the `preload` method's return type is annotated by a `NonOptionalAlternative` holding an alternative return type (cf.  $f_{4.2}$  in Fig. 3.1f). Notice that non-optional language concepts are declared in CoreVar tailorings (explained shortly in Section 7.4).

### 7.2.4 Remove Variability Operation

Removing variability from a specific variational AST node is straightforward. The operation's algorithm simply removes the fragment annotation and the corresponding VP, if no other fragment fills it.

### 7.2.5 Select and Appear Operation

The operation enables reusing an AST node. Developers explicitly execute a *select* operation, which is then followed by (multiple) *appear* operations. In fact, *select* and *appear* is analogous to copy and paste without introducing a physical copy, but only references. Let  $n_p$  be the node to be reused (i.e., selected) and  $n_1, n_2, \dots, n_n$  the nodes where  $n_p$  shall appear. Notice that  $n_p$  must be annotated with a fragment to enable reuse (cf. Fig. 7.3b). So, if  $n_p$  is not variational  $FN(n_p) = \perp$ , then the *select* operation's algorithm calls the assign variability operation to mark  $n_p$  optional. Then,  $VP(FN(n_p))$ —that is, the VP associated with  $n_p$ —is cached for connecting it with placeholders. Each *appear* operation then annotates the target node under the cursor (e.g.,  $n_1$ )

with a placeholder, which in turn refers to the cached VP. Figure 7.3b gives an example, where  $vp_2$  is referenced by two placeholders.

### 7.3 COREVAR'S VARIANT DERIVATION FACILITIES

The derivation of variants is a three-step process. First, we calculate the set of fragments contained in the variant by composing feature modules. Second, we remove the placeholders referring to a VP, which is not filled by a fragment. If the VP is filled by a fragment, the corresponding fragment's AST node replaces the placeholder. Third, we remove any variability by iterating over all fragments in the tree to either remove the fragment, to remove the annotated node, or to restructure the tree (for wrapper and non-optional alternative nodes).

#### 7.3.1 Composition of Feature Modules

Composing modules results in a transient variant set  $F_{\text{variant}}$ , containing all fragments of a variant. SDA provides three operations to compose modules: *addition* (+), *subtraction* (-), and *overriding* ( $\rightarrow$ ), which we adopt in PEOPL as follows.

Let  $F_m$  and  $F_n$  be the set of fragments associated with a module  $m$  and  $n$ , respectively. The addition of two modules  $m + n$  fails if  $m$  and  $n$  contain conflicting fragments:  $\text{dom}(m) \cap \text{dom}(n) \neq \emptyset$ . In Figure 7.1, fragments  $f_{4.1}$  and  $f_{4.2}$  are conflicting, since both assign fragments to the VP  $vp_4$ .

Without conflicting fragments, addition results in a greater set of fragments  $F_{m+n} = \{F_m \cup F_n\}$  and therefore a larger module, serving as input for further operations. Notice that the fragments of the larger module reflect a preliminary or the final variant set  $F_{\text{variant}}$ . The subtraction of  $m - n$  is the set  $F_{m-n} = \{f | f \in F_m \wedge VP(f) \notin \text{dom}(n)\}$ . In other words, we remove the fragments of  $m$  that share a VP with fragments of  $n$ , so subtraction removes fragments from the variant set. Overriding is simply a combination of addition and subtraction:  $m \rightarrow n =_df m + (n - m)$  to enable replacement.

Table 7.1 shows different compositions for our Berkeley DB example (Fig. 3.1). For instance, configuration (2), adding the modules *Base* and *Memory\_Budget*, results in a valid set of fragments. In contrast, configuration (3), adding *Base* and *Statistics*, is erroneous, as fragments  $f_{4.1}$  and  $f_{4.2}$  fill the same VP  $vp_4$ .

To resolve conflicting fragments, we use subtraction and overriding. For instance, configuration (4) removes  $f_{4.1}$ . The resulting fragment set is valid, but the AST invalid (both return types pruned), which is detected during derivation. Type-checking module composition is part of our future work.

Using overriding, developers decide between conflicting fragments. For instance, configuration (5) denotes that all fragments of *Base* re-

No	Module configuration	Variant's fragment set ( $F_{\text{variant}}$ )	Valid
(1)	<i>Base</i>	$f_0, f_1, f_{4.1}, f_5, f_8, f_{11}$	✓
(2)	<i>Base + Memory_Budget</i>	$f_0, f_1, f_2, f_{4.1}, f_5, f_6, f_8, f_{11}$	✓
(3)	<i>Base + Statistics</i>	$f_0, f_1, f_3, f_4, f_5, f_7, f_8, f_9,$ $f_{10}, f_{11}, f_{12}$	✗
(4)	<i>Base - Statistics</i>	$f_0, f_1, f_3, f_5, f_7, f_8, f_9,$ $f_{10}, f_{11}, f_{12}$	✗
(5)	<i>Base</i> $\rightarrow$ <i>Statistics</i>	$f_0, f_1, f_3, f_{4.1}, f_5, f_7, \dots$	✓
(6)	<i>Statistics</i> $\rightarrow$ <i>Base</i>	$f_0, f_1, f_3, f_{4.2}, f_5, f_7, \dots$	✓

Table 7.1: Example module compositions for Berkeley DB (cf. Fig. 3.1)

place those conflicting with *Statistics*. Notice that ordering matters for overriding, since configuration (6) includes the fragments of *Statistics* instead. Moreover, notice that a feature selection can be realized by overriding modules according to their declaration order. For instance, configuration (6) reflects a selection of *Base* and *Statistics* based on the ordering defined in Figure 3.1b (i.e., *Base* has lowest priority).

### 7.3.2 Remove or Replace Placeholders

To remove or replace placeholders, we iterate over them. For each placeholder, an algorithm checks whether the corresponding VP is filled by a fragment in the variant's fragment set  $F_{\text{variant}}$ . In fact, let  $p$  be the placeholder to be checked. Then,  $\text{appear}(p_{\text{check}})$  yields the VP  $vp_{\text{check}}$  to which the placeholder refers. Subsequently, our algorithm uses the helper function  $F_{\text{vp}}(vp_{\text{check}})$  to yield the set of fragments filling the VP  $vp_{\text{check}}$ . The placeholder  $p$  is removed if  $F_{\text{vp}}(vp_{\text{check}}) \cap F_{\text{variant}} = \emptyset$ . Otherwise we check for each fragment  $f_1, f_2, \dots, f_n \in F_{\text{vp}}(vp_{\text{check}})$  whether it is in  $F_{\text{variant}}$ . Notice that only one fragment  $f_i \in F_{\text{variant}}$  can fill  $vp_{\text{check}}$ , since else the variant is invalid, which is detected during composition. Since  $f_i$  marks an AST node as optional, we can simply yield this node  $\text{vast}(f_i)$  and replace the placeholder  $p$  with it—that is, we copy  $\text{vast}(f_i)$  to the position of the placeholder  $p$ .

On a final note, in our proof of concept implementation placeholders are realized as AST node annotations. Thus, we remove or replace the placeholder's parent and not the placeholder itself in practice.

### 7.3.3 Remove Variability from the Variational AST

We now remove any variability (i.e., fragment) from our AST such that  $\text{vast}(F) = \emptyset$ . Figure 7.5 shows our algorithm, which takes the variational AST  $\text{vast}$  and the variant's fragment set  $F_{\text{variant}}$  as input. We iterate over all fragments in the variational AST (Line 2). If the



```

1 func removeVariability(func vast, set<fragment> F_variant) {
2   for each f ∈ dom(vast) {
3     if(f ∈ F_variant) f.delete; // delete only the fragment
4     else {
5       if(hasWrapperAnnotation(vast(f)) { // handling the wrapper
6         for each n ∈ getWrappee(vast(f)).children {
7           vast(f).add prev-sibling(n); // moving the content up
8         }
9       } else if(hasNonOptionalAlternatives(vast(f))) {
10        node a = popFirstAlternative(vast(f)).getNodeIHold();
11        a.addAll(getNonOptionalAlternatives(vast(f)));
12        vast(f).replace with(a); // replace the node
13        return;
14      }
15      vast(f).delete; // delete the node and its fragment
16    } } }

```

Figure 7.5: Algorithm to remove variability from the AST

fragment is in  $F_{\text{variant}}$ , we simply delete the fragment to remove variability (Line 3). Otherwise, if the AST node has a wrapper annotation, we move the wrappee's children up in the tree as siblings of the wrapper (Line 7) and remove the wrapper (Line 15). If the AST node is non-optional (i.e., has `NonOptionalAlternatives`), we pop the first alternative and get the node it holds (Line 10), such as the alternative return type `PreloadStats` in Fig. 3.1f. Since an AST node may have multiple non-optional alternatives, we must add all of them to the popped alternative (Line 11). Then, we can safely replace the non-optional node in the tree with it (Line 12). If the AST node neither has a wrapper annotation nor is a non-optional alternative, we simply remove the AST node (Line 15).

## 7.4 COREVAR'S TAILORING INFRASTRUCTURE AND DSLS

PEoPL enables tailoring CoreVar to a specific target language. We now explain the shape of tailorings in general, and in the next section we explain concrete tailorings using Java and fault tree examples.

### 7.4.1 Annotatable Nodes Declaration

Without restriction, the editing operations of CoreVar allow annotating any AST node (also non-optional ones) with fragments, which may lead to syntactically incorrect variants. To declare annotatable nodes and restrict editing operations to a meaningful level, CoreVar provides *can-assign-variability* and *can-assign-alternative* declarations (i.e., DSLs used for tailorings). Moreover, language engineers can declare a custom behavior for operations and language concepts of a given target language (e.g., extending the `assign alternative` operation with custom behavior for method declarations). Figure 7.6 provides an overview showing the shape of these declarations.

```

Annotatable node declarations and custom behavior for <language>
Can-assign-variability declarations
1  Inclusions:
2  simple inclusion for concepts: <list of concepts>
3  parameterized inclusion for node: (sourceNode) -> boolean {
4    <return true (included) or false (not included)>
5  }
6  << more inclusions >>
7  Exclusions (overrides inclusions):
8  simple exclusion for concepts: <list of concepts>
9  parameterized exclusion for node: (sourceNode) -> boolean {
10   <return true (excluded) or false (not excluded)>
11 }
12 << more exclusions >>

Can-assign-alternative declarations
13 all rules from can-assign-variability: <default: true>
14 non-optional node concepts: <list of concepts>
15 << inclusions and exclusions >>

Custom behavior declarations
16 custom alternative operation for <language concept>
17 (originalNode) -> node<> {
18   <return the node alternative to the original node>
19 }
20 << more custom alternative creations >>

```

Figure 7.6: The shape of the annotatable nodes declaration DSL

Can-assign-variability declarations restrict our assign variability operation. The declarations are distinguished into concept instance *inclusions* and *exclusions*. Both declarations can be either *simple* or *parameterized*. In a *simple inclusion*, language engineers provide a list of annotatable language concepts (Line 2), which enables the annotation of concept instances (and their subconcept instances due to concept inheritance). At runtime the assign variability operation checks whether the selected concept instance of a tailored language can be optional. *Parameterized inclusions* enable more flexible runtime checks (Line 3–5), since engineers can specify whether a node can be annotated using program logic (e.g., a Boolean expression). If the parameterized inclusion returns true, the node is annotatable, otherwise not. Exclusions override inclusions. This way, engineers can include a concept and its subconcept instances, and refine this inclusion using restrictions. Exclusions are also simple (Line 8) or parameterized (Lines 9–11) and implemented analogously to inclusions.

The can-assign-alternative declaration restricts CoreVar's assign alternative operation. In fact, it allows adopting the rules declared in can-assign-variability (Line 13), and declaring non-optional language concepts (Line 14). Notice that the latter enables the assign alternative operation to annotate the original, non-optional AST node with `NonOptionalAlternative` instances. In addition, engineers can add new inclusions and exclusions for the assign alternative operation.

Aside from can-assign-variability and can-assign-alternative declarations, language engineers can also specify custom behavior for CoreVar's editing operations. For instance, it is possible to customize the behavior of the assign alternative operation (Lines 16–19). This way, initialized siblings can be created in a custom setting.

#### 7.4.2 Wrapper Declaration

Which nodes in the AST can have a wrapper annotation—where the wrapping node is variable, but not its subtree (wrapper body)—is target-language-dependent. Figure 7.7 shows the shape of our wrapper declarations DSL. A wrapper declaration specifies the wrapper's language concept and the corresponding wrappee (child node). At runtime, the declarations are used by the assign wrapper operation to annotate the wrapper and refer to the wrappee. These annotations and references, in turn, are used during variant derivation to replace the wrapper with its wrappee (if the wrapper is not included in the variant).

```
Wrappers that can be partially annotated in <language>
instance of <language concept> replaced by its <child node>;
<< more wrapper declarations >>
```

Figure 7.7: The shape of the wrapper declaration DSL

#### 7.4.3 Further Declarations

While tailoring CoreVar to a target language, language engineers typically make further declarations and implementations.

1. To ease the handling of a group of AST siblings that belong to the same module, CoreVar's IFeatureGroup convenience interface concept can be implemented. The IFeatureGroup interface provides the basic infrastructure for creating, merging and splitting groups. This behavior can be refined in the concrete implementation of a CoreVar tailoring language (e.g., JavaVar). To generate a language-specific grouping behavior and making our generic editing operations aware of concrete groups, CoreVar provides a feature group declaration DSL as illustrated in Figure 7.8. Engineers declare which concrete IFeatureGroup

```
Feature group declarations for <language>
concrete IFeatureGroup implementation <IFeatureGroup language concept>
groups <language concept>
<< more IFeatureGroup implementations >>
```

Figure 7.8: The shape of the feature group declaration DSL

concept (i.e., the concept implementing the `IFeatureGroup` interface) groups which concrete concept instances (e.g., statements in Java).

2. Since ASTs should relate to at least one top-level feature (for meaningful variational ASTs), language engineers should implement the `ICompilationUnit` and `ICompilationUnitContainer` interface concepts (cf. Fig. 7.4). Then, the annotation of root nodes with fragments is enforced and compilation units that are alternative to each other can be created.
3. To analyze and manage the SPL, tailoring CoreVar typically also includes declaring variability-specific type-system and data-flow rules for the target language (cf. Ch. 9).

## 7.5 TAILORING COREVAR TO A TARGET LANGUAGE

We now tailor CoreVar to Java and fault trees.

### 7.5.1 Tailoring CoreVar to Java

**ANNOTATABLE NODES DECLARATION.** Figure 7.9a shows an excerpt of our annotatable nodes declaration for Java (implemented in JavaVar). For example, fragments can annotate `Statement` concept instances (Line 5), and thus also all `Statement` subconcept instances (e.g., `IfStatement` and `TryStatement` instances). Moreover, we declare that `throwsItems` of method and constructor declarations can be annotated (Lines 12–19)

In JavaVar’s `can-assign-alternative` declaration, we adopt the rules of `can-assign-variability` (Line 25), add new rules, and declare non-optional nodes. For instance, JavaVar enables developers to annotate the language concept instances of `Type`, `Expression`, and `Visibility` with `NonOptionalAlternative` instances (cf. Figure 7.9a, Lines 26–28, and the `NonOptionalAlternative` in Fig. 3.1f).

All in all, due to concept inheritance, not many declarations are needed for Java. The `can-assign-variability` declaration has nine simple and eight parameterized inclusions, as well as two simple and one parameterized exclusion. The `can-assign-alternative` declaration adopts these rules plus three non-optional node inclusions, one simple and one parameterized inclusion.

Moreover, JavaVar customizes the `assign alternative` operation for `BaseMethodDeclaration` (Lines 31–34). In particular, we create an alternative method declaration with the original method’s name and a random suffix (Line 33).

**WRAPPER DECLARATION.** Figure 7.9b shows JavaVar’s four wrapper declarations. For instance, a `TryStatement` can be replaced by its body. Notice that the `AbstractLoopStatement` concept is abstract

```

Annotatable node declarations and custom behavior for Java


---


Can-assign-variability declarations
1  Inclusions:
2  simple inclusion for concepts: Interface,
3                                BaseMethodDeclaration,
4                                BaseVariableDeclaration,
5                                Statement,
6                                SwitchCase,
7                                ParameterDeclaration,
8                                DotExpression,
9                                StaticInitializer,
10                               IncompleteMemberDeclaration
11
12 parameterized inclusion for node: (sourceNode) -> boolean {
13     return (sourceNode.parent.isInstanceOf(BaseMethodDeclaration)
14     && sourceNode.hasRole(BaseMethodDeclaration : throwsItem));
15 }
16
17 parameterized inclusion for node: (sourceNode) -> boolean {
18     return (sourceNode.parent.isInstanceOf(ConstructorDeclaration)
19     && sourceNode.hasRole(ConstructorDeclaration : throwsItem));
20 }
21 ...
22 << more inclusions >>
23
24 Exclusions (overrides inclusions):
25 simple exclusion for concepts: PlaceholderMember,
26                                TernaryOperatorExpression
27 ...
28 << more exclusions >>
29
Can-assign-alternative declarations


---


30 all rules from can-assign-variability: <default: true>
31 non-optional node concepts: Type,
32                                Expression,
33                                Visibility
34 << inclusions and exclusions >>
35
Custom behavior declarations


---


36 custom alternative operation for BaseMethodDeclaration
37 (originalNode) -> node<BaseMethodDeclaration> {
38     node<BaseMethodDeclaration> altMeth = originalNode.concept.new instance();
39     altMeth.name = originalNode.name + "_" + (int) (Math.random() * 900);
40     return altMeth;
41 }
42 << more custom alternative creations >>

```

(a) Annotatable nodes declaration for Java

```

Wrappers that can be partially annotated in Java
instance of AbstractLoopStatement replaced by its body;
instance of IfStatement replaced by its trueBody;
instance of SynchronizedStatement replaced by its block;
instance of TryStatement replaced by its body; ...

```

(b) Wrapper declaration for Java

```

Feature group declarations for Java


---


concrete IFeatureGroup implementation FeatureBlock
groups Statement
<< more IFeatureGroup implementations >>

```

(c) FeatureBlock declaration for Java

Figure 7.9: Tailoring CoreVar to Java in JavaVar

and replaced by its body. This way, concrete language subconcepts of `AbstractLoopStatement`, such as `WhileStatement` and `ForStatement`, are inherently supported.

**FURTHER DECLARATIONS.** `JavaVar` also declares variability-specific type-system and data-flow rules. Moreover, it extends the Java language with an implementation of the `IFeatureGroup` convenience concept that eases handling variability. In fact, `JavaVar`'s `FeatureBlock` concept implements the `IFeatureGroup` interface to group statements belonging to the same module (cf. Fig. 3.1f). In fact, we enforce that any statement (except partially annotated wrappers) are contained by at least one `FeatureBlock` (e.g.,  $f_1$  in Fig. 3.1f). Otherwise, individual statements would need to be annotated (as they are siblings in the AST). Notice that in the projections, the block's statement list is just rendered without showing curly braces (cf. statement-level vertical bars in Figure 3.1c). A `FeatureBlock` also extends its enclosing statement list's scope to make the `FeatureBlock`'s statements visible to its siblings. During variant derivation, if the `FeatureBlock`'s module is in the variant, it is replaced by its statements, otherwise removed.

In addition, `JavaVar` implements the interfaces `ICompilationUnit` and `ICompilationUnitContainer` in the concepts `ClassContainer` and `VariationalClassConcept`, respectively (cf. Fig. 7.4a). This way, each AST relates to at least one module.

### 7.5.2 Tailoring `CoreVar` to Fault Trees

The language structure of fault trees is simple (cf. Sec. 3.4.2). It is comprised of language concepts reflecting events (*basic*, *intermediate*, and *top*), gates (e.g., *and*, *or* and *not*), and facilities to attach/detach nodes from a fault tree. Tailoring `CoreVar` to fault trees is straightforward, since `FaultTreeVar` simply allows annotating events and gates. Gates wrap gates—that is, a gate can be replaced by another gate. For instance, in Figure 3.10b, the gate *Sensor detected collision* can be replaced by the gate *Unavoidable obstacle* in variants (cf. Fig. 3.10d). In fact, we customized the wrapper behavior such that gates with only one child gate are replaced by it automatically. In `FaultTreeVar`, we do not implement a feature group, since gates already group fault tree nodes.

## 7.6 CONCLUDING REMARKS

`CoreVar` enables adding variability to arbitrary languages. It provides variant derivation facilities and a set of generic, extensible editing operations. We demonstrated that tailoring `CoreVar` to mainstream languages such as Java is straightforward using `CoreVar`'s tailoring

facilities and DSL (i.e., the engineering effort is low). We have also shown that non-textual languages are supported (i.e., fault trees).

## PEOPL'S EXTERNAL REPRESENTATIONS

Thus far, we discussed PEOPL's internal variability representation. In this chapter, we show how we realized PEOPL's external variability projections on top. We start with CoreVar's generic editors (annotative, variant and reuse). Then, we illustrate how the current language-dependent modular and blended projections are realized for a given target language. Finally, we discuss the realization of PEOPL's modeling facilities (i.e., simple, advanced, and expert SPL declarations).

## 8.1 IMPLEMENTATION FACILITIES

Recap that each language concept requires a projectional editor, defining the rules for rendering language concepts into concrete syntax (cf. Sec. 6.2.2). For instance, Java's BlockStatement editor renders its StatementList and surrounds it with curly braces. Moreover, recap that editors can accommodate editor hints, defining in which context the rendering rules are to be applied. A language concept can have different editors through different hints, and thus the concrete syntax can appear in different ways. We leverage hints in PEOPL to switch external variability representations. PEOPL's variability-related language concept is the fragment. For each of a fragment's external representations, we implement a projectional editor (with a specific hint) that is oblivious to the target language (e.g., Java).

Next, we explain how we realize our projectional editors.

8.1.1 *Rendering Annotations*

Realizing textual and visual annotation projections is easy. Figure 8.1 shows a visual annotative editor for Fragments—CoreVar's main concept for making ASTs variability-aware. Recap that editor definitions consist of editor cells [195]. The *[annotated node]* cell embeds the editor of the AST node annotated with the fragment (Lines 3, 4, and 6). The fragment editor renders fragment concept instances in three ways—that is, the editor checks whether the annotation is disciplined, within a line of code, or complex:

1. Disciplined annotations are rendered as a vertical bar and a feature module selection (Line 3). The macros `#VerticalBar#` and `#Module#` refer to so-called *editor components*—editors reusable among different editors. We leverage these components to reuse vertical and horizontal bars across different editors. The  $\oplus$ -sign



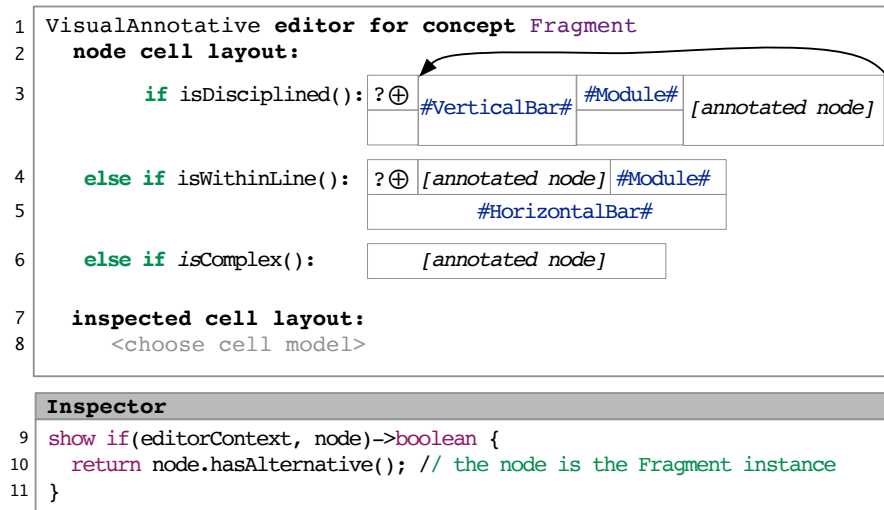


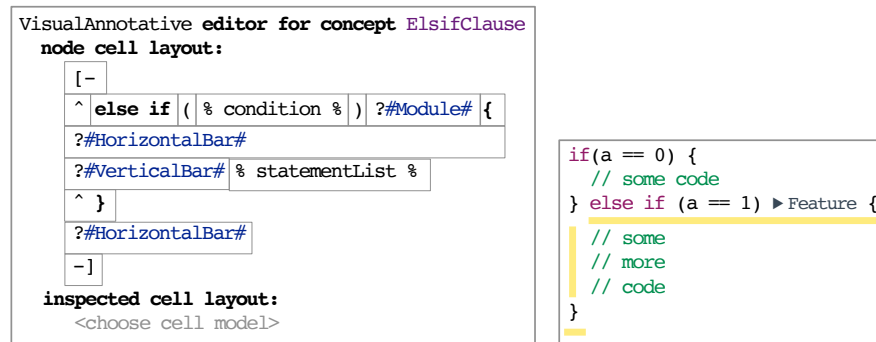
Figure 8.1: Simplified visual annotative projectional editor for fragments (upper half) with a cell inspector adding a rendering condition (show if) for the  $\oplus$ -sign (lower half)

is a constant cell (Line 3). The question mark denotes that a rendering condition is attached to the cell. In fact, for each editor cell a so-called *inspector* allows configuring the cell, inter alia, adding a rendering condition. So, we check in the  $\oplus$ -sign's inspector simply whether the fragment has an alternative (Fig. 8.1, Line 10)—that is, the  $\oplus$ -sign is rendered if the fragment has alternatives assigned to the same VP.

2. Undisciplined annotations within a line of code are underlined with a horizontal bar (Lines 4–5). The  $\oplus$ -sign is rendered if the fragment has an alternative.
3. Annotations requiring a more specific syntax are propagated to the customized target node's editor (Line 6), which targets a target-language-specific concept that recognizes a fragment and provides respective coloring.

For instance, colored else-if clauses require a more complex coloring. Figure 8.2a shows the editor, which renders a combination of vertical and horizontal bars. Notice that the variability-related editor components (e.g., `HorizontalBar`) are optional. In fact, if no fragment annotates the else-if clause, no module information and bars are rendered. Figure 8.2b shows an example rendering.

Non-optional alternatives—AST nodes that cannot be simply removed without invalidating the tree—also require a more specific rendering. Figure 8.3 shows the default, colored concept editor for `NonOptionalAlternative` instances. The `NonOptionalAlternative` editor basically renders the node annotated with the `NonOptionalAlternative` instance and the



(a) A visual annotative editor for colored else-if clauses (b) A colored rendering of an annotated else-if clause

Figure 8.2: Projectional editor (a) and concrete syntax (b) for variability-aware else-if clauses

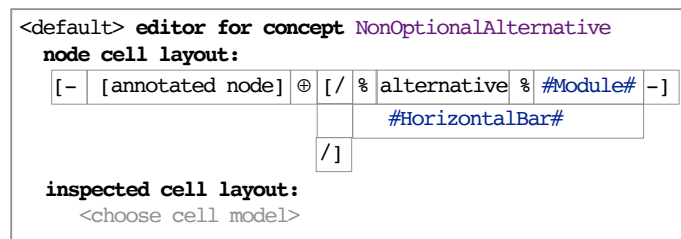


Figure 8.3: Default editor for colored non-optional alternatives, which is independent of target languages

node—the alternative—it holds. To visualize mutual exclusiveness the editor renders the  $\oplus$ -sign.

On a finale note, the textual annotative editors look similar and just add keywords, such as `#ifdef`. Thus, rendering annotations is typically simple.

### 8.1.2 Rendering Variants

Realizing variant projections is also straightforward. Proactive and reactive variant editors for the `Fragment` concept simply check whether the `Fragment` instance is in the variant's fragment set (cf. Sec. 7.3.1). Figure 8.4 illustrates this ideas for a reactive variant editor. The editor has an optional annotated node cell, which conditionally embeds the editor of the annotated node. In the cell's inspector, we add the rendering condition. In fact, the annotated node is hidden, if the fragment is not in the fragment's variant set. Complex renderings—that is, if the target-concept editor handles the fragment—constitute an exception. For instance, the `wrapper` concept editor renders the `wrappee` if the node is not in the variant's fragment set.

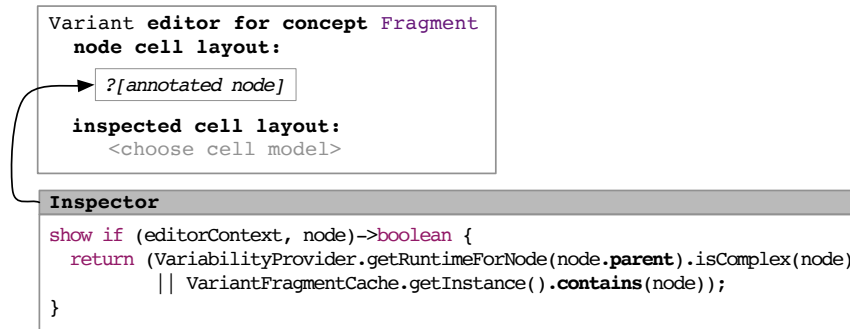


Figure 8.4: Reactive variant editor (without coloring)

The proactive (i.e., colored) variant editor is basically a clone of the colored annotative editor (cf. Fig. 8.1), but additionally checks whether the fragment is in the variant’s fragment set.

On a final note, to provide a variant-specific file explorer as discussed in Section 3.5, we simply check for each root node (e.g., class or interface) if it is in the current variant’s fragment set.

### 8.1.3 Rendering Reuse

For our reuse projection (cf. Sec. 3.3.7), we simply render the content filling the placeholder’s VP. Figure 8.5 shows a basic Placeholder editor where we use a so-called *custom cell*—that is, a cell enabling to accommodate cells for other nodes in the AST. In our reuse projection, the Placeholder editor’s custom cell renders other AST nodes—the ones filling the VP—instead of the placeholder itself.

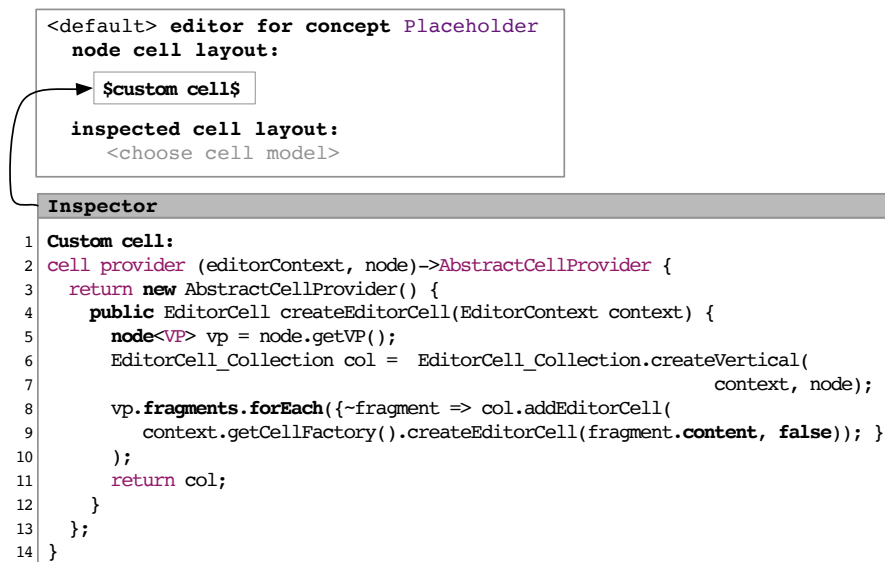


Figure 8.5: Simple editor for reuse projections

The custom cell's inspector enables implementing the program logic for rendering placeholders. First, we create an abstract cell provider that enables creating a custom editor cell (Lines 3–4). Then, we get the VP to which the placeholder refers (Line 5). Subsequently, we create a vertical cell collection—that is, an editor cell, which holds other cells and renders each cell in a vertical ordering (Lines 6–7). Thereafter, we iterate over all fragments that can fill the VP, create an editor cell for the fragment's content (i.e., the node it annotates), and add the content to the cell collection (Lines 8–10). Finally, we return the cell collection, which now contains all the content of a placeholder's VP—that is, all nodes that the VP's fragments annotate.

#### 8.1.4 *Rendering Modules*

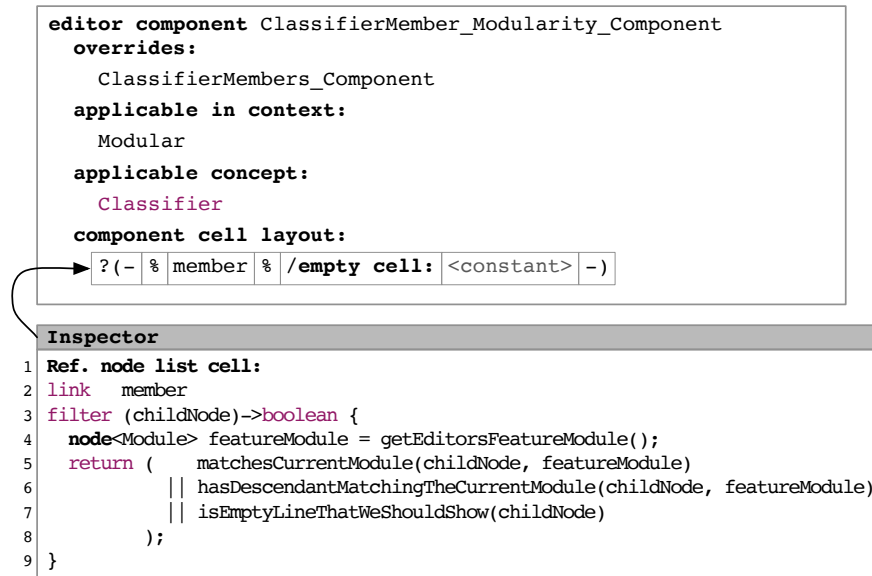
A module projection enables developers to explore the SPL's code and files in a modular fashion. For each file, we simply check whether it is introduced or refined by the selected module (e.g., a *Memory\_Budget* file explorer for the example depicted in Fig. 3.1d would show the `DatabaseImpl` file). In contrast to projecting annotations and variants, projecting modular code is currently language-dependent. In fact, the editor rendering fragments cannot simply hide annotated nodes—that is, we also need to show refined AST nodes, which do not relate to the selected feature module, but have a descendant that relates to the selected module.

Luckily, it is still feasible to project a mainstream language's AST (e.g., a Java AST) into a concrete syntax, representing feature modules (similar to the ones of FH and AHEAD). In fact, we only need three fragment-aware editors, defining how to render the Java language concepts `ClassifierMember` (e.g., method declarations), `IVisible` (e.g., *public* or *private*), and `StatementList` (e.g., a method declaration's or a block statement's body) in the presence of variability. In other words, we override these editors (from Java) with editors handling variability, induced by fragments. In the following, we discuss how we realize these three editors and provide editing support.

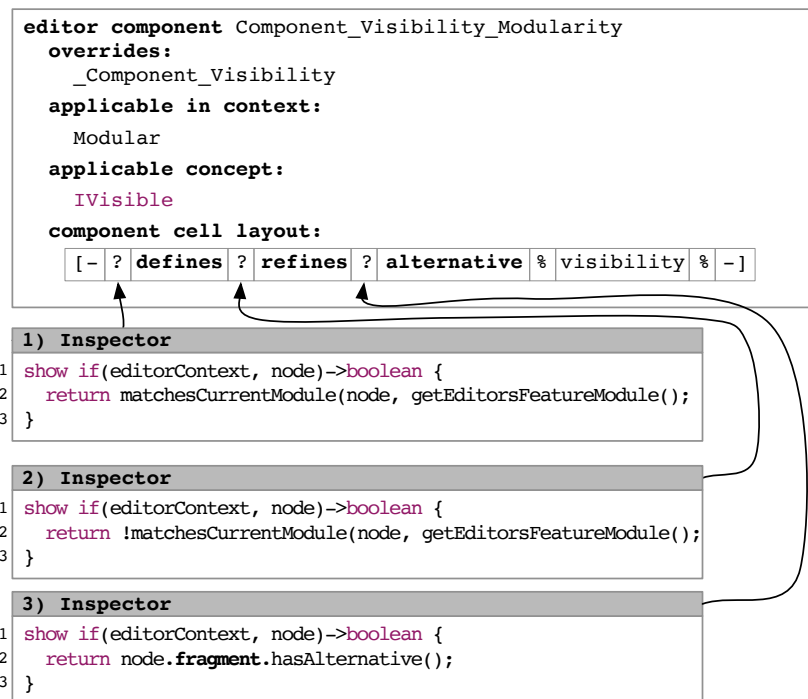
##### 8.1.4.1 *Editor I: Conditionally Rendering Classifier Members*

A fragment-aware `ClassifierMember` editor component renders classifier member instances conditionally, as illustrated in Figure 8.6a. If the classifier member itself or one of its descendants is annotated with a fragment of the currently edited module, the classifier member is rendered (Lines 5 and 6). For instance, as illustrated in Figure 3.1d, Line 3, the class `PreloadProcessor` (a member of `DatabaseImpl`) is shown in the modular editor of *Memory\_Budget*, since a descendant `FeatureBlock` is annotated with fragment  $f_2$  of *Memory\_Budget*.

Moreover, an empty line ratio defines whether an empty line, which is actually also realized as a classifier member instance, is



(a) An editor filtering classifier members for rendering feature modules



(b) An editor rendering defines, refines and alternative keywords

Figure 8.6: Enabling modular renderings for Java

rendered or hidden (Line 7). In fact, for clarity reasons, we do not render more than two empty lines between methods and fields.

#### 8.1.4.2 Editor II: Rendering Defines, Refines and Alternative Keywords

To understand in a program's modular implementation whether classes and members are introductions, refinements or alternatives,

we render the keywords *defines*, *refines*, and *alternative* into the concrete syntax. On this purpose, we override the IVisible editor component, originally rendering only the member’s visibility (e.g., *public* and *private* keywords). In particular, the *defines*-keyword is shown if the node’s (e.g., method declaration’s or class’) module matches the currently edited module (cf. first inspector in Fig. 8.6b, Line 2). The *refines*-keyword is shown if they do not match (cf. second inspector, Line 2). The *alternative*-keyword is shown if the node has a fragment, which is alternative to another fragment or fragments—that is, the fragments share a VP.

Figure 3.1 gives an example. The *Memory\_Budget* feature module only refines the class *DatabaseImpl* (i.e., Fig. 3.1f  $f_0$  is associated with *Base*), and thus the *refines* keyword is shown (Fig. 3.1d, Line 2).

#### 8.1.4.3 Editor III: Rendering and Editing Method Bodies

To show a method declaration’s body in a modular fashion, we conditionally render statements and the original-keyword. The latter can be edited as well (e.g., moved to a new position). In the following, we describe the key ideas involved in this process: (i) reordering the AST according to a given refinement hierarchy, (ii) conditionally rendering statements, and (iii) rendering and moving the original-keyword.

**AST REORDERING BASED ON THE REFINEMENT HIERARCHY.** Recap that in classical modular approaches, a refinement hierarchy denotes the ordering in which the individual modules, and thus the individual method declarations, are to be composed (cf. Section 2.4.3). In fact, during variant derivation, classical modular approaches replace each occurrence of an original-keyword (or a similar keyword referring to the original implementation) with a method call to the refined method declaration (cf. Fig. 2.9 and Fig. 2.10). Thus, a different ordering of feature modules leads to the generation of semantically different variants that potentially contain multiple nested method calls.

In PEoPL, we use feature blocks instead of nested method calls. In fact, instead of spreading the implementation across different method declarations, we persist all variability internally—that is, in the variational AST—in a single method declaration. Thereby, feature blocks can be ordered in correspondence to the refinement hierarchy (if desired by the developer). Figure 8.7 shows an ordered method declaration’s example AST (cf. Fig. 2.9 and Fig. 2.10).

An algorithm to reorder a method declaration is simple. We start by moving all feature blocks and wrappers that refine the method into a so-called *reordering table* (in memory). Thereby, we record the relative AST position of each statement to the initial feature block (i.e., the one that introduces the method’s behavior). In particular, we start at the initial feature block and add all previous siblings and all next

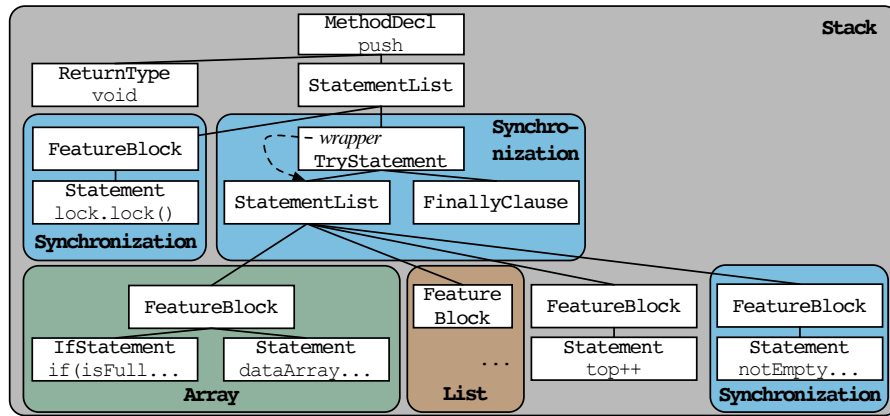


Figure 8.7: AST ordering based on a refinement hierarchy. Ordering of feature modules: *Stack*, *List*, *Array*, *Synchronization*

Feature module	Fragment (incl. subtrees)	Relative position
<i>List</i>	FeatureBlock	previous
<i>Array</i>	FeatureBlock	previous
<i>Synchronization</i>	FeatureBlock (notEmpty)	next
<i>Synchronization</i>	TryStatement	parent
<i>Synchronization</i>	FeatureBlock (lock)	previous

Table 8.1: Reordering table for the variational AST depicted in Figure 8.7

siblings to the table. If the initial feature block is wrapped, we add its wrapper (i.e., its parent) to the table. Thereby, the initial feature block replaces the wrapper, and thus remains in the method declaration. We continue by adding previous siblings, next siblings, and wrappers to the table, until we reach the method declaration’s statement list. Then, the method only contains the initial feature block. Using this approach, the ordering table is sorted and the relative positions of all statements are maintained (e.g., siblings of parents). Reordering table 8.1 gives an example for the variational AST depicted in Figure 8.7.

To reorder the AST, we add the statements from the table back into the method declaration’s statement list considering the new refinement hierarchy and the relative positions. In particular, feature blocks with the *previous* flag are added at the beginning of the method’s statement list, feature blocks with the *next* flag are added at the end of the method’s statement list. Statements with the *parent* flag wrap any statements introduced thus far—that is, all statements in the method’s statement list. Subsequently, the method’s statement list only contains this wrapper. Any other statements are added as siblings of the wrapper (and not the initial feature block) and so on. This way, multiple nestings through wrappers are possible.

Figure 8.8 gives a simple step-by-step example ordering the feature modules stored in reordering table 8.1 as follows: *Stack*, *Array*,

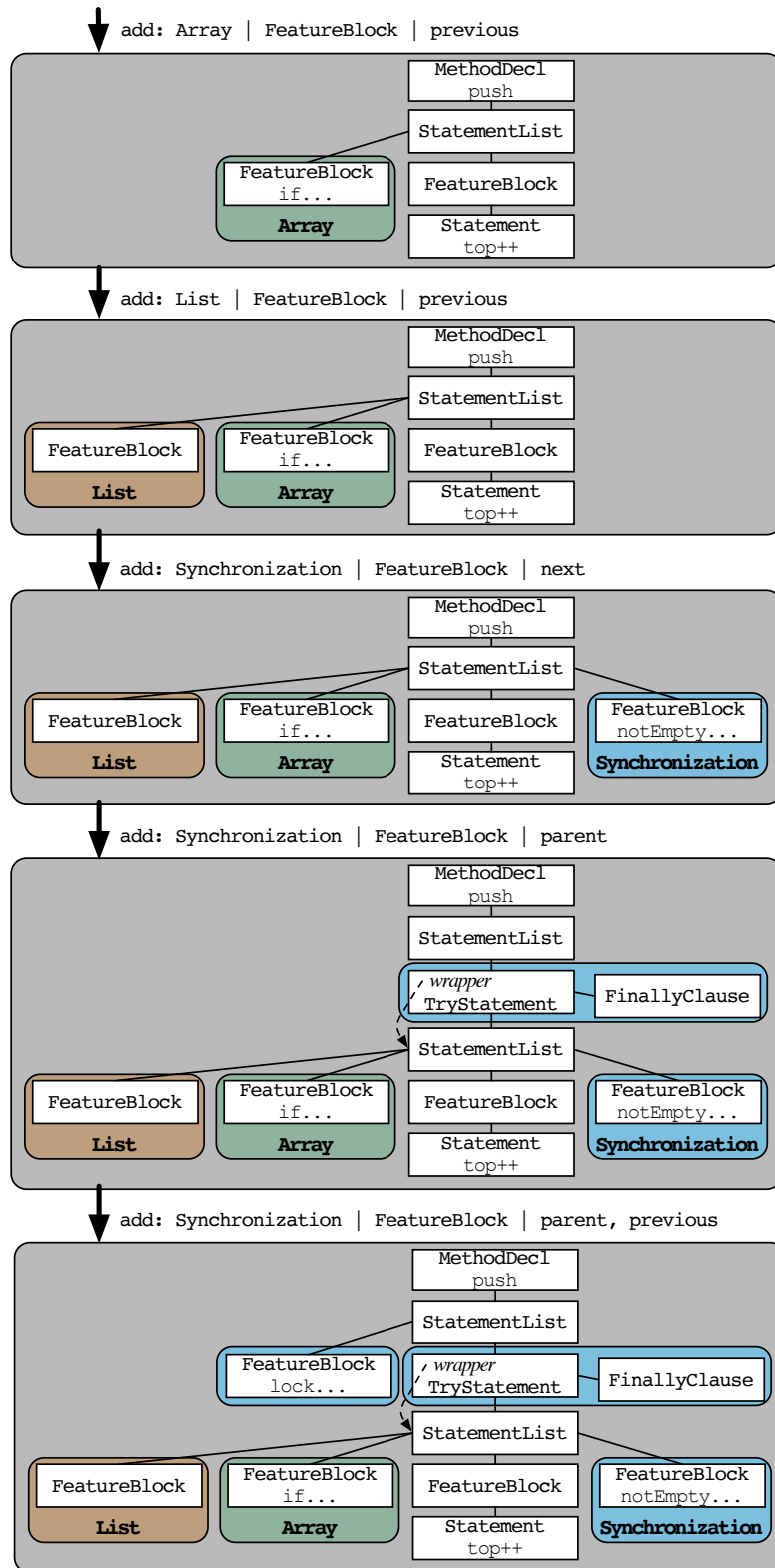


Figure 8.8: AST reordering based on reordering table 8.1. Ordering of feature modules: *Stack*, *Array*, *List*, *Synchronization*



*List*, *Synchronization*. According to this ordering, we start with the feature block belonging to the feature module *Array*, since it is the first feature to refine the codebase introduced by the feature *Stack*. The block’s relative position is *previous* according to the table, and thus we add it as the method declaration’s first statement. Next, we add the `FeatureBlock` belonging to the feature *List* as the method’s first statement, since the block’s relative position is also *previous*. Subsequently, we add the statements belonging to the feature module *Synchronization*. We start with the feature block containing the `notEmpty` statement, since it is the first statement in the table belonging to *Synchronization*. We add the block as the method’s last statement, since the block’s relative position is *next*. The `TryStatement` is the next statement in the ordering table that is assigned to the feature module *Synchronization*. The `TryStatement` serves as a *parent*, and thus we add it as a wrapper of the base code and any other statements added thus far. Finally, due to its *previous* flag, the last feature block belonging to *Synchronization*—that is, the one with the `lock` statement—is added as the first element of the method declaration.

We now discuss another simple step-by-step example. Using the final variational AST depicted in Figure 8.8 (i.e., the bottom AST), we create the reordering table 8.2. Then, we reorder the feature modules as follows: *Stack*, *Synchronization*, *Array*, *List*. Figure 8.9 shows each step. We start by adding *Synchronization* statements as discussed above. Then, we proceed with the feature blocks belonging to *Array* and *List*. Their relative position is *previous*, and thus we add them as the first statement of the method’s statement list. Using our ordering algorithm, we can reorder the AST depicted in Figure 8.9 back into the tree depicted Figure 8.8.

Notice that our reordering algorithm is limited to statement-level variability. That is, fine-grained extensions such as nested feature blocks, refinements in the middle of the method, or parameter variability cannot be reordered. However, neither can these extensions be implemented in classical modular approaches without workarounds such as hook methods. Employing workarounds would allow for reordering to a certain degree. Yet, we argue that in these cases other

Feature module	Fragment (incl. subtrees)	Relative position
<i>Array</i>	<code>FeatureBlock</code>	<i>previous</i>
<i>List</i>	<code>FeatureBlock</code>	<i>previous</i>
<i>Synchronization</i>	<code>FeatureBlock (notEmpty)</code>	<i>next</i>
<i>Synchronization</i>	<code>TryStatement</code>	<i>parent</i>
<i>Synchronization</i>	<code>FeatureBlock (lock)</code>	<i>previous</i>

Table 8.2: Reordering table for the final variational AST depicted in Figure 8.8

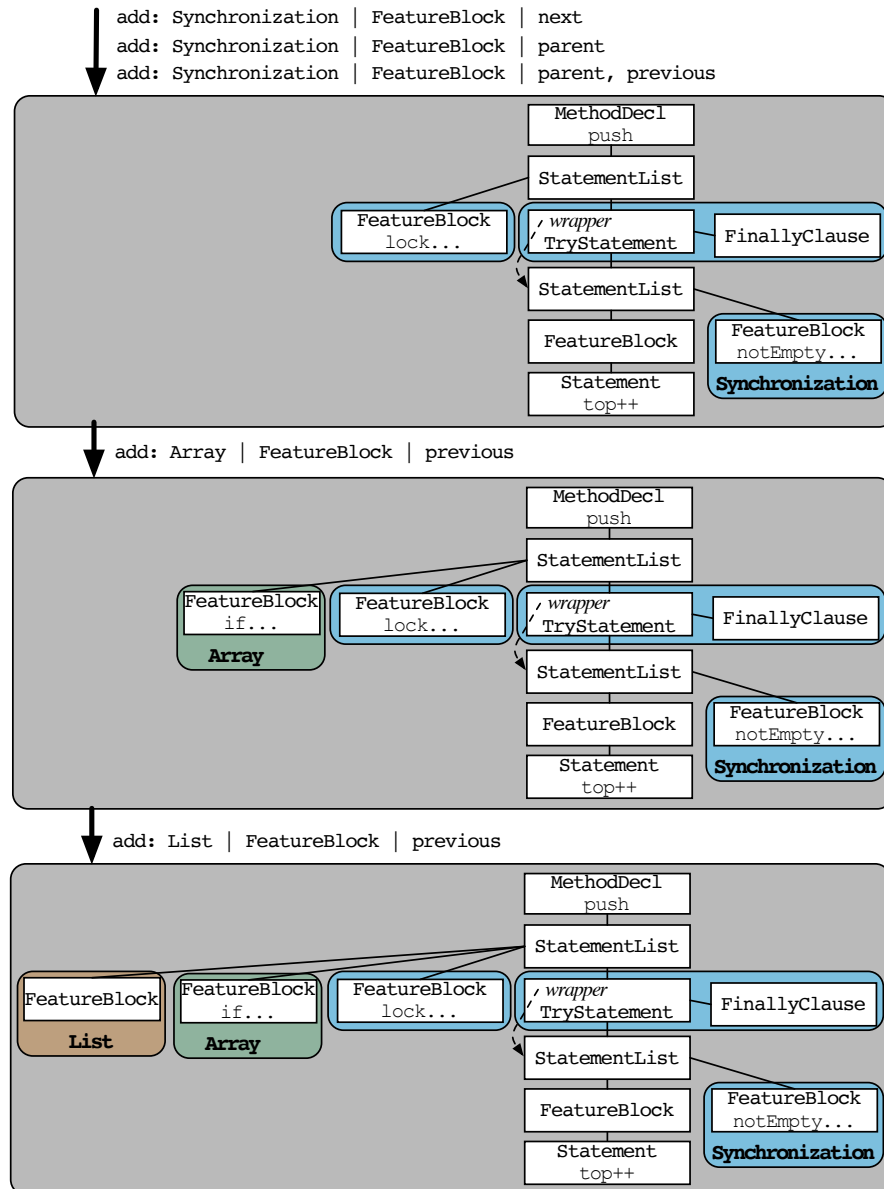


Figure 8.9: AST reordering based on reordering table 8.2. Ordering of feature modules: *Stack*, *Synchronization*, *Array*, *List*

external representations, such as our fade-in modules or embedding annotations into modules, are better suited to maintain the program’s readability. This way, developers can edit fine-grained code and expression-level variability (e.g., refinement in an if-statement’s condition) from a non-classical modular representation without being distracted by irrelevant code of other features.

Moreover, notice that custom orderings—for instance, declared in annotative external representations—that do not reflect the refinement hierarchy should typically not be reordered to maintain the correct program behavior intended by the developer. In fact, such methods should not be editable from a classical modular projection.



tional ASTs depicted in Figure 8.7 and Figure 8.8 leads to the same concrete syntax and behavior of the individual feature modules.

Wrappers belonging to a module are simply rendered. In case of wrappers not belonging to the module, we omit rendering the wrapper, while the wrappee’s statement list provides the next level of investigation for FeatureBlocks and wrappers. As a result, only statements of the current feature module are shown in a classical modular representation. For example, in Figure 8.10, we render the TryStatement and its finally-clause, since they belong to the projected feature module. Moreover, investigating the TryStatement’s statement list, we also render the feature block containing the `notEmpty.signalAll()` statement. Notice that the feature block including the increment of the variable `top` belongs to the codebase. We handle codebase feature blocks differently as we will discuss next.

RENDERING THE ORIGINAL-KEYWORD WITH EDITING SUPPORT. In our standard, classical modular editor, we also need to show an original-keyword for refined method declarations. In a nutshell, PEOPL simply renders the codebase’s FeatureBlock as an original-keyword (matching the method’s signature). This rendering is straightforward, since PEOPL uses feature blocks to persist all variability in a single method declaration. Figure 8.10 shows an example rendering. Figure 3.1d (Line 5) gives another example (cf.  $f_1$  in Fig. 3.1f). So, in contrast to classical modular approaches—where variability is persisted across different, real method declarations—PEOPL’s original-keyword is not a real method call. Consequently, the keyword is currently restricted to the statement level, which however sufficed for our case studies.

To support the original-keyword on the expression level (i.e., as a real method call), we could implement further (but likely more complex) on-the-fly tree transformations and rendering rules. In fact, we would simply create a new method declaration and refer to it—as in classical modular approaches. Yet, we argue that to implement fine-grained changes—beyond statement level—developers should rather switch to fade-in feature modules or embed annotations, since they are more concise and also allow custom orderings.

Aside from just rendering the original-keyword, PEOPL also supports editing it. In fact, PEOPL supports a full modular editing experience on the statement-level. Developers simply type *original* at the desired program position, which automatically transforms the AST, with the result that the original-keyword is rendered at the desired position. The corresponding algorithm is simple, but requires an ordered variational AST, since maintaining the relative position of all other feature blocks and wrappers to the codebase is crucial. That is, we only want to change the behavior of the current feature module and not the behavior of the others. All in all, we distinguish three different editing scenarios, which we discuss in detail next.

1. *Move the original-keyword within the same statement list*

Moving the original-keyword within a statement list is straightforward. Figure 8.11 gives an example. The developer simply moves the cursor to the target program position (e.g., by creating an empty statement above the comment). Then, she starts typing *original*, which eventually triggers an automatic tree transformation algorithm:

- (i) Split the target feature block at the target position (i.e., at an empty statement) into left (previous) feature block and right (next) feature block, and remove the empty statement where the developer typed *original*.
- (ii) Move the left block to the beginning and the right block to the end of the current statement list (to maintain the relative positions of any external module's feature block to the codebase).
- (iii) Reorder the AST according to the refinement hierarchy (to also maintain the correct program behavior).

Notice that reordering removes empty feature blocks and merges feature blocks whenever possible. For instance, if the codebase's feature block has two next-sibling blocks belonging to the same feature, we simply merge these blocks. All in all, the algorithm produces correct results in any statement list—that is, the statement list of the current method or wrapper. Moreover, the original-keyword always occupies the position where the developer typed, since we simply render the codebase's feature block.

2. *Move the original-keyword into a wrapper*

Developers can also move the original-keyword into statements wrapping other statements (i.e., wrappers). Figure 8.12 gives an example of typing *original* within a `try`-statement, which eventually initiates a simple tree transformation algorithm:

- (i) Move the target feature block containing the wrapper to be transformed (i.e., the one containing the empty target statement) to the codebase's feature block or the wrapper that contains the codebase (i.e., the codebase could be wrapped by the current or any other feature module internally). As a result, the target feature block is the next sibling of the codebase (or its wrapper). Notice that by moving only the feature block belonging to the current module, we maintain the relative positions to the codebase of any other feature block and wrapper.
- (ii) Split the target feature block into left feature block, target wrapper, and right feature block. This way, we separate the target wrapper from its previous and next statements.

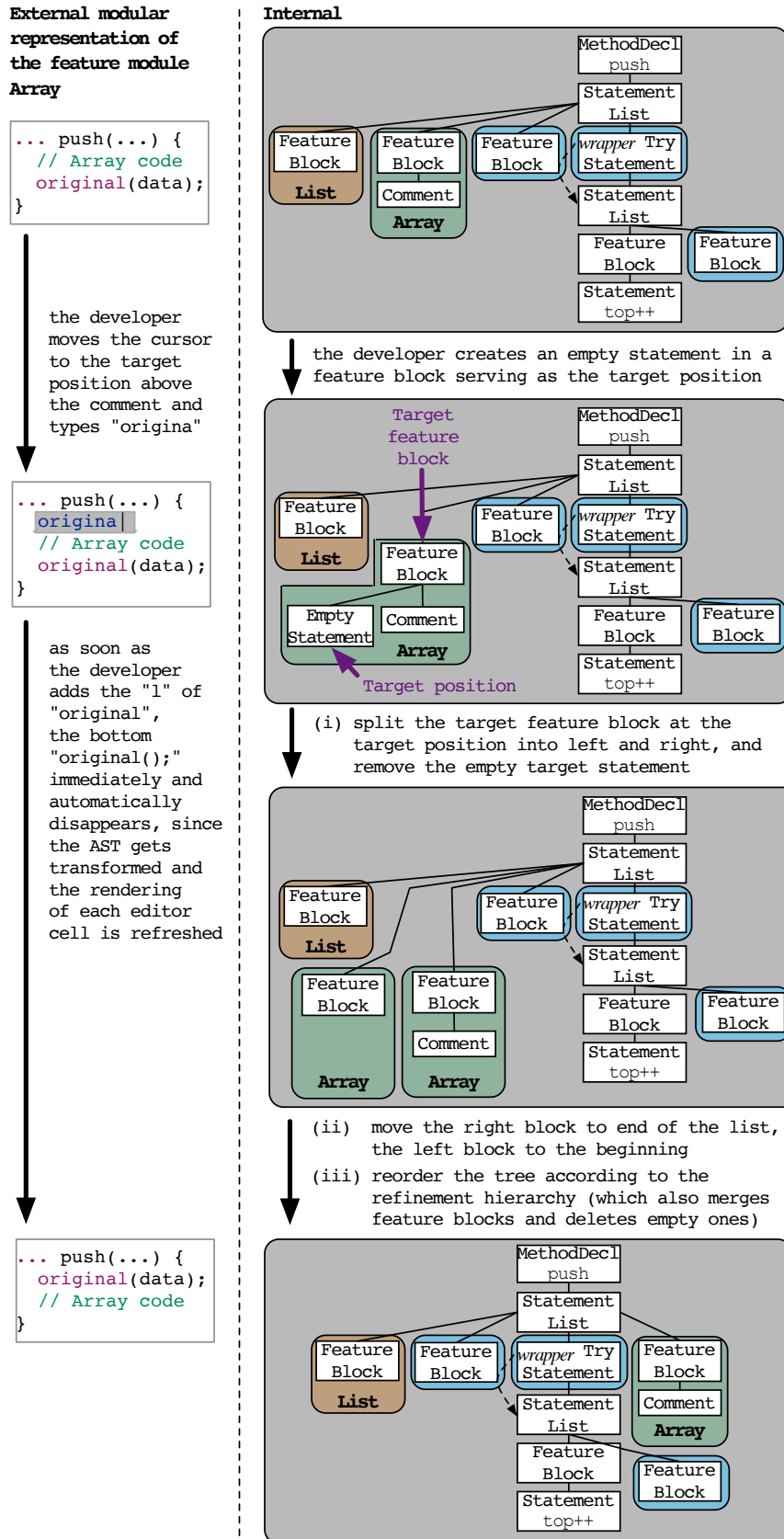


Figure 8.11: Moving the original-keyword within a statement list and re-ordering the tree

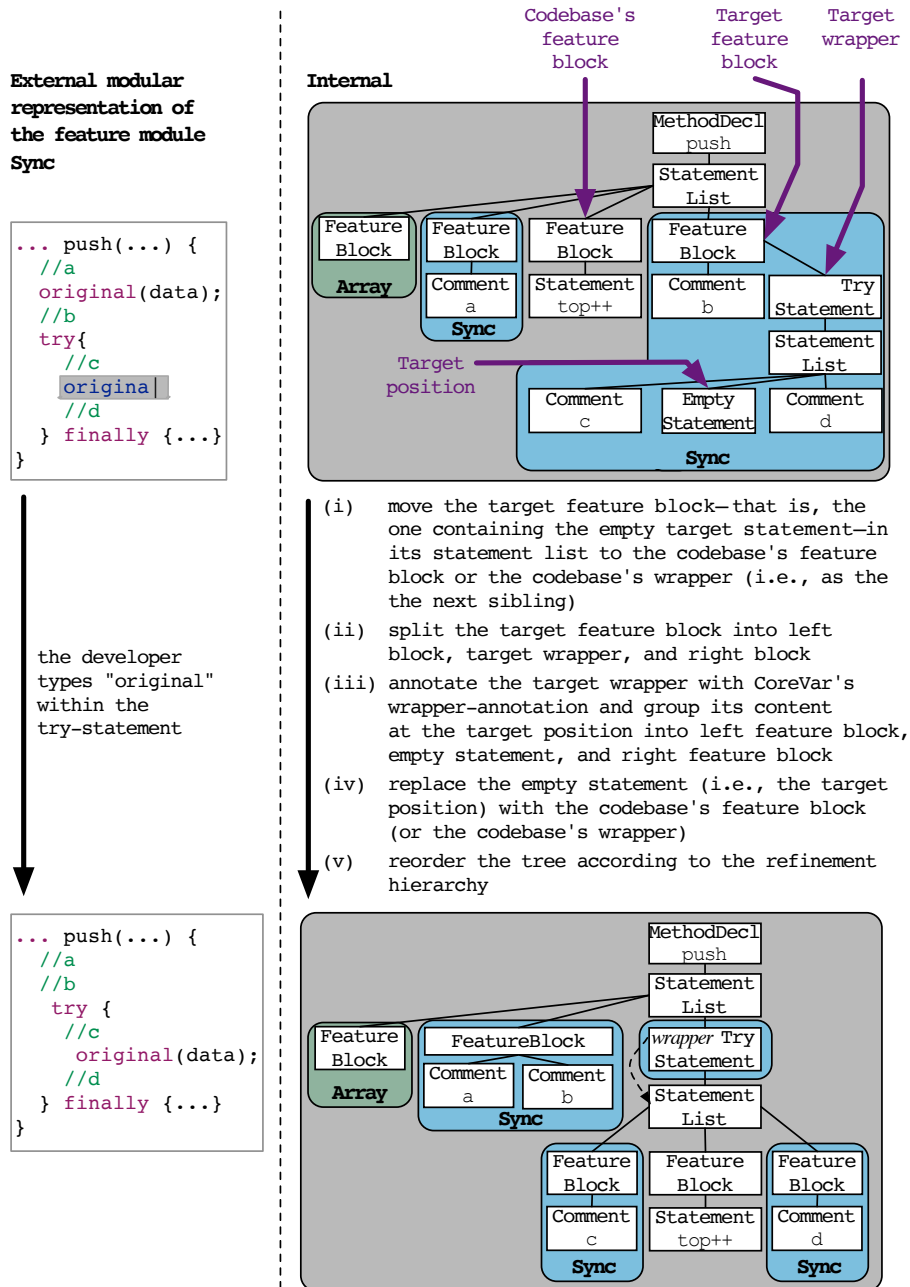


Figure 8.12: Moving the original-keyword into a wrapper and reordering the tree

(iii) Annotate the target wrapper (or target wrappers) with CoreVar's wrapper-annotation. Then, group the content of the target wrapper in a feature block at the target position into left feature block, empty statement (i.e., the target position is not grouped by a feature block), and right feature block. Notice that if wrappers are nested (e.g., we move the *original* keyword into the true parts of multiple if-statements), we split into left feature block, nested wrapper, and right feature block for each wrapper. As a

result, we separate the target position from its previous and next statements (and allow nested wrappers).

- (iv) Replace the empty statement with the codebase's feature block (or the codebase's wrapper). Notice that this strategy maintains the relative positions of any other feature block and wrapper to the codebase, since we just move the codebase into the next sibling.
  - (v) Reorder the tree according to the refinement hierarchy to enforce correct behavior. Recap that this process removes empty feature blocks (i.e., the ones without statements) and merges direct feature block siblings belonging to the same feature module (e.g., the feature blocks in Fig. 8.12 that contain comment a and b before reordering).
3. *Move the original-keyword outside a wrapper*

Moving the *original-keyword* outside a wrapper is also straightforward. Basically, it is the inverse process to moving the keyword into a wrapper. Figure 8.13 gives an example of the tree transformation algorithm:

- (i) Move the target feature block containing the empty target statement to the affected wrapper (i.e., the wrapper that is to be transformed). As a result, depending on the target feature block's relative position, it is now the previous or next sibling of the affected wrapper. Thereby, we maintain relative positions of all wrappers and blocks in the AST.
- (ii) Replace the empty statement with the codebase's feature block (or the codebase's wrappers introduced by other features), and thus maintain relative positions.
- (iii) Remove the wrapper-annotation of the affected wrapper, since it does not wrap the codebase anymore. Moreover, add the affected wrapper to a new top level feature block belonging to the feature.
- (iv) Remove any feature block within the affected wrapper, since we only need the previously added top level feature block. That is, any statement must be contained by a top level feature block, and thus other blocks belonging to the same feature are irrelevant.
- (v) Reorder the tree according to the refinement hierarchy to enforce correct behavior.

On a final note, using these algorithms sufficed for our case studies. Yet, the approach is limited to the statement level, since the *original-keyword* is not a true method call, but a simple rendering of the codebase's feature block. Thus, the keyword cannot be used multiple times to call the original codebase or used within an expression (e.g., in the control flow). Developers can, however, still add



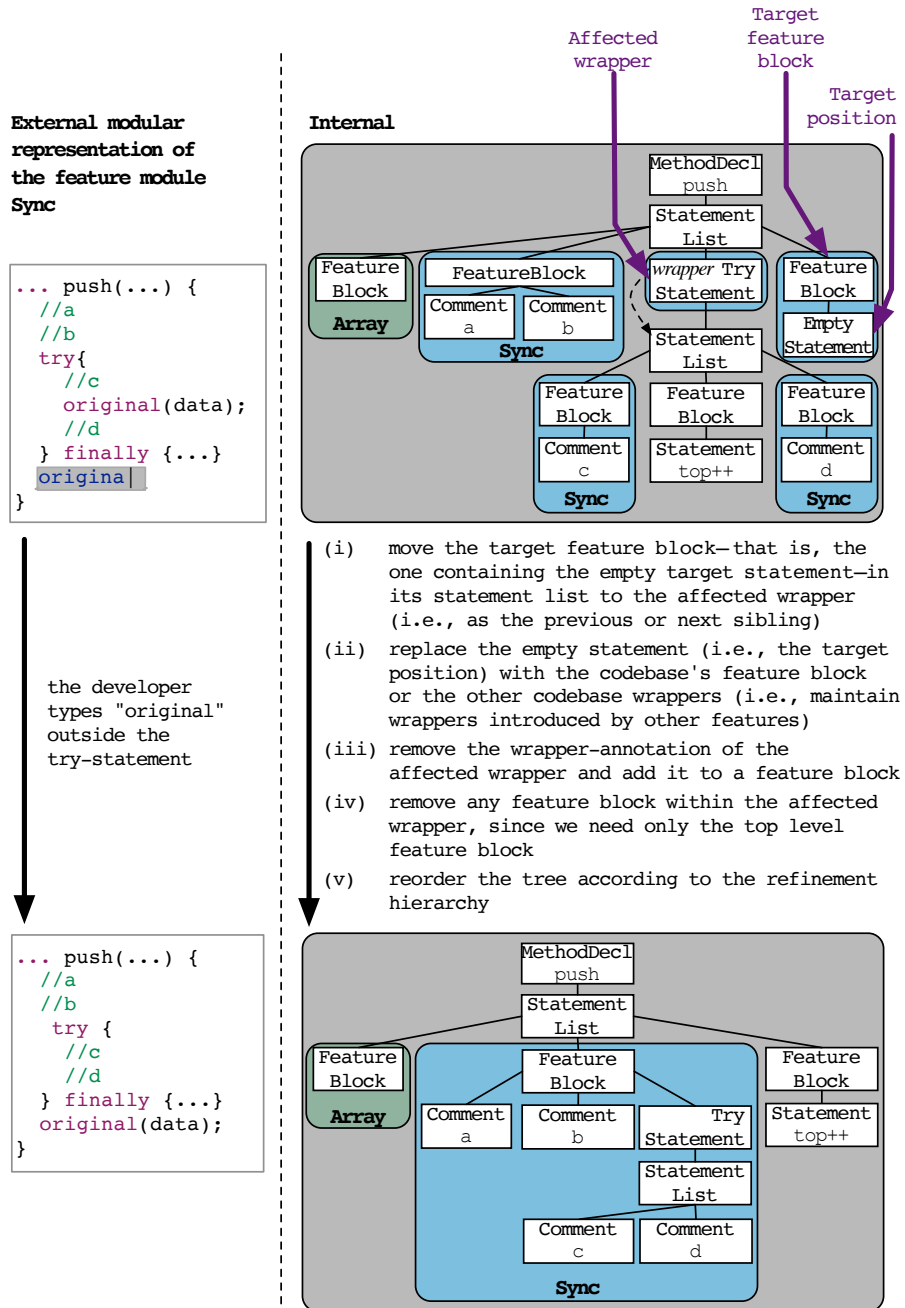


Figure 8.13: Moving the original-keyword outside a wrapper and reordering the tree

such methods manually, refine them and use them as desired. Moreover, we could implement more advanced automatic tree transformations to reflect such scenarios, which is—together with corresponding advanced annotative renderings that hide the automatically created method declarations—subject of our future work.

### 8.1.5 *Rendering Fade-in Feature Modules*

Our fade-in feature module projection is similar to the standard feature module projection. We use different keywords, but the major difference is that we define so-called *surrounding concepts*. For Java, surrounding concepts are classes, interfaces, classifier members, and statements. So if, for instance, an expression is annotated with a fragment, we also render the surrounding concept's instance (in grey)—that is, the statement instance which holds an annotated expression (cf. Fig. 3.4b). Furthermore there are colored method parameters. The projection renders the classifier member holding the parameter (in grey) as well (cf. Fig. 3.4b). External feature blocks (i.e., the ones whose fragment relates to a different module) are collapsed into a keyword in the concrete syntax.

### 8.1.6 *Blending Renderings*

Blended projections simply reuse editor components defined by the projections involved. For instance, to blend annotations into modules, elements of the annotative and modular fragment editor are reused as well as horizontal and vertical bars.

## 8.2 MODELING FACILITIES

Recap that variability can be modeled using a simple and an advanced SPL declaration (cf. Sec. 4.1). We now discuss how the two ways for modeling SPLs are realized. Moreover, we introduce an *expert product line configuration* which enables developers to use CoreVar's module operations (addition, subtraction, and overriding) to configure an SPL (cf. CoreVar's variant derivation facilities in Sec. 7.3). We start with the expert product line configuration, since simple and advanced SPL declarations automatically transform feature (module) selections into such a configuration.

### 8.2.1 *Expert Product Line Configuration*

PEoPL's internal representation enables composing variants to use algebraic expressions over feature modules (cf. Sec. 7.3). Three operators are available in such expressions: addition (+), subtraction (−), and overriding (→). We conceive language concepts reflecting the behavior of these operators in CoreVar (e.g., an `OverridingOperator` language concept). Expert developers can use the corresponding concept instances to construct variants as illustrated in Figure 8.14 (cf. our concrete Berkeley DB calculations in Table 7.1).

Figure 8.15a shows the `ExpertConfiguration` root concept, which provides the entry point for constructing variants from algebraic ex-

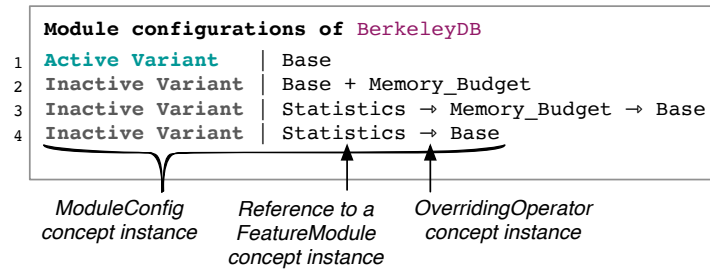


Figure 8.14: Expert configuration (i.e., ExpertConfiguration instance) for the Berkeley DB example (cf. Fig. 3.1)

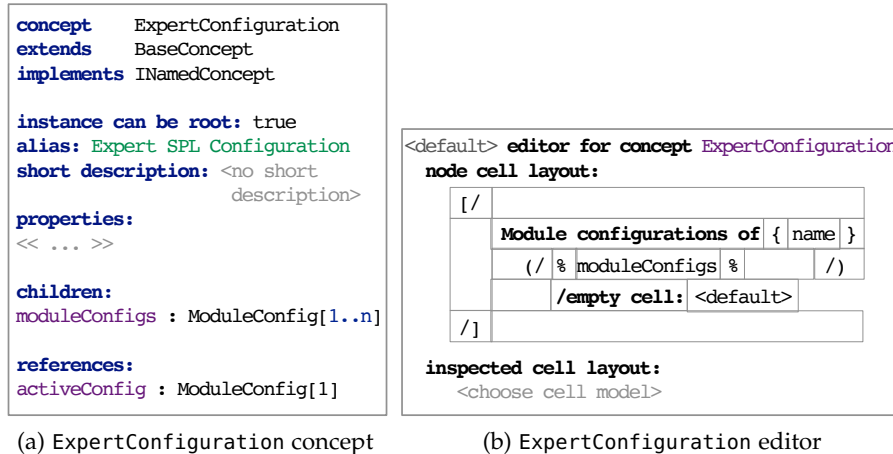


Figure 8.15: ExpertConfiguration concept and editor, which expert developers use to configure SPL variants

pressions over feature modules. An ExpertConfiguration instance holds a set of module configurations, while pointing to a configuration representing the currently active variant (cf. Fig. 8.14, Line 1, and the activeConfig reference in Fig. 8.15a). The ExpertConfiguration concept editor (Fig. 8.15b) renders ExpertConfiguration concept instances as illustrated in Figure 8.14. For each concrete module configuration, the editor embeds the editor of the ModuleConfig concept, which enables developers to type algebraic expressions over feature modules (Fig. 8.14, right half of Lines 1–4).

Notice that, in an expression over feature modules, developers can only select the modules, which have been declared in a simple product line declaration—that is, the concept instance accommodating feature module instances (discussed next).

### 8.2.2 Simple Product Line Declaration and Feature Module Selection

The easiest way to declare feature modules and constraints over feature modules are simple product line declarations (cf. Sec. 4.1). To derive a variant, developers select a set of features. We discuss the

concepts and editors for declaring product lines and selecting modules next.

**DECLARATION.** We provide a simple product line declaration as shown in Figure 3.1b by implementing a `SimpleDeclaration` language concept and a corresponding editor (together with simple checking rules ensuring a satisfiable feature selection).

A `SimpleDeclaration` instance holds a set of feature modules and constraints (cf. Fig. 8.16a). Notice that for constraints, we also provide language concepts enabling the construction of propositional formulas (e.g., the `ImpliesOperator` language concept).

Figure 8.16b shows the `SimpleDeclaration` concept editor, which enables manipulating `SimpleDeclaration` root instances in the AST. The editor renders a module-specific constraints section and a module declarations section by embedding the target editor of each concrete constraint and feature module. Figure 8.17 shows the editor of the `FeatureModule` concept. It renders a feature module's name with a vertical, colored bar, using the module's color (cf. Fig. 3.1b), and, if existent, the dependencies to other feature modules (i.e., the artifact-related feature dependencies).

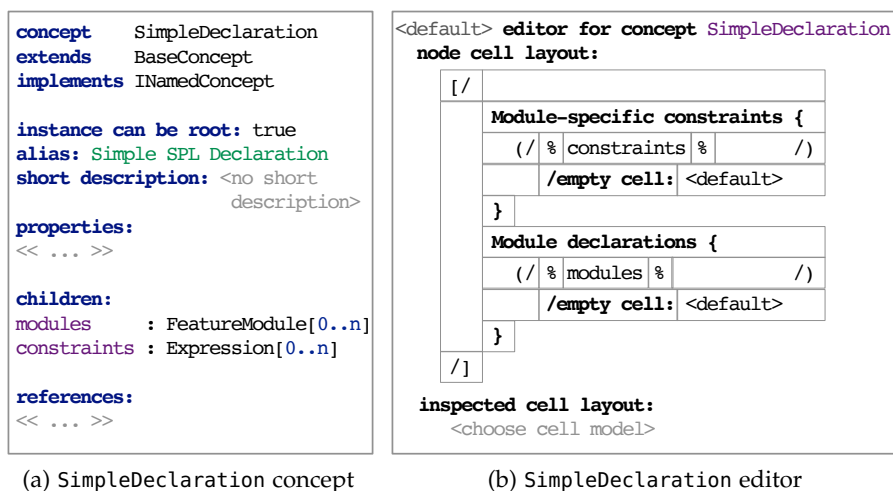


Figure 8.16: `SimpleDeclaration` concept and editor, which developers use for simple SPL declarations

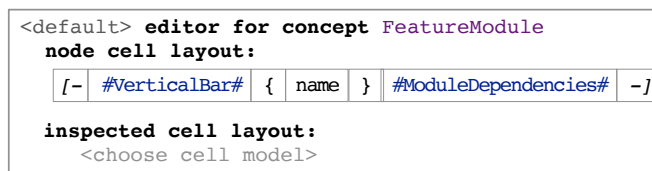


Figure 8.17: Editor rendering modules using its color, name, and dependencies

FEATURE MODULE SELECTION. For configuring variants without having expert knowledge of CoreVar’s algebraic operations, developers can use the SimpleVariantConfiguration language concept, which enables straightforward feature module selections. Figure 8.18a shows the language concept, Figure 8.18b the concept editor, and Figure 8.18c gives a simple example.

Notice that developers must refer to an ExpertConfiguration (cf. Fig. 8.18a, variantConfigs, and Fig. 8.18c, Line 1). Based on a feature module selection, an algorithm provided with CoreVar automatically constructs a corresponding algebraic expressions over feature modules in the expert configuration. In fact, each module selection directly refers to a corresponding expression. During expression construction, the algorithm takes the declaration ordering of feature modules into account. Thereby, the algorithm uses the overriding operator to apply feature modules in the given ordering starting with the

```

concept SimpleVariantConfiguration extends BaseConcept
                                implements INamedConcept

instance can be root: true
alias: Simple Variant Configuration
short description: <no short description>
properties: << ... >>

children:
moduleSelection : ModuleSelection[0..n]

references:
variantConfigs : ExpertConfiguration[1]

```

(a) SimpleVariantConfiguration concept

```

<default> editor for concept SimpleVariantConfiguration
node cell layout:
[ /
  Simple variant configuration { name } {
    Variant configurations: ( % variantConfigs % -> * R/O model access * )
    Variants {
      ( / % moduleSelections % /empty cell: <default> / )
    }
  }
 / ]
inspected cell layout:
<choose cell model>

```

(b) SimpleVariantConfiguration editor

```

Simple variant configuration BerkeleyDB
1  Variant configurations: BerkeleyDB
2  Variants {
3    Active Variant | Base
4    Inactive Variant | Base, Memory_Budget, Statistics
5    Inactive Variant | Base, Statistics
6  }

```

(c) SimpleVariantConfiguration example

Figure 8.18: SimpleVariantConfiguration concept and editor, which developers use for feature module selections (of variants)

highest priority (i.e., from bottom to top). For instance, the feature module declaration ordering in Figure 3.1b is used by the algorithm to construct the algebraic expression over feature modules in Line 4 of Fig. 8.14 from the module selection in Line 5 of Fig. 8.18c. As a result, the configuration in Line 5 of Figure 8.18c directly refers to the expression in Line 4 of Figure 8.14. Note that if the selection changes, the expression changes as well.

### 8.2.3 Advanced Product Line Declaration

The advanced SPL declaration increases the developer’s variability modeling flexibility in comparison to using a simple declaration (cf. Sec. 4.1.2). It adds an additional layer of abstraction: features that are mapped to feature modules. Altogether, an advanced SPL declaration enables developers to declare features, feature modules, constraints, mappings, and variants.

To enable developers to model variability in an advanced fashion as illustrated in Figure 4.2, we conceive an `AdvancedDeclaration` language concept and editor. Figure 8.19 shows the details. Actually, developers must select concept instances of `SimpleDeclaration` and `ExpertConfiguration` (cf. references in Fig. 8.19a, and Fig. 8.19b, Lines 2–3). Notice that in our advanced declaration example in Figure 4.2, we omit these references for clarity reasons. In our actual implementation, the references are used by `CoreVar` algorithms to automatically

1. create `FeatureModule` concept instances in the selected, referenced `SimpleDeclaration` instance (Fig. 8.19b, Line 7), and
2. create algebraic expressions over feature modules from feature selections in the `ExpertConfiguration` instance.

So a selection of features basically leads to a selection of feature modules (in the given ordering of the mappings section), which in turn is transformed into an algebraic expression over feature modules. Realizing removal in this process is straightforward. First, we construct a chain of feature modules  $m_1 \rightarrow m_2 \rightarrow \dots \rightarrow m_n$ . In a second step, we apply the feature modules that remove the feature details:  $(m_1 \rightarrow m_2 \rightarrow \dots \rightarrow m_i) - m_{i+1} - m_{i+2} - \dots - m_n$ .

## 8.3 CONCLUDING REMARKS

Using projectional editors to create PEOPL’s external representations is powerful, flexible, and enables a rapid prototyping. New variability implementation and variability modeling facilities can easily be added, extended, and exchanged. We conceived PEOPL’s modeling facilities in language-independent fashion, and thus developers can use

```

concept AdvancedDeclaration extends BaseConcept
                                implements INamedConcept

instance can be root: true
alias: Advanced SPL Declaration
short description: <no short description>
properties: << ... >>

children:
features      : Feature[1..n]
constraints   : Expression[0..n]
mappings      : Mappings[0..n]
featureSelections : FeatureSelection[0..n]

references:
moduleDeclarations : SimpleDeclaration[1]
variantConfigurations : ExpertConfiguration[1]

```

(a) AdvancedDeclaration concept

```

<default> editor for concept AdvancedDeclaration
node cell layout:
1  [ /
2  SPL { name } {
3  Module declarations: ( % moduleDeclarations % -> * R/O model access * )
4  Variant configurations: ( % variantConfigurations % -> * R/O model access * )
5
6  Features { ( - % features % /empty cell: <default> - ) }
7  Modules { ( - % moduleDeclarations % -> ( / % modules % /empty cell: <default> / ) }
8  Constraints { ( / % constraints % /empty cell: <default> / ) }
9  Mappings {
10 ( / % mappings % /empty cell: <default> / )
11 }
12 Variants {
13 ( / % featureSelections % /empty cell: <default> / )
14 }
15 }
16 [ /

inspected cell layout:
<choose cell model>

```

(b) AdvancedDeclaration editor

Figure 8.19: AdvancedDeclaration concept and editor, which developers use for advanced SPL declarations

them for arbitrary target languages. PEOPL's variability representation editors are mainly black-box generic (annotative, variant, reuse). Modular editors are language-dependent (for now), but CoreVar provides the infrastructure to easily implement the required editors in target-language-specific CoreVar extensions (e.g., JavaVar).



Engineering software product lines includes modeling and implementing variability. A common problem is the gap between the two. Artifact-related feature dependencies need to be lifted up into the variability model using constraints. These must be maintained to ensure SPL correctness—a cause of subtle errors, since changes of the implementation require a developer's awareness of corresponding constraints [137, 152]. From the SPL developer's perspective, PEOPL provides two facilities to mitigate this issue (as presented in Sec. 4.2): feature module constraints extraction (Sec. 9.1) and variant-aware data-flow analysis (Sec. 9.2). We now discuss how the two are realized.

### 9.1 ARTIFACT-RELATED FEATURE DEPENDENCY ANALYSIS

Next, we realize our dependency checker, which automatically extracts artifact-related feature dependencies as introduced in Section 4.2.1. PEOPL's feature model depicted in Figure 6.3 shows that CoreVar contributes a basic, generic dependency extraction infrastructure (based on its internals). On top, language engineers realize a concrete extraction for a given target language (e.g., JavaVar implements the dependency extraction for Java). We now discuss the concrete realizations in CoreVar and JavaVar.

#### 9.1.1 CoreVar: Dependency Extraction Infrastructure

CoreVar's internal representation enables a straightforward extraction of dependencies between feature modules. Recap that according to our internal representation any AST node is contained by at least one feature module—that is, the top-level feature module annotating the root node with a fragment. The basic idea of the extraction algorithm is to check whether the containing feature module of a referring node differs from the containing module of the corresponding declaring node. For instance, imagine that a method call contained by a feature module *A* refers to a method declaration contained by a module *B*. Then, *A* depends on *B*, since a dangling reference would appear if we only select *A*, but not *B*.

Figure 9.1 shows the fundamental extraction algorithm implementing these ideas. The algorithm's entry point is the `checkDependency` method returning a so-called *dependency message*, which can be used as a type system message (Line 2). If there is no dependency, the algo-

```

1 public class DepChecker {
2     public static DepMessage checkDependency(node<> ref, node<> decl) {
3         DepMessage depMsg = new DepMessage();
4         // we treat code external to the SPL (i.e., library calls)
5         // as mandatory code, which is always available
6         if ( !decl.containing root.isInstanceOf(ICompilationUnitContainer)
7             && decl.containing root.fragment.isNull) return null;
8
9         node<FeatureModule> refModule = getModuleOfNode(ref);
10        node<FeatureModule> declModule = getModuleOfNode(decl);
11        if (refModule.isNotNull && declModule.isNotNull) {
12            if (isContainedByModule(ref, declModule)) return null;
13            if (refModule != declModule) depMsg.refDependency(refModule, declModule);
14        }
15        return depMsg;
16    }
17
18    public static node<> getModuleOfNode(node<> node) {
19        return getNextVariationalNode(node).module;
20    }
21
22    public static node<> getNextVariationalNode(node<> node) {
23        boolean foundWrappee = false;
24        foreach varNodeCandidate in node U node.ancestors {
25            if (isWrappee(varNodeCandidate)) {
26                foundWrappee = true;
27            } else if (foundWrappee && hasWrapperAnnotation(varNodeCandidate)) {
28                // wrappers are replaced by wrappees, and, thus, do not denote
29                // the next variational node (cf. next else if, which is skipped)
30                foundWrappee = false;
31            } else if (varNodeCandidate.fragment.isNotNull) {
32                return varNodeCandidate;
33            }
34        }
35        return null;
36    }
37
38    public static boolean isContainedByModule(node<> node,
39                                             node<FeatureModule> declModule) {
40        do {
41            currentNode = getNextVariationalNode(currentNode.parent);
42            if (currentNode.module.isNotNull && currentNode.module == declModule) {
43                } while (currentNode.isNotNull);
44            return false;
45        }
46    }

```

Figure 9.1: Basic algorithm for extracting artifact-related feature dependencies

rithm returns *null*. In CoreVar tailorings, we call the method for checking the dependency between a concrete referring node and its declarations. Notice that code external to the SPL (e.g., a node referring to a declaration in an external library) is treated as mandatory, and thus there is no dependency (Lines 6–7). In fact, the algorithm checks whether the AST’s root node is an *ICompilationUnitContainer* or has a fragment annotation.

Then, the checker algorithm obtains the containing module of the reference (Line 9) and the declaration (Line 10). In particular, the algorithm calls the *getModuleOfNode* method, which in turn calls the *getNextVariationalNode* method to search for the module of the

node’s next variational ancestor (Line 19). The search also includes the given node itself (Line 24) since it could be variational as well. Moreover, notice that the search algorithm is aware of wrapper annotations. Recap that during variant derivation a wrapper can be replaced by its wrappee. So, the code within the wrappee is not dependent of the wrapper (Line 23 and Lines 25–30). In any other case, the algorithm simply returns the node that is annotated with a fragment (Lines 31–33).

The algorithm also checks whether any module of the referring node’s ancestors matches the declaration’s module (Line 12 and Lines 38–45). Figure 9.2 gives a simple example. According to our algorithm, there is no dependency from the referring node’s module (green) to the declaration node’s module (gray). The reason is that Foo’s method bar is only included in variants if the gray and green features are selected. In fact, if gray is not selected, then there is no method bar either, so the algorithm basically traverses up in the tree and compares modules (Lines 41–42).

Finally, the algorithm checks whether the referring node’s module and the declaration node’s module differ. If they differ, a dependency is found (cf. Fig. 4.4) and the dependency message is set accordingly.



```
public class Foo {
  public void bar() {
    Ele.ben();
  }
}

public class Ele {
  public static void ben() {
    // my code
  }
}
```

Figure 9.2: Feature module containment example

### 9.1.2 *JavaVar: Java-specific Dependency Extraction*

Using CoreVar’s generic dependency extraction infrastructure, we now implement a concrete dependency extraction for Java. Figure 9.3 shows the simple algorithm for running a full dependency check. It takes a Java program as input (Line 2) and maintains a list of dependency messages (Line 3). Then, the algorithm conducts a dependency check over several concrete language concept instances using the generic dependency checker (Lines 4–23). For instance, we iterate over all `VariableReference` instances in the AST (Line 16). For each variable reference, we perform a dependency check using the variable reference itself and the variable declaration it refers to (Line 17–18).

Notice that checking the dependencies of a method call requires a more detailed investigation (Lines 20–23). In fact, the Java dependency checker not only checks the dependencies of the method call and the method declaration itself, but also of every parameter involved (Lines 31–34). Using this check, we found 57 dependencies

```

1 public class JavaDepChecker {
2     public static list<DepMessage> checkDependencies(JavaProgram p) {
3         list<DepMessage> depMsgs = new list<DepMessage>;
4         foreach classCreator in p.nodes(DefaultClassCreator) {
5             depMsgs.add(DepChecker.checkDependency(classCreator,
6                                                         classCreator.classifier));
7         }
8         foreach classifierType in p.nodes(ClassifierType) {
9             depMsgs.add(DepChecker.checkDependency(classifierType,
10                                                         classifierType.classifier));
11         }
12         foreach fieldRef in p.nodes(FieldReferenceOperation) {
13             depMsgs.add(DepChecker.checkDependency(fieldRef,
14                                                         fieldRef.fieldDeclaration));
15         }
16         foreach variableRef in p.nodes(VariableReference) {
17             depMsgs.add(DepChecker.checkDependency(variableRef,
18                                                         variableRef.variableDeclaration));
19         }
20         foreach methodCall in p.nodes(MethodCall) {
21             depMsgs.unite(JavaDepChecker.checkMethod(methodCall,
22                                                         methodCall.methodDeclaration));
23         }
24         return depMsgs;
25     }
26     public static list<DepMessage> checkMethod(node<MethodCall> call,
27                                                 node<MethodDeclaration> decl) {
28         list<DepMessage> depMsgs = new list<DepMessage>;
29         depMsgs.add(DepChecker.checkDependency(call, decl));
30         // check dependencies of parameters
31         for(int i = 0; i < call.parameter.size, i++) {
32             depMsgs.add(DepChecker.checkDependency(call.parameter[i],
33                                                         decl.parameter[i]));
34         }
35         return depMsgs;
36     }
37 }

```

Figure 9.3: Algorithm for Java-specific dependency extractions

that were not declared in the feature model of the CIDE version of Berkeley DB, and thus incorrect variants could have been generated.

In our implementation, we also implemented these dependency checks using MPS' type system infrastructure. This way, dependencies can be detected at development time without requiring developers to explicitly run the dependency checker algorithm depicted in Figure 9.3.

On a final note, our MPS implementation of the dependency checker algorithm is fast, because of three main reasons. First, there is no text parsing involved to construct the AST. Instead the AST is directly constructed via editing operations. Second, references are actively maintained in the AST by MPS, so there is no need for constructing references before analyzing the AST. Third, language concept instances are cached according to their concept in MPS. Thus, getting all concept instances of a certain concept is quick.

## 9.2 VARIANT-BASED DATA-FLOW ANALYSIS

PEoPL’s variant-based data-flow analysis is currently language-dependent and must be implemented for all language concepts whose instances contribute to a program’s data flow. Based on CoreVar’s internal variability representation, we realize such an analysis for Java. Using the data-flow DSL provided with MPS [181], we implement variability-specific checking rules for 44 Java-specific language concepts. Notice that CoreVar and the strategy for our data-flow analysis is independent of an implementation technology (i.e., MPS). With our implementation, we provide evidence that implementing the analysis is practical.

We start by discussing the necessary background for understanding MPS’ data-flow builder DSL. Subsequently, we discuss how we use this DSL to construct variant-specific data-flows from a variational AST.

## 9.2.1 Background

MPS provides a simple assembly-like DSL<sup>12</sup> for building a so-called *data-flow graph* (DFG) from a program. The data-flow builder takes a language concept instance as an input to programmatically construct the DFG from the AST. Notice that each DFG node refers to its corresponding AST node to support tracing [181]. We now briefly discuss the data-flow builder language elements necessary for understanding our examples and for building a DFG:

1. `code for` calls the data-flow builder of another concept instance (i.e., AST node) by creating an edge between the two DFG nodes. This way, we can programatically denote when and where the subgraph of another language concept is embedded into the DFG. Figure 9.4a gives a simple example. A `BlockStatement` instance simply embeds the data-flow builder of its `StatementList` instance.
2. `read` denotes a read from a variable.
3. `write` indicates a write to a variable.
4. `nop` marks an empty operation used for language concepts not manipulating the data flow. Using `nop`, we maintain traceability [181].
5. `jump` enables us to unconditionally jump to another position in the DFG. In the data-flow builder, we use labels as jump targets. Then, `jump` creates an edge to the target DFG node. For instance,

<sup>1</sup> <https://confluence.jetbrains.com/display/MPSD20171/Data+flow>

<sup>2</sup> <https://confluence.jetbrains.com/display/MPSD20171/Dataflow>

```

1 data flow builder for BlockStatement {
2   (node)->void {
3     code for node.statements
4   }

```

(a) Data-flow builder for Java's BlockStatement

```

1 data flow builder for BlockStatement {
2   (node)->void {
3     if ( node.fragment.isNull
4         || VariantFragmentCache.getInstance().contains(node) ) {
5       code for node.statements
6     }
7   }
8 }

```

(b) Overridden data-flow builder for Java's BlockStatement recognizing variants

Figure 9.4: Data-flow builder for the BlockStatement concept in Java (a) and its overridden version in JavaVar (b)

Java's BreakStatement jumps after the loop or switch statement, containing the statement.

6. if jump indicates a conditional jump to another position in the DFG. For example, at the head of a while-loop, we either continue with the loop's body or jump after the loop (based on the condition).
7. ret indicates the return from the current method.

Language engineers use these operations to enable the construction of DFGs for their languages.

### 9.2.2 Building Variant-specific Data-Flows

MPS is packaged with data-flow builders for Java language concepts. Thus, it is possible to check the data flow of Java programs. However, the analysis is not variability-aware, and thus problems as illustrated in Figure 4.5 can appear (where a variable is initialized in the SPL, but not in a variant).

Fortunately, making the data-flow analysis variant-aware is straightforward. We simply clone-and-own the data-flow builders provided for Java and add variant checks—that is, we check whether a node is in the variant's fragment set and build the DFG accordingly. For instance, we clone-and-own Java's BlockStatement data-flow builder (Fig. 9.4a). Figure 9.4b shows our implementation. The BlockStatement instance's statement list (Line 5) is only included in the DFG if the BlockStatement instance has no fragment annotation—that is, the node is not variational—or has a fragment that is in the variant's fragment cache.

Figure 9.5a gives a more complex example overriding the data-flow builder of Java’s `WhileStatement`. The basic idea is simple: we need support for partially annotated wrappers as illustrated in Figure 9.5b. In fact, there are two different variants, either the entire while-statement (Lines 1–3) or only the body (Line 2) is included, since wrappees can replace their wrapper. Figure 9.5c shows the DFG for the full variant, Figure 9.5d for the variant containing only the body. To be able to create these two DFG variants, we make the `WhileStatement`’s data-flow builder variant-aware as illustrated in Figure 9.5a. Notice that in Line 15, we unconditionally call the data-flow builder of the `WhileStatement`’s statement list (the wrappee). Any other structural element is made optional by checking for fragment annotations and whether the fragment is included in the variant’s fragment set (Lines 3–14 and Lines 16–19). In particular, we do not include the DFG nodes including the code for the condition (Lines 3–14) and the node that jumps to the while’s head (Lines 16–19) if the fragment is not in the variant’s fragment set.

```

1 data flow builder for WhileStatement {
2   (node)->void {
3     if ( node.fragment.isNull
4         || VariantFragmentCache.getInstance().contains(node)) {
5       code for node.condition
6       if(node.condition.isInstanceOf(BooleanConstant)) {
7         node<BooleanConstant> constant = node.condition : BooleanConstant;
8         if(!constant.value) {
9           jump after node
10        }
11      } else {
12        ifjump after node
13      }
14    }
15    code for node.body
16    if ( node.fragment.isNull
17        || VariantFragmentCache.getInstance().contains(node)) {
18      { jump before node }
19    }
20  }
21 }

```

(a) Data-flow builder for Java’s `WhileStatement`

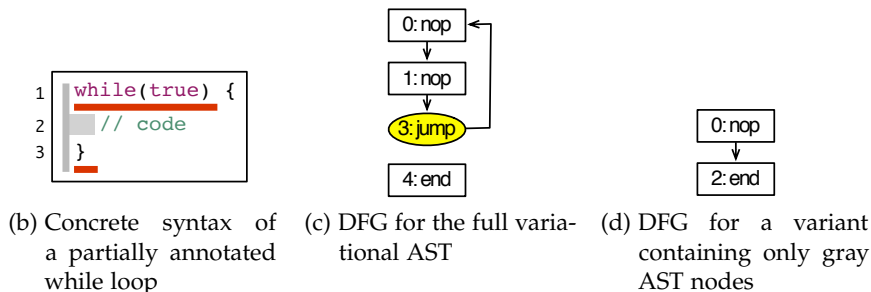


Figure 9.5: Variant-aware data-flow analysis of Java’s `WhileStatement`: data-flow builder (a), and example (b–d)

Figure 9.6 shows more complex DFG variants constructed from the AST of our analysis example depicted in Figure 4.5. In the example, the variable `entryType` is not initialized if the feature module *Transactions* is not selected—that is, `entryType` is initialized in the *Transactions* code (Lines 9 and 12). In the DFG variant with both features being selected, the variable `entryType` is initialized (Fig. 9.6a). In fact, in any branch of the if-statement, we write `entryType` (nodes 24 and 39), and thus `entryType` is initialized during the read (node 54). However, if *Transactions* is not selected (Fig. 9.6b), then there is no write of `entryType`, and thus the read is uninitialized (node 24).

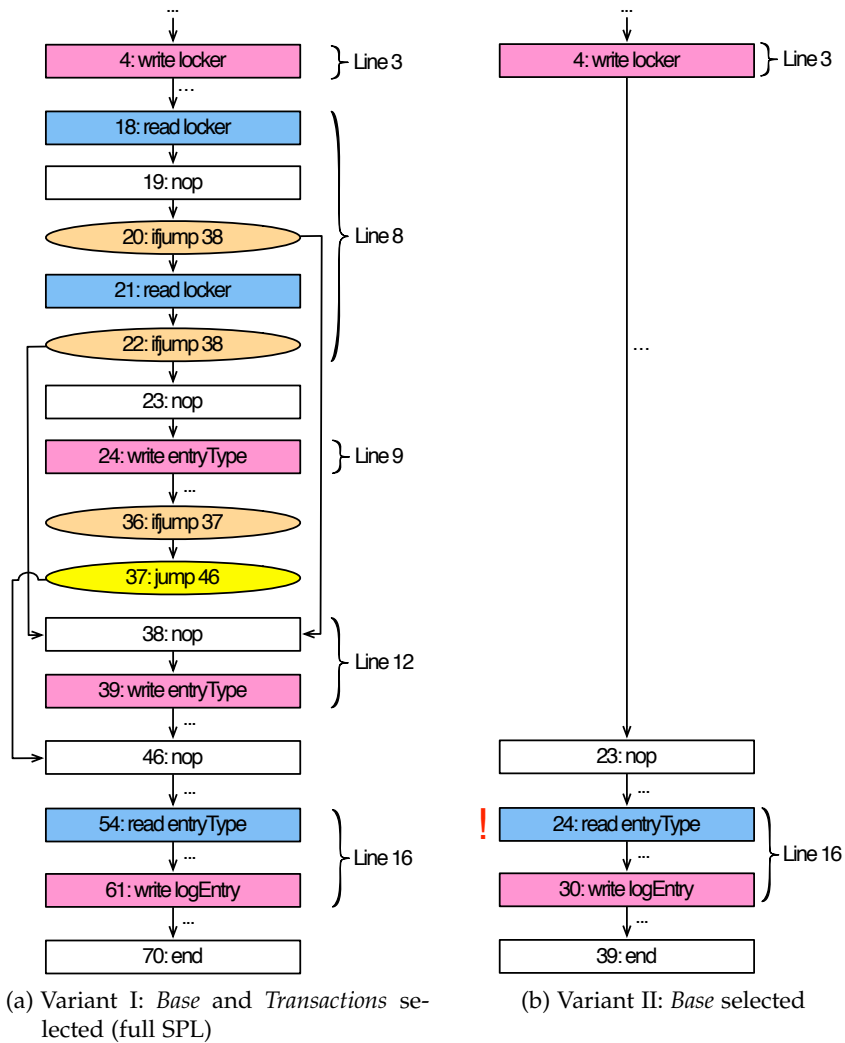


Figure 9.6: DFG variants of the BerkeleyDB example depicted in Figure 4.5 (cf. Line numbers)



### 9.3 CONCLUDING REMARKS

Realizing dependency extraction for a target (mainstream) language, such as Java, is straightforward. However, language engineers must understand the structure of the target language to make sure that all necessary language concepts are checked. Otherwise, the calculated set of dependencies is incomplete, which is a cause for subtle errors. Likewise, realizing a variant-aware data-flow analysis is easy, but requires more engineering effort—that is, we must clone-and-own the variability-unaware data-flow builders and adapt them. For Java, these are 44 language concepts which can be quickly adapted (i.e., it took us two hours to identify and clone-and-own all necessary data-flow builders). However, future work should focus on a better reuse of data-flow builders to enable a more maintainable and less invasive solution.

## EVALUATION II: THE LANGUAGE ENGINEER'S VIEW

We now evaluate the realization of PEOPL from the language engineer's perspective. As in Section 5.1, we use and extend existing classification frameworks [80, 100, 118]. In fact, we quantify the practicality of realizing new external variability representations and new target languages. Moreover, discuss the actual effort for these realizations. We conclude that PEOPL is a productive approach.

### 10.1 CLASSIFICATION II: TOOL REALIZATION

Next, we compare PEOPL and classical variability implementation approaches (i.e., the ones from Sec. 5.1) with regard to *language independence* and *variability representation independence*. Table 10.1 shows our results, which we discuss in detail next.

#### 10.1.1 Language Independence

With regard to *language independence* all approaches perform similarly well. The CPP (●●) is a simple text-processor, and thus not bound to C. The VCS (●●) builds upon the same principles. However, both are bound to text, requiring workarounds for graphical languages.

CIDE (●●) and FH (●●) can be extended for arbitrary target languages. Both take a FeatureBNF language grammar attributed with

	Variability Implementation Approach						
	CPP	CIDE	FH	DeltaJ	VCS	Refactoring <sup>1</sup>	PEoPL
Language independence	●●	●●	●●	○○ <sup>2</sup>	●●	●●	●●
Variability representation independence	○○	●○	○○	○○	●○	●○	●●

●● very good, ●○ good, ●○ medium, ○○ poor, ○○ no support

<sup>1</sup>Refactoring engine to transform CIDE into FH implementations and vice versa [103].

<sup>2</sup>DeltaJ is bound to Java, yet the underlying technique *delta modeling* is language-independent [158, 168].

Table 10.1: Classification of variability implementation techniques from the language engineer's perspective (extended/adapted [100, 118])

variability concepts as an input, and automatically generate the necessary infrastructure, such as a variability-aware parser and pretty printer [10, 105]. Thus, making new languages variability-aware is a simple engineering task. Note that a refactoring engine (●○) that transforms CIDE into FH implementations and vice versa maintains the same language independence as both approaches use FeatureBNF to attribute grammars [10, 105].

DeltaJ (○○) has been built specifically for Java. Yet, DeltaJ implements the concepts of delta modeling, a technique independent of the target language [158]. The tool suite DeltaEcore leverages this language independence and allows language engineers to automatically create a delta language for a given target language [168].

PEoPL's (●○) meta-model is independent of the target language and the artifact type. To make a target language variability-aware, language designers should at least specify the attributable language concepts (from which AST nodes are instantiated). Note that this is pretty similar to attributing a grammar, yet provides more flexibility. For instance, the CoreVar tailoring infrastructure supports including/excluding language concepts and their combinations expressed over a Boolean formula. Yet, to improve the editing experience and robustness, language designers also should implement a modular variability representation as well as language-specific dependency extraction and data-flow analysis rules. All in all, although making languages variability-aware is a simple engineering task, it would be nice to automatize these tasks in the future. To automatize editor creation, the recent concept of *grammar cells* is a first step in the right direction [195].

### 10.1.2 Variability Representation Independence

Variability representation independence measures the possibility and effort to integrate diverse SPL implementation techniques in a single, common environment. Thus, it reflects the fundamental freedom of a developer to choose between different variability representations.

Aside from their core functionality, the CPP, FH, DeltaJ and VCS do not support any additional SPL implementation techniques (○○). CIDE (●○) provides a simple *show or hide mechanism* for features. This way, at least variant editing can be emulated to a certain degree. Yet, a plugin mechanism for adding new variability representations is not available.

In contrast, refactoring engines (●○) are extensible by design—that is, language engineers just add transformation rules. Thus, refactoring engines basically support the integration of variability representations. Figures 10.1a and 10.1b show two possibilities for refactoring variational models: complete and two-way refactoring. Notice that a complete refactoring is impractical, since the involved transforma-

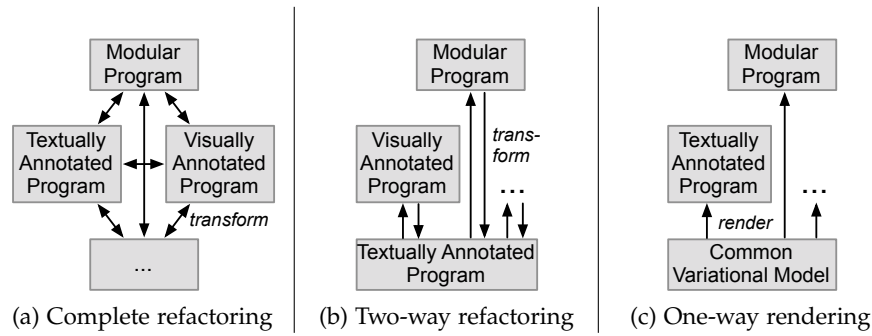


Figure 10.1: Refactoring versus rendering variability representations

tions form a complete graph. That is, a developer must implement transformations from and into any variability representation to be supported by the programming environment. As a result,  $n(n - 1)$  transformations need to be implemented. Moreover, it is unclear how to support a refactoring from and into variants as they are constituted of a reduced feature set.

Two-way refactoring is more practical as developers only need to implement transformations from and into the most expressive variability representation employed (cf. expressive power in Section 5.1.9). For instance, two steps would be necessary to transform a visually annotated program into a modular program. First, we transform the visually annotated program into a textually annotated program, which serves as a transient variability representation. Second, we transform the transient representation into the modular program. Persisting the transient representation in a common core would make variant editing possible, since all information are maintained in the common core (similar to PEOPL). In fact, VCS (●○) provides a view on a variant by employing transformations from and into a single variability representation. More transformations could make the approach more flexible. Notice that the CIDE/FH refactoring engine directly transforms FH into CIDE programs and vice versa, but it could be easily extended to support the proposed two-way refactoring. Still, two-way refactoring requires a significant engineering effort as two potentially very different variational models need to be transformed into each other.

In PEOPL (●●), the effort to realize the desired freedom of freely choosing a variability representation is lower. In fact, instead of implementing complicated transformations, developers must only implement simple one-way rendering rules. Figure 10.1c gives a conceptual overview. A rendering rule is a description of how to render an internal variability representation into a concrete variability-aware syntax. In other words, based on a common core, language engineers just implement rendering rules that reflect the concrete technique to be supported. Editing is performed directly on the common core. However, not all external representations are independent of the target lan-

guage (e.g., Java). Thus, a current drawback is that for each new target language (e.g., C) some external representations must be cloned and owned from an existing target language (e.g., modular and blended representations). In fact, (semi-)automatizing the creation of external representations of these variability representations would be valuable future work. This way, new target languages could be instantly enriched with a set of SPL implementation techniques.

## 10.2 EFFORT FOR EXTENDING PEOPPL

Based on PEoPL's language and variability representation independence the effort for extending PEoPL is low. We now report our experiences.

### 10.2.1 *New Target Languages*

Tailoring PEoPL to a target language, using annotatable nodes and wrapper declarations is as easy as annotating grammars [7, 105]. The effort for creating the declarations depends on the target language's complexity, but was moderate for C [74] and Java (as presented in this dissertation). In fact, thanks to CoreVar's simple language structure and tailoring infrastructure, extending C and Java just takes a few hours (with experience). The engineering intensive part is the implementation of the language-specific data-flow analysis (clown-and-own) and the language-specific external variability representations. Their creation should be automatized in the future.

Moreover, thanks to concept inheritance, languages building upon MPS' Java such as MPS' closure language are inherently supported. The reason is that MPS supports unlimited language composition—that is, different languages can be easily integrated. For instance, a logging language that introduces a new logging statement into Java is directly supported by JavaVar, since a logging statement is just a statement. Tailoring is also flexible, since even non-textual languages such as MPS' math language with its math symbols are supported out of the box.

### 10.2.2 *New Variability Representations*

The realization effort for creating new external representations (rendering rules) is moderate. For instance, it took us only two hours to implement the blended projection (i.e., the editor infrastructure and the Java-specific implementation). However, implementing for instance a modular projection with advanced editing support from scratch requires more engineering effort (e.g., restructuring the tree when typing the original keyword). So, generating projections and editing rules for variability representations, potentially exploiting the

new concept of *grammar cells* [195] for defining projectional editors, would be valuable.

## RELATED WORK

---

We already discussed many differences of PEOPL to other variability implementation approaches throughout this dissertation (cf. Ch. 3 and Sec. 5.1). In the following, we summarize and review the most important related work on integrating SPL implementation techniques. Moreover, we review other important parser-based and projectional approaches.

### 11.1 INTEGRATION OF VARIABILITY REPRESENTATIONS

Only few approaches exist to integrate the very different annotative and modular SPL techniques.

The SDA is a formal model of feature modularity [23, 24]. PEOPL is the first approach to put the corresponding ideas into practice. In fact, we build upon the formalizations, yet use injective partial functions for feature modules to forbid assigning the same fragment to different variation points. This is no limitation in practice, but eases our implementation. To model such (rare) homogeneous extensions [5, 9, 49], we allow variation points to appear multiple times via placeholders. Using these concepts, we formalize variational ASTs, propose a concrete language structure, and conceive a full-fledged tool infrastructure (modeling, implementation, and analysis facilities).

The *compositional choice calculus* is a formal language that combines annotative and compositional techniques [204]. The approach allows adding, removing and replacing feature details. Walkingshaw and Ostermann build upon the calculus, and propose a formal model of editable views on variational software [205]. In particular, they propose to encode choices, in a generic AST, which is used to generate different (simplified) documents. These can be edited, parsed and committed back into the generic AST. These concepts have been also used for a parser-based variant editor [178]. Notice that we also represent the source code in a single, common variational AST that supports alternative paths (i.e., choices). Yet, approaches based on the choice calculus only have one external variability representation (i.e., choice calculus). In other words, implementing variability depends on the syntax of the choice-calculus language. In contrast, PEOPL relies on projectional editing, supports extending arbitrary languages and provides an extensible set of external variability representations. The key difference is that instead of parsing and committing the changes made to a document, modifications are carried out directly on the underlying representation.

Based on parsing, Kästner et al. refactor compositional (FH) into annotative (CIDE) variability representations and vice versa [103]. Yet, refactoring does neither support a fluent movement between techniques nor any other advanced editing support provided by PEOPL. Moreover, parallel editing of different refactored representations can cause inconsistencies, challenging developers.

The *XML Variant Configuration Language (XVCL)* [94, 208] is another attempt to integrate annotative and modular compositional approaches. Annotations are made in the program's text, and thus syntactic correctness is not guaranteed (unlike PEOPL). So-called *break-point annotations* (a.k.a. *hooks* or *extension points*) allow developers to separate implementations by inlining code that is specified externally (similar to the hook method approach, we discussed in Sec. 2.5.1). Yet, stepwise refinement is not supported. In contrast to PEOPL, the concrete syntax is fixed, and thus different variability representations are not supported.

FeatureCopp [118] is another parser-based approach trying to combine annotative and modular techniques. FeatureCopp is realized as a simple CPP-like preprocessor. Similar to XVCL, annotations enable inlining “modular” units via extension points. As a result, true modularity is not supported, since there is no cohesive view on a feature that includes all extension points. The main problem with such manually added extension points, however, is that they are fragile—that is, just renaming an extension point definition invalidates all corresponding extension points, so refactoring is necessary. There is also no advanced editing facility as proposed with PEOPL enabling to change the concrete syntax on demand. For instance, it is not possible to include contextual information of an extension point on demand. All in all, it is unclear how to realize alternative extension points and why one should use the FeatureCopp's inlining mechanism instead of annotated method calls—which, in fact, would ease type checking.

## 11.2 PARSER-BASED VARIABILITY IMPLEMENTATION TECHNIQUES

Most variability implementation techniques and tools are text-based, and thus require a parser. This is a key difference to PEOPL since it separates variability into an internal and external representation leaving the concrete syntax to projections.

CIDE is a parser-based, annotative approach sharing some concepts with PEOPL, such as annotating ASTs and wrappers [105]. Yet, CIDE's views on the code are less sophisticated. For instance, CIDE only supports an annotative and variant view on the code, and thus true modularity is not supported. Moreover, an artifact cannot be edited in parallel using different views, and new views cannot be plugged. CIDE is also less expressive, since its preprocessor only allows adding fea-



ture details, neither replacing nor removing them. In contrast, PEOPL provides support for alternatives (i.e., multiple fragments can fill a VP), which can be also explicitly assigned to non-optional nodes by the developer. CIDE, in turn, only allows language engineers to provide a default value for non-optional language concepts in a grammar specification, which is opaque to the SPL developer.

DeltaJ is another related modular approach that allows developers to add, replace, and remove feature artifacts by applying deltas [115, 157, 158]. Like the meta-model employed in PEOPL, the underlying modeling technique of DeltaJ, delta modeling, is language-independent [168] and not bound to text. In fact, delta modeling has already been applied in various scenarios, for instance, the graphical programming environment of Matlab/Simulink [88]. The key difference to delta-oriented approaches is that we embed in PEOPL all variants into a single variational AST (instead of incrementally applying deltas). Moreover, we leave the concrete syntax of modules open to the different projections.

FeatureIDE [187] is an Eclipse framework that integrates several approaches, such as FH[7], DeltaJ [115], and tools to cope with variability (e.g., feature-context interfaces [160]). The key difference is that the very same feature artifact cannot be explored using different representations. Moreover, if developers use different representations for different feature artifacts, uniformity breaks.

### 11.3 PROJECTIONAL VARIABILITY IMPLEMENTATION TECHNIQUES

A projectional approach to implement SPLs was proposed before [197, 200]. The language family mbeddr provides the only other projectional way to implement SPLs [201], applying disciplined C preprocessor concepts to the underlying AST. mbeddr is also implemented using MPS, but does not focus on providing multiple external representations. Moreover, annotatable nodes cannot be specified and there is no support for partial annotations (i.e., annotating only the wrapper and not the wrappee). As a result, it is unclear how the preprocessor can be applied to other (graphical) languages than mbeddr C in a meaningful way.

On a final note on the strength of projectional editing. Most modular variability implementation approaches, such as DeltaJ, FH [7], and AHEAD [22] enable method replacement in an implicit fashion. For instance, developers simply omit the `original-keyword` or a pendant on the statement level. We argue that in large methods it might be difficult to distinguish between a method replacement and refinement as one needs to search for the `original-keyword` first. Moreover, to make mutual exclusiveness explicit, developers are required to lift this information up into the SPL's declaration using constraints.

In PEOPL, we embed alternatives directly into the variational AST (i.e., multiple fragments assigned to the same VP). The advantage of this approach is that alternatives are already explicit in the implementation and do not need to be lifted up into a constraint manually. In fact, using a projectional editor, we can show textual or visual markers to indicate a method's refinement or replacement.

#### 11.4 OTHER RELATED APPROACHES

Outside the SPL context, so-called *effective views* have been proposed to extract the code from a database into two different text-based views (classes and modules) [93]. Changes made to the concrete syntax can be parsed and merged back into the database. Thus, editing inconsistencies may appear, which is not an issue in projectional editing, since any edit is atomic and directly changes the AST. Moreover, the approach neither supports editing nor generating variants and is tailored/limited to two views not reflecting our notion of features.

## CONCLUSION AND FUTURE WORK

---

We presented the PEOPL approach, which aims at combining the distinct advantages of different representations of variability. It relies on establishing a unified, internal representation that is separated from multiple external representations. These can be used in parallel and on demand for engineering the same variable software artifact. We designed seven complementing representations, allowing developers to edit artifacts using textual and visual annotations, (fade-in) feature modules, a blending of annotations into modules, reusable artifacts, and to edit individual variants. We also conceived different facilities for modeling, analyzing, and managing SPL variability.

We realized the PEOPL approach as a full IDE, building upon the projectional language workbench MPS. We discussed PEOPL's tooling and architecture showing that the tool provides a well-defined structure and can be easily extended. By declaring annotatable nodes, wrappers, and further convenience language concepts, we provide an exemplary tailoring of PEOPL to two target languages: Java and fault trees. Moreover, we realize a Java-specific extraction of feature artifact dependencies and a variability-aware data flow analysis. In the context of the PEOPL project, a tailoring to C has been already realized [74]. Our experience shows that the effort for tailoring PEOPL to a target language and creating new external variability representations is low.

We evaluated PEOPL using three different methodologies. First, from the SPL developer's and the language engineer's perspective, we classify PEOPL together with other variability implementation approaches using a well defined set of quality criteria. This way, the strength of PEOPL is made explicit in comparison to contemporary work underlining that PEOPL is indeed a desirable approach. Second, we adopted eight Java SPLs, showing PEOPL's expressiveness, scalability, and benefits. Most importantly, this evaluation shows that it is in fact feasible to separate internal and external representations, supporting very different ways of editing feature artifacts. Latencies to calculate variants are low, which together with our qualitative experiences from adopting the SPLs and our user studies, evidences a smooth editing experience. Third, we conducted two pilot user studies to learn more about PEOPL's usability. Our results show the practicality of the approach in general, but also underline that modular approaches must be taught in more detail to use diverse external representations efficiently.

With regard to future work, we plan to use and extend PEOPL in the following ways.

#### 1. PROJECTIONAL DELTA-ORIENTED PROGRAMMING

PEoPL already provides the basis for rendering the variational AST into a concrete syntax reflecting Delta] modules [25]. We plan to take our current modular projectional editors as a basis. This way, addition and modification will be supported by the new editor directly. To enable removal, we currently use dummy nodes (i.e., siblings of the node to be removed) [25]. A more concise solution would be to introduce a new *removal annotation*, which could be used by developers to tag the nodes to be removed explicitly.

All in all, due to the great flexibility of projectional editing, it is possible to design editors that conform to the DOP syntax and look exactly the same. Yet, in contrast to editing plain text, a developer's editing gestures change the AST directly. Designing editors that allow a flexible, more text-like editing can be challenging, yet is possible. We plan to provide automatic AST transformations based on string patterns to allow typing the modifies keyword for example.

#### 2. PROJECTIONAL EDITING OF HOMOGENEOUS EXTENSIONS

PEoPL's variational AST supports heterogeneous and homogeneous extensions. The latter is known from AOP, where advice can be applied to multiple join points. To enable such homogeneous extensions, we basically create cross-tree references in the AST and render the reused element in different program positions. From the developer's perspective, a copy-and-paste-like process enables this reuse of code elements. We plan to enable developers to manage homogeneous extensions in a more sophisticated fashion from modules, using quantification and weaving (as known from AOP). This way, developers can provide a query (e.g., the name of methods to be extended), which automatically creates the necessary references in the AST.

#### 3. GENERATING RUNTIME VARIABILITY ON DEMAND

PEoPL removes any variability from a variational AST to create software variants (i.e., any variability is removed in a model-to-text transformation). This way, PEoPL is currently bound to compile-time variability. We plan to provide the concepts and a prototypical implementation of transformation rules for generating runtime variability on demand. Our objective is to control variation at runtime (e.g., based on external parameters), which also enables us to employ common dynamic analysis methods [134, 148].

#### 4. FROM VERSION-CONTROL TO VARIATION-CONTROL

With PEoPL, we conceived ways of embedding variability into an AST. We plan to provide the concepts and a prototypical implementation for synchronizing this common, variational AST,

using version control systems. This way, we enable collaborative SPL development and customers can be provided with a specific variant (subset) containing only relevant features. Then, changes made by the customer can be merged back into the main repository.

#### 5. VARIABILITY-AWARE TYPE CHECKING

We do not provide full, variability-aware type-checking (cf. [186]). Thus, there are some problems we cannot detect in advance, such as duplicated method signatures that occur due to parameter removal. Addressing this issue is our future work. Yet, our dependency checker and variant-based data-flow analysis already help to resolve many related issues.

#### 6. CONTROLLED EXPERIMENT AND LONGITUDINAL STUDY

Finally, we also plan to conduct a controlled experiment with users to investigate exact usage scenarios of multiple projections. On this purpose, we plan to create a catalogue of tasks providing hypothesis for which variability representation might leverage in which situation. Then, we evaluate these hypothesis with the aim to provide sound empirical results. This way, we could create a recommender system, supporting developers in choosing the best technique for a given task. On this basis, it would be interesting to investigate how external representations are used in a longitudinal study. In addition, conducting a controlled experiment and longitudinal study with experienced industrial developers would be valuable. However, they typically work with annotative approaches, and thus a sufficient training for modular approaches would be required as well.

Finally, we hope that language designers and tool vendors will create further projections for other variability representations, and tailorings for more languages, beyond Java, C, and fault trees.

## BIBLIOGRAPHY

---

- [1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. “42 variability bugs in the linux kernel.” In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2014, pp. 421–432 (cit. on pp. [16](#), [35](#)).
- [2] Vander Alves, Pedro Matos Jr., Leonardo Cole, Alexandre Vasconcelos, Paulo Borba, and Geber Ramalho. “Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming.” In: *Transactions on Aspect-Oriented Software Development IV*. Berlin, Heidelberg: Springer, 2007, 117–142. ISBN: 978-3-540-77042-8 (cit. on pp. [14](#), [27](#)).
- [3] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Ștefan Stănculescu, Andrzej Wasowski, and Ina Schäfer. “Flexible Product Line Engineering with a Virtual Platform.” In: *Companion Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 532–535 (cit. on pp. [10](#), [68](#)).
- [4] Giuliano Antoniol, Ettore Merlo, Yann-Gaël Guéhéneuc, and Houari A Sahraoui. “On Feature Traceability in Object Oriented Programs.” In: *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*. New York, NY, USA: ACM, 2005, pp. 73–78 (cit. on pp. [36](#), [72](#)).
- [5] Sven Apel. “How AspectJ is Used: An Analysis of Eleven AspectJ Programs.” In: *Journal of Object Technology* 9.1 (2010), pp. 117–142 (cit. on pp. [49](#), [75](#), [162](#)).
- [6] Sven Apel and Christian Kästner. “An Overview of Feature-Oriented Software Development.” In: *Journal of Object Technology* 8.5 (2009), pp. 49–84 (cit. on p. [36](#)).
- [7] Sven Apel, Christian Kästner, and Christian Lengauer. “Language-Independent and Automated Software Composition: The FeatureHouse Experience.” In: *IEEE Transactions on Software Engineering* 39.1 (2013), pp. 63–79 (cit. on pp. [2](#), [3](#), [14](#), [15](#), [22](#), [27](#), [36](#), [74](#), [75](#), [76](#), [81](#), [160](#), [164](#)).
- [8] Sven Apel, Christian Kästner, and Salvador Trujillo. “On the Necessity of Empirical Studies in the Assessment of Modularization Mechanisms for Crosscutting Concerns.” In: *Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 1–6 (cit. on p. [32](#)).

- [9] Sven Apel, Thomas Leich, and Gunter Saake. "Aspectual Feature Modules." In: *IEEE Transactions on Software Engineering* 34.2 (2008), pp. 162–180 (cit. on pp. 49, 74, 162).
- [10] Sven Apel and Christian Lengauer. "Superimposition: A Language-Independent Approach to Software Composition." In: *Proceedings of the 7th International Conference on Software Composition*. Berlin, Heidelberg: Springer, 2008, pp. 20–35 (cit. on pp. 27, 158).
- [11] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. "FeatureC++: On the Symbiosis of Feature-oriented and Aspect-oriented Programming." In: *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*. Berlin, Heidelberg: Springer, 2005, pp. 125–140 (cit. on pp. 2, 14, 27).
- [12] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kastner. "Detecting Dependences and Interactions in Feature-Oriented Design." In: *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 161–170 (cit. on p. 63).
- [13] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. "Language-independent Reference Checking in Software Product Lines." In: *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2010, pp. 65–71 (cit. on p. 63).
- [14] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-37520-0 (cit. on pp. 2, 10, 11, 14, 15, 18, 48, 49, 66, 67, 69, 70, 71, 72, 73).
- [15] Uwe Aßmann. "From Modular to Composition Systems." In: *Invasive Software Composition*. Berlin, Heidelberg: Springer, 2003, pp. 63–104. ISBN: 978-3-662-05082-8 (cit. on p. 15).
- [16] David L. Atkins, Thomas Ball, Todd L. Graves, and Audris Mockus. "Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor." In: *IEEE Transactions on Software Engineering* 28.7 (2002), pp. 625–637 (cit. on pp. 14, 28, 35).
- [17] Alan D. Baddeley. "Is working memory still working?" In: *American Psychologist* 56.11 (2001), pp. 851–864 (cit. on p. 33).
- [18] Kacper Bąk, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. "Clafer: Unifying Class and Feature Modeling." In: *Software & Systems Modeling* 15.3 (2016), pp. 811–845 (cit. on p. 12).

- [19] François Bancilhon and Nicolas Spyratos. "Update Semantics of Relational Views." In: *ACM Transactions on Database Systems* 6.4 (1981), pp. 557–575 (cit. on p. 28).
- [20] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 1998. ISBN: 978-3-642-37520-0 (cit. on p. 15).
- [21] Don Batory. "Feature Models, Grammars, and Propositional Formulas." In: *Proceedings of the 9th International Conference on Software Product Lines*. Berlin, Heidelberg: Springer, 2005, pp. 7–20 (cit. on pp. 12, 14).
- [22] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. "Scaling step-wise refinement." In: *IEEE Transactions on Software Engineering* 30.6 (2004), pp. 355–371 (cit. on pp. 2, 3, 14, 15, 22, 27, 70, 75, 164).
- [23] Don Batory, Peter Höfner, Bernhard Möller, and Andreas Zelend. "Features, Modularity, and Variation Points." In: *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2013, pp. 9–16 (cit. on p. 162).
- [24] Don Batory, Peter Höfner, Dominik Köppl, Bernhard Möller, and Andreas Zelend. "Structured Document Algebra in Action." In: *Software, Services and Systems*. Vol. LNCS Volume 8950. 2. Cham: Springer International Publishing, 2015, pp. 291–311. ISBN: 978-3-319-15545-6 (cit. on pp. 105, 162).
- [25] Benjamin Behringer and Moritz Fey. "Implementing Delta-oriented SPLs Using PEOPL: An Example Scenario and Case Study." In: *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2016, pp. 28–38 (cit. on pp. 74, 81, 167).
- [26] Benjamin Behringer, Martina Lehser, and Steffen Rothkugel. "Towards Feature-Oriented Fault Tree Analysis." In: *Proceedings of the IEEE 38th International Computer Software and Applications Conference Workshops*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 522–527 (cit. on pp. 52, 53).
- [27] Benjamin Behringer, Jochen Palz, and Thorsten Berger. "PEOPL: Projectional Editing of Product Lines." In: *Proceedings of the 39th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Computer Society Press, 2017, pp. 563–574 (cit. on p. 71).
- [28] Benjamin Behringer and Steffen Rothkugel. "Integrating Feature-based Implementation Approaches Using a Common Graph-based Representation." In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2016, pp. 1504–1511 (cit. on pp. 102, 107).



- [29] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. “Variability Modeling in the Real: A Perspective from the Operating Systems Domain.” In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2010, pp. 73–82 (cit. on p. 16).
- [30] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. “What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines.” In: *Proceedings of the 19th International Conference on Software Product Line*. New York, NY, USA: ACM, 2015, pp. 16–25 (cit. on pp. 2, 10).
- [31] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweessap Dangprasert, and Janet Siegmund. “Efficiency of Projectional Editing: A Controlled Experiment.” In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2016, pp. 763–774 (cit. on pp. 39, 69, 93).
- [32] Lorenzo Bettini, Ferruccio Damiani, and Ina Schäfer. “Implementing Software Product Lines Using Traits.” In: *Proceedings of the ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2010, pp. 2096–2102 (cit. on p. 27).
- [33] Lorenzo Bettini, Ferruccio Damiani, and Ina Schäfer. “Compositional Type Checking of Delta-oriented Software Product Lines.” In: *Acta Informatica* 50.2 (2013), pp. 77–122 (cit. on pp. 25, 34).
- [34] Lorenzo Bettini, Ferruccio Damiani, and Ina Schäfer. “Implementing Type-safe Software Product Lines Using Parametric Traits.” In: *Science of Computer Programming* 97.P3 (2015), pp. 282–308 (cit. on p. 27).
- [35] Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. “Variability Management with Feature Models.” In: *Science of Computer Programming* 53.3 (2004), pp. 333–352 (cit. on p. 21).
- [36] Ted J. Biggerstaff. “A Perspective of Generative Reuse.” In: *Annals of Software Engineering* 5.1 (1998), pp. 169–226 (cit. on p. 36).
- [37] Eric Bodden, Tárzis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. “SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years.” In: *SIGPLAN Notices* 48.6 (2013), pp. 355–364 (cit. on p. 64).
- [38] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981. ISBN: 978-0-138-22122-5 (cit. on p. 31).

- [39] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. “Intraprocedural Dataflow Analysis for Software Product Lines.” In: *Transactions on Aspect-Oriented Software Development X* 7800.Chapter 3 (2013), pp. 73–108 (cit. on p. 64).
- [40] Gilad Bracha and William Cook. “Mixin-based Inheritance.” In: *Proceedings of the European Conference on Object-oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 1990, pp. 303–311 (cit. on p. 27).
- [41] Avi Bryant, Andrew Catton, Kris De Volder, and Gail C. Murphy. “Explicit programming.” In: *Proceedings of the 1st International Conference on Aspect-oriented Software Development*. New York, NY, USA: ACM, 2002, pp. 10–18 (cit. on p. 15).
- [42] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. “Feature interaction: A Critical Review and Considered Forecast.” In: *Computer Networks* 41.1 (2003), pp. 115–141 (cit. on p. 17).
- [43] Luca Cardelli. “Program Fragments, Linking, and Modularization.” In: New York, NY, USA: ACM, 1997, pp. 266–277 (cit. on p. 71).
- [44] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. “Symbolic Model Checking of Software Product Lines.” In: *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA: ACM, 2011, pp. 321–330 (cit. on p. 65).
- [45] Paul Clements and Charles Kreuger. “Point: Being Proactive Pays Off / Counterpoint: Eliminating the Adoption Barrier.” In: *IEEE Software* 19.4 (2002), pp. 28–31 (cit. on p. 68).
- [46] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0-201-70332-7 (cit. on pp. 2, 10).
- [47] David Coppit and Benjamin Cox. “Software plans for separation of concerns.” In: *Proceedings of the 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*. Lancaster, UK, 2004 (cit. on p. 44).
- [48] David Coppit, Robert R. Painter, and Meghan Revelle. “Spotlight: A Prototype Tool for Software Plans.” In: *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 2007, pp. 754–757 (cit. on pp. 22, 44).
- [49] Adrian Coyler, Awais Rashid, and Gordon Blair. *On the Separation of Concerns in Program Families*. Tech. rep. COMP-001-2004. Lancaster University, 2004 (cit. on pp. 49, 74, 162).

- [50] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., 2000. ISBN: 0-201-30977-7 (cit. on p. 72).
- [51] 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 (cit. on p. 12).
- [52] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F Terwilliger. “Bidirectional Transformations: A Cross-Discipline Perspective.” In: *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*. Berlin, Heidelberg: Springer, 2009, pp. 260–283 (cit. on p. 28).
- [53] Ferruccio Damiani and Ina Schäfer. “Family-Based Analysis of Type Safety for Delta-Oriented Software Product Lines.” In: *Proceedings of the 5th International Conference on Leveraging Applications of Formal Methods, Verification and Validation: Technologies for Mastering Change - Volume Part I*. Berlin, Heidelberg: Springer, 2012, pp. 193–207 (cit. on p. 34).
- [54] Umeshwar Dayal and Philip A. Bernstein. “On the Correct Translation of Update Operations on Relational Views.” In: *ACM Transactions on Database Systems* 7.3 (1982), pp. 381–416 (cit. on p. 28).
- [55] Josh Dehlinger and Robyn R. Lutz. “Software Fault Tree Analysis for Product Lines.” In: *Proceedings of the 8th IEEE International Conference on High Assurance Systems Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 12–21 (cit. on p. 52).
- [56] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, 1976. ISBN: 013215871X (cit. on pp. 15, 22, 33).
- [57] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. “Variability Abstractions: Trading Precision for Speed in Family-Based Analyses.” In: *Proceedings of the 29th European Conference on Object-Oriented Programming*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 247–270 (cit. on p. 64).
- [58] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P Black. “Traits: A Mechanism for Fine-grained Reuse.” In: *ACM Transactions on Programming Languages and Systems* 28.2 (2006), pp. 331–388 (cit. on p. 27).
- [59] Alastair Dunsmore and Marc Roper. *A Comparative Evaluation of Program Comprehension Measures*. Tech. rep. EFOCS-35-2000. University of Strathclyde, 2000 (cit. on p. 31).

- [60] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. "The State of the Art in Language Workbenches." In: *Software Language Engineering*. Cham: Springer, 2013, pp. 26–28 (cit. on pp. 39, 97, 98).
- [61] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. "Evaluating and Comparing Language Workbenches." In: *Computer Languages, Systems & Structures* 44.PA (2015), pp. 24–47 (cit. on pp. 39, 97, 98).
- [62] Michael D. Ernst, Greg J. Badros, and David Notkin. "An Empirical Analysis of C Preprocessor Use." In: *IEEE Transactions on Software Engineering* 28.12 (2002), pp. 1146–1170 (cit. on p. 32).
- [63] Martin Erwig and Eric Walkingshaw. "The Choice Calculus: A Representation for Software Variation." In: *ACM Transactions on Software Engineering and Methodology* 21.1 (2011), 6:1–6:27 (cit. on p. 21).
- [64] D. Faust and Chris Verhoef. "Software Product Line Migration and Deployment." In: *Software: Practice and Experience* 33.10 (2003), pp. 933–955 (cit. on pp. 10, 68).
- [65] Jean-Marie Favre. "Preprocessors from an Abstract Point of View." In: *Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1996 (cit. on pp. 2, 16, 32).
- [66] Jean-Marie Favre. "Understanding-In-The-Large." In: *Proceedings of the 5th International Workshop on Program Comprehension*. Los Alamitos, CA, USA: IEEE Computer Society, 1997, pp. 29–38 (cit. on p. 16).
- [67] Janet Feigenspan, Christian Kästner, Sven Apel, and Thomas Leich. "How to Compare Program Comprehension in FOSD Empirically: An Experience Report." In: *Proceedings of the First International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2009, pp. 55–62 (cit. on pp. 11, 32).
- [68] Janet Feigenspan, Maria Papendieck, Christian Kästner, Mathias Frisch, and Raimund Dachsel. "FeatureCommander: Colorful #Ifdef World." In: *Proceedings of the 15th International Software Product Line Conference, Volume 2*. New York, NY, USA: ACM, 2011, 48:1–48:2 (cit. on p. 22).

- [69] Janet Feigenspan, Michael Schulze, Maria Papendieck, Christian Kästner, Raimund Dachsel, Veit Köppen, and Mathias Frisch. "Using background colors to support program comprehension in software product lines." In: *15th Annual Conference on Evaluation Assessment in Software Engineering*. Institution of Engineering and Technology, 2011, pp. 66–75 (cit. on pp. 22, 32).
- [70] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. "Measuring Programming Experience." In: *Proceedings of the 20th IEEE International Conference on Program Comprehension*. IEEE Computer Society Press, 2012, pp. 73–82 (cit. on p. 85).
- [71] Janet Feigenspan, Michael Schulze, Maria Papendieck, Christian Kästner, Raimund Dachsel, Veit Köppen, Mathias Frisch, and Gunter Saake. "Supporting program comprehension in large preprocessor-based software product lines." In: *IET Software* 6.6 (2012), pp. 488–501 (cit. on p. 22).
- [72] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachsel, Maria Papendieck, Thomas Leich, and Gunter Saake. "Do Background Colors Improve Program Comprehension in the #Ifdef Hell?" In: *Empirical Software Engineering* 18.4 (2013), pp. 699–745 (cit. on pp. 22, 32).
- [73] Wolfram Fenske and Sandro Schulze. "Code Smells Revisited: A Variability Perspective." In: *Proceedings of the 9th International Workshop on Variability Modelling of Software-intensive Systems*. New York, NY, USA: ACM, 2015, 3:3–3:10 (cit. on pp. 32, 34, 35).
- [74] Moritz Fey. "Projektionales Editieren von C-basierten Software-Produktlinien (German)." MA thesis. Saarland University of Applied Sciences, 2017 (cit. on pp. 42, 98, 160, 166).
- [75] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. "Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants." In: *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 391–400 (cit. on pp. 10, 68).
- [76] J. Nathan Foster, Michael B Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. "Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem." In: *ACM Transactions on Programming Languages and Systems* 29.3 (2007), 17:1–17:65 (cit. on p. 28).

- [77] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-48567-2 (cit. on p. 34).
- [78] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* 2005. URL: <http://www.martinfowler.com/articles/languageWorkbench.html> (visited on 05/01/2017) (cit. on p. 97).
- [79] Konstantin Friesen. “Entwicklung einer Werkzeugunterstützung für DeltaJava und Evaluierung der Sprache anhand einer Fallstudie (German).” MA thesis. TU Braunschweig, Sept. 2012 (cit. on p. 78).
- [80] Cristina Gacek and Michalis Anastasopoulos. “Implementing Product Line Variabilities.” In: *Proceedings of the Symposium on Software Reusability: Putting Software Reuse in Context*. New York, NY, USA: ACM, 2001, pp. 109–117 (cit. on pp. 15, 32, 66, 157).
- [81] Carolina Gómez, Peter Liggesmeyer, and Ariane Sutor. “Variability Management of Safety and Reliability Models: An Intermediate Model towards Systematic Reuse of Component Fault Trees.” In: *Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security*. Berlin, Heidelberg: Springer, 2010, pp. 28–40 (cit. on p. 52).
- [82] Pete Goodliffe. *Becoming a Better Programmer: A Handbook for People Who Care About Code*. O’Reilly Media Inc., 2014. ISBN: 978-1491905531 (cit. on p. 35).
- [83] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java(TM) Language Specification, 3rd Edition*. Addison-Wesley Professional, 2005. ISBN: 0321246780 (cit. on p. 20).
- [84] Alexander Grebhahn, Sebastian Kuckuk, Christian Schmitt, Harald Köstler, Norbert Siegmund, Sven Apel, Frank Hannig, and Jürgen Teich. “Experiments on Optimizing the Performance of Stencil Codes with SPL Conqueror.” In: *Parallel Processing Letters* 24.03 (2014), pp. 1–19 (cit. on p. 51).
- [85] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. “Exploiting the Map Metaphor in a Tool for Software Evolution.” In: *Proceedings of the 23rd International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 265–274 (cit. on p. 22).
- [86] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. “On the Notion of Variability in Software Product Lines.” In: *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 45–54 (cit. on p. 36).

- [87] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schäfer. “Delta-oriented Architectural Variability Using MontiCore.” In: *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*. New York, NY, USA: ACM, 2011, 6:1–6:10 (cit. on p. 27).
- [88] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schäfer. “First-class Variability Modeling in Matlab/Simulink.” In: *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems*. New York, NY, USA: ACM, 2013, 4:1–4:8 (cit. on pp. 27, 164).
- [89] Ying Hu, Ettore Merlo, Michel Dagenais, and Bruno Laguë. “C/C++ Conditional Compilation Analysis using Symbolic Execution.” In: *Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 196–206 (cit. on p. 32).
- [90] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. “Preprocessor-based Variability in Open-source and Industrial Software Systems: An Empirical Study.” In: *Empirical Software Engineering* 21.2 (2016), pp. 449–482 (cit. on pp. 32, 68).
- [91] Jiyong Jang, Abeer Agrawal, and David Brumley. “ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions.” In: *Proceedings of the IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 48–62 (cit. on p. 10).
- [92] Doug Janzen and Kris De Volder. “Navigating and Querying Code Without Getting Lost.” In: *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*. New York, NY, USA: ACM, 2003, pp. 178–187 (cit. on p. 22).
- [93] Doug Janzen and Kris De Volder. “Programming with Cross-cutting Effective Views.” In: *Proceedings of the European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer, 2004, pp. 197–220 (cit. on p. 165).
- [94] Stan Jarzabek, Paul Bassett, Hongyu Zhang, and Weishan Zhang. “XVCL: XML-based Variant Configuration Language.” In: *Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 810–811 (cit. on pp. 2, 14, 21, 163).
- [95] Merijn de Jonge and Joost Visser. “Grammars as Feature Diagrams.” In: *Proceedings of the 7th International Conference on Software Reuse (Workshop on Generative Programming)*. Berlin, Heidelberg: Springer, 2002, pp. 23–24 (cit. on p. 12).

- [96] Elmar Jürgens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. “Do Code Clones Matter?” In: *Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 485–495 (cit. on pp. 48, 75).
- [97] Christian Kaestner. “Virtual Separation of Concerns: Toward Preprocessors 2.0.” PhD thesis. Otto-von-Guericke-Universität Magdeburg, May 2010 (cit. on p. 15).
- [98] Bernhard Kaiser, Peter Liggesmeyer, and Oliver Mäckel. “A New Component Concept for Fault Trees.” In: *Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software - Volume 33*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2003, pp. 37–46 (cit. on p. 52).
- [99] 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. CMU, 1990 (cit. on pp. 12, 13, 15).
- [100] Christian Kästner and Sven Apel. “Integrating Compositional and Annotative Approaches for Product Line Engineering.” In: *Proceedings of GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*. Passau: University of Passau, 2008, pp. 35–40 (cit. on pp. 3, 5, 11, 36, 66, 67, 68, 69, 72, 157).
- [101] Christian Kästner, Sven Apel, and Don Batory. “A Case Study Implementing Features Using AspectJ.” In: *Proceedings of the 11th International Software Product Line Conference*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 223–232 (cit. on pp. 49, 76, 81).
- [102] Christian Kästner, Sven Apel, and Martin Kuhlemann. “Granularity in Software Product Lines.” In: *Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA: ACM, 2008, pp. 311–320 (cit. on pp. 2, 11, 14, 19, 28, 30, 31, 32, 33, 45, 69, 71, 76, 78, 81).
- [103] Christian Kästner, Sven Apel, and Martin Kuhlemann. “A Model of Refactoring Physically and Virtually Separated Features.” In: *Proceedings of the 8th International Conference on Generative Programming and Component Engineering*. New York, NY, USA: ACM, 2009, pp. 157–166 (cit. on pp. 3, 36, 66, 67, 72, 157, 163).
- [104] Christian Kästner, Salvador Trujillo, and Sven Apel. “Visualizing Software Product Line Variabilities in Source Code.” In: *Proceedings of SPLC Workshop on Visualization in Software Product Line Engineering*. Lero: University of Limerick, 2008, pp. 303–313 (cit. on pp. 14, 28).



- [105] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don Batory. “Guaranteeing Syntactic Correctness for All Product Line Variants: A Language-Independent Approach.” In: *Proceedings of the 47th International Conference on Objects, Components, Models and Patterns TOOLS EUROPE*. Berlin, Heidelberg: Springer, 2009, pp. 175–194 (cit. on pp. 20, 71, 158, 160, 163).
- [106] Christian Kästner, Sven Apel, Syed Saif Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. “On the Impact of the Optional Feature Problem: Analysis and Case Studies.” In: *Proceedings of the 13th International Software Product Line Conference*. Pittsburgh, PA, USA: Carnegie Mellon University, 2009, pp. 181–190 (cit. on p. 63).
- [107] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. “Type Checking Annotation-based Product Lines.” In: *ACM Transactions on Software Engineering and Methodology* 21.3 (2012), 14:1–14:39 (cit. on p. 65).
- [108] Dirk O. Keck and Paul J. Kühn. “The Feature and Service Interaction Problem in Telecommunications Systems: A Survey.” In: *IEEE Transactions on Software Engineering* 24.10 (1998), pp. 779–796 (cit. on p. 17).
- [109] Mik Kersten and Gail C. Murphy. “Mylar: A degree-of-interest Model for IDEs.” In: *Proceedings of the 4th International Conference on Aspect-oriented Software Development*. New York, NY, USA: ACM, 2005, pp. 159–168 (cit. on pp. 33, 85).
- [110] Gregor Kiczales and Mira Mezini. “Separation of Concerns with Procedures, Annotations, Advice and Pointcuts.” In: *Proceedings of the 19th European Conference on Object-Oriented Programming*. Vol. 3586. Chapter 9. Berlin, Heidelberg: Springer, 2005, pp. 195–213 (cit. on p. 15).
- [111] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. “Aspect-oriented programming.” In: *Proceedings of the 11th European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer, 1997, pp. 220–242 (cit. on pp. 14, 27, 49).
- [112] Laurent Kirsch, Jean Botev, and Steffen Rothkugel. “Snippets and Component-Based Authoring Tools for Reusing and Connecting Documents.” In: *Journal of Digital Information Management* 10.6 (2012), pp. 399–409 (cit. on p. 102).
- [113] Laurent Kirsch, Jean Botev, and Steffen Rothkugel. “The Snippet Platform Architecture: Dynamic and Interactive Compound Documents.” In: *International Journal of Future Computer and Communication* 3.3 (2013), pp. 161–167 (cit. on p. 102).

- [114] Jürgen Koenemann and Scott P. Robertson. “Expert Problem Solving Strategies for Program Comprehension.” In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, 1991, pp. 125–130 (cit. on p. 31).
- [115] Jonathan Koscielny, Sönke Holthusen, Ina Schäfer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. “DeltaJ 1.5: Delta-oriented Programming for Java 1.5.” In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. New York, NY, USA: ACM, 2014, pp. 63–74 (cit. on pp. 2, 15, 25, 60, 62, 76, 78, 164).
- [116] Maren Krone and Gregor Snelting. “On the Inference of Configuration Structures from Source Code.” In: *Proceedings of 16th International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society, 1994, pp. 49–57 (cit. on p. 32).
- [117] Charles W. Krueger. “Easing the Transition to Software Mass Customization.” In: *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*. London, UK: Springer, 2002, pp. 282–293 (cit. on p. 21).
- [118] Jacob Krüger, Ivonne Schröter, Andy Kenner, Christopher Kruczek, and Thomas Leich. “FeatureCoPP: Compositional Annotations.” In: *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2016, pp. 74–84 (cit. on pp. 3, 5, 15, 66, 67, 157, 163).
- [119] Vincent J. Kruskal. “Managing Multi-Version Programs with an Editor.” In: *IBM Journal of Research and Development* 28.1 (1984), pp. 74–81 (cit. on p. 28).
- [120] Bernt Kullbach and Volker Riediger. “Folding: An Approach to Enable Program Understanding of Preprocessed Languages.” In: *Proceedings of the 8th Working Conference on Reverse Engineering*. IEEE Computer Society, 2001, pp. 3–12 (cit. on p. 28).
- [121] Duc Le, Eric Walkingshaw, and Martin Erwig. “#Idef Confirmed Harmful: Promoting Understandable Software Variation.” In: *IEEE Symposium on Visual Languages and Human-Centric Computing*. 2011, pp. 143–150 (cit. on pp. 2, 14, 16).
- [122] Jaejoon Lee, Dirk Muthig, and Matthias Naab. “An Approach for Developing Service Oriented Product Lines.” In: *Proceedings of the 12th International Software Product Line Conference*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 275–284 (cit. on p. 15).

- [123] Christian Lengauer, Sven Apel, Matthias Bolten, Armin Größlinger, Frank Hannig, Harald Köstler, Ulrich Rude, Jürgen Teich, Alexander Grebhahn, Stefan Kronawitter, Sebastian Kuckuk, Hannah Rittich, and Christian Schmitt. “ExaStencils - Advanced Stencil-Code Engineering.” In: *Euro-Par: Parallel Processing Workshops, Part II. Lecture Notes in Computer Science*. Cham: Springer, 2014, pp. 553–564 (cit. on p. 51).
- [124] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. “CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code.” In: *IEEE Transactions on Software Engineering* 32.3 (2006), pp. 176–192 (cit. on pp. 48, 75).
- [125] Jörg Liebig, Christian Kästner, and Sven Apel. “Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code.” In: *Proceedings of the 10th International Conference on Aspect-oriented Software Development*. New York, NY, USA: ACM, 2011, pp. 191–202 (cit. on pp. 18, 20, 32, 71).
- [126] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. “An Analysis of the Variability in Forty Preprocessor-based Software Product Lines.” In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA: ACM, 2010, pp. 105–114 (cit. on pp. 16, 32, 49, 75).
- [127] Jia Liu, Don Batory, and Srinivas Nedunuri. “Modeling Interactions in Feature Oriented Software Designs.” In: *Proceedings of the International Conference on Feature Interactions in Telecommunications and Software Systems*. IOS Press, 2005, pp. 178–197 (cit. on p. 63).
- [128] Roberto E. Lopez-Herrejon and Don Batory. “A Standard Problem for Evaluating Product-Line Methodologies.” In: *Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering*. London, UK: Springer, 2001, pp. 10–24 (cit. on pp. 76, 78).
- [129] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. “Evolution of the Linux Kernel Variability Model.” In: *Proceedings of the 14th International Conference on Software Product Lines: going beyond*. Berlin, Heidelberg: Springer, 2010, pp. 136–150 (cit. on p. 12).
- [130] Hidehiko Masuhara and Gregor Kiczales. “Modeling Cross-cutting in Aspect-Oriented Mechanisms.” In: *Proceedings of the 17th European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer, 2003, pp. 2–28 (cit. on p. 74).

- [131] Anneliese von Mayrhauser, A. Marie Vans, and Adele E. Howe. “Program Understanding Behaviour During Enhancement of Large-scale Software.” In: *Journal of Software Maintenance* 9.5 (1997), pp. 299–327 (cit. on p. 31).
- [132] Thomas Mechenbier. “Visual and Textual Projectional Editing of Fault Trees.” Bachelor Thesis. Saarland University of Applied Sciences, 2017 (cit. on p. 52).
- [133] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. “The Love/Hate Relationship with the C Preprocessor - An Interview Study.” In: *29th European Conference on Object-Oriented Programming*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 495–518 (cit. on p. 35).
- [134] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. “On essential configuration complexity: measuring interactions in highly-configurable systems.” In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2016, pp. 483–494 (cit. on p. 167).
- [135] Jean Melo, Claus Brabrand, and Andrzej Wasowski. “How Does the Degree of Variability Affect Bug Finding?” In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: ACM, 2016, pp. 679–690 (cit. on pp. 2, 32, 33, 35, 76).
- [136] Thilo Mende, Rainer Koschke, and Felix Beckwermert. “An Evaluation of Code Similarity Identification for the Grow-and-prune Model.” In: *Journal of Software Maintenance and Evolution* 21.2 (2009), pp. 143–169 (cit. on pp. 10, 68).
- [137] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. “Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study.” In: *IEEE Transactions on Software Engineering* 41.8 (2015), pp. 820–841 (cit. on pp. 6, 63, 64, 148).
- [138] L. J. Najjar. *Using colors effectively*. Tech. rep. TR52.0018. IBM Corp., Atlanta, GA, 1990 (cit. on p. 32).
- [139] Armstrong Nhlabatsi, Robin Laney, and Bashar Nuseibeh. “Feature interaction: the security threat from within software systems.” In: *Progress in Informatics* 5 (2008), pp. 75–89 (cit. on p. 17).
- [140] Harold Ossher and William Harrison. “Combination of Inheritance Hierarchies.” In: *ACM SIGPLAN Notices* 27.10 (1992), pp. 25–40 (cit. on p. 36).

- [141] Harold Ossher and Peri Tarr. “Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software.” In: *Communications of the ACM* 44.10 (2001), pp. 43–50 (cit. on pp. 33, 71).
- [142] David Lorge Parnas. “On the Criteria to Be Used in Decomposing Systems into Modules.” In: *Communications of the ACM* 15.12 (1972), pp. 1053–1058 (cit. on pp. 22, 33).
- [143] David Lorge Parnas. “On the Design and Development of Program Families.” In: *IEEE Transactions on Software Engineering* 2.1 (1976), pp. 1–9 (cit. on p. 22).
- [144] David Lorge Parnas. “Designing Software for Ease of Extension and Contraction.” In: *Proceedings of the 3rd International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 1978, pp. 264–277 (cit. on p. 33).
- [145] Vaclav Pech, Alex Shatalin, and Markus Völter. “JetBrains MPS As a Tool for Extending Java.” In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. New York, NY, USA: ACM, 2013, pp. 165–168 (cit. on pp. 4, 39, 97).
- [146] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005. ISBN: 978-3-540-24372-4 (cit. on pp. 2, 10, 15).
- [147] Christian Prehofer. “Feature-Oriented Programming: A Fresh Look At Objects.” In: *Proceedings of the 11th European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer, 1997, pp. 419–443 (cit. on pp. 2, 11, 14, 15, 22, 63).
- [148] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S Foster, and Adam A Porter. “Using symbolic evaluation to understand behavior in configurable software systems.” In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA: ACM, 2010, pp. 445–454 (cit. on p. 167).
- [149] John Reppy and Aaron Turon. “Metaprogramming with Traits.” In: *Proceedings of the 21st European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer, 2007, pp. 373–398 (cit. on p. 15).
- [150] Márcio Ribeiro, Paulo Borba, and Christian Kästner. “Feature Maintenance with Emergent Interfaces.” In: *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 989–1000 (cit. on pp. 14, 35).
- [151] Martin P. Robillard and Gail C. Murphy. “Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies.” In: *Proceedings of the 24th International Conference on Software Engineering*. 2002, 406–416 (cit. on p. 22).

- [152] Iran Rodrigues, Márcio Ribeiro, Flávio Medeiros, Paulo Borba, Balduino Fonseca, and Rohit Gheyi. "Assessing Fine-grained Feature Dependencies." In: *Information and Software Technology* 78.C (2016), pp. 27–52 (cit. on pp. [63](#), [148](#)).
- [153] Chanchal Kumar Roy and James R Cordy. *A Survey on Software Clone Detection Research*. Tech. rep. 2007 (cit. on p. [3](#)).
- [154] Mazen Saleh and Hassan Gomaa. "Separation of Concerns in Software Product Line Engineering." In: *Proceedings of the Workshop on Modeling and Analysis of Concerns in Software*. New York, NY, USA: ACM, 2005, pp. 1–5 (cit. on p. [21](#)).
- [155] Alcemir Rodrigues Santos and Eduardo Santana de Almeida. "Do #Ifdef-based Variation Points Realize Feature Model Constraints?" In: *ACM SIGSOFT Software Engineering Notes* 40.6 (2015), pp. 1–5 (cit. on p. [16](#)).
- [156] Ina Schäfer, Lorenzo Bettini, and Ferruccio Damiani. "Compositional Type-checking for Delta-oriented Programming." In: *Proceedings of the 10th International Conference on Aspect-oriented Software Development*. New York, NY, USA: ACM, 2011, pp. 43–56 (cit. on p. [34](#)).
- [157] Ina Schäfer and Ferruccio Damiani. "Pure Delta-oriented Programming." In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2010, pp. 49–56 (cit. on pp. [2](#), [14](#), [15](#), [25](#), [60](#), [62](#), [164](#)).
- [158] Ina Schäfer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. "Delta-Oriented Programming of Software Product Lines." In: *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*. Berlin, Heidelberg: Springer, 2010, pp. 77–91 (cit. on pp. [2](#), [14](#), [15](#), [25](#), [68](#), [157](#), [158](#), [164](#)).
- [159] Ina Schäfer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. "Software Diversity: State of the Art and Perspectives." In: *International Journal on Software Tools for Technology Transfer* 14.5 (2012), pp. 477–495 (cit. on pp. [2](#), [10](#), [11](#), [14](#)).
- [160] Reimar Schröter, Norbert Siegmund, Thomas Thüm, and Gunter Saake. "Feature-context Interfaces: Tailored Programming Interfaces for Software Product Lines." In: *Proceedings of the 18th International Software Product Line Conference*. New York, NY, USA: ACM, 2014, pp. 102–111 (cit. on pp. [34](#), [73](#), [164](#)).

- [161] Michael Schulze, Jan Mauersberger, and Danilo Beuche. “Functional Safety and Variability: Can It Be Brought Together?” In: *Proceedings of the 17th International Software Product Line Conference*. New York, NY, USA: ACM, 2013, pp. 236–243 (cit. on p. 52).
- [162] Sandro Schulze. “Analysis and Removal of Code Clones in Software Product Lines.” PhD thesis. Otto-von-Guericke-Universität Magdeburg, 2013, pp. 1–163 (cit. on pp. 3, 11, 30, 31, 71).
- [163] Sandro Schulze, Sven Apel, and Christian Kästner. “Code Clones in Feature-oriented Software Product Lines.” In: *Proceedings of the 9th International Conference on Generative Programming and Component Engineering*. New York, NY, USA: ACM, 2010, pp. 103–112 (cit. on p. 76).
- [164] Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. “Does the Discipline of Preprocessor Annotations Matter?: A Controlled Experiment.” In: *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*. New York, NY, USA: ACM, 2013, pp. 65–74 (cit. on p. 35).
- [165] Sven Schuster and Sandro Schulze. “Object-oriented Design in Feature-oriented Programming.” In: *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2012, pp. 25–28 (cit. on p. 11).
- [166] Felix Schwägerl, Thomas Buchmann, and Bernhard Westfechtel. “SuperMod - A Model-Driven Tool that Combines Version Control and Software Product Line Engineering.” In: *Proceedings of the 10th International Conference on Software Paradigm Trends*. 2015, pp. 5–18 (cit. on p. 28).
- [167] Christoph Seidl, Ina Schäfer, and Uwe Aßmann. “Variability-aware Safety Analysis Using Delta Component Fault Diagrams.” In: *17th International Software Product Line Conference*. ACM, 2013, pp. 2–9 (cit. on pp. 52, 53, 55).
- [168] Christoph Seidl, Ina Schäfer, and Uwe Aßmann. “DeltaEcore - A Model-Based Delta Language Generation Framework.” In: *Modellierung*. GI, 2014, pp. 81–96 (cit. on pp. 27, 157, 158, 164).
- [169] Janet Siegmund. “Program Comprehension: Past, Present, and Future.” In: *Leaders of Tomorrow Symposium: Future of Software Engineering*. IEEE Computer Society, 2016, pp. 13–20 (cit. on p. 31).
- [170] Janet Siegmund and Jana Schumann. “Confounding Parameters on Program Comprehension: A Literature Survey.” In: *Empirical Software Engineering* 20.4 (2015), 1159—1192 (cit. on p. 32).

- [171] Janet Siegmund, Christian Kästner, Jörg Liebig, and Sven Apel. “Comparing Program Comprehension of Physically and Virtually Separated Concerns.” In: *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2012, pp. 17–24 (cit. on pp. 3, 32, 33, 36, 66, 93).
- [172] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. “Understanding Understanding Source Code with Functional Magnetic Resonance Imaging.” In: *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 378–389 (cit. on p. 32).
- [173] Charles Simonyi. *The death of computer languages, the birth of intentional programming*. Tech. rep. MSR-TR-95-52. Microsoft Research, 1995 (cit. on p. 97).
- [174] Charles Simonyi, Magnus Christerson, and Shane Clifford. “Intentional Software.” In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. New York, NY, USA: ACM, 2006, pp. 451–464 (cit. on p. 97).
- [175] Nieraj Singh, Celina Gibbs, and Yvonne Coady. “C-CLR: A Tool for Navigating Highly Configurable System Software.” In: *Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software*. New York, NY, USA: ACM, 2007 (cit. on pp. 28, 33).
- [176] Yannis Smaragdakis and Don Batory. “Mixin Layers: An Object-oriented Implementation Technique for Refinements and Collaboration-based Designs.” In: *ACM Transactions on Software Engineering and Methodology* 11.2 (2002), pp. 215–255 (cit. on p. 27).
- [177] Henry Spencer and Collyer Geoff. “#Ifdef Considered Harmful, or Portability Experience With C News.” In: *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 1992, pp. 185–197 (cit. on pp. 2, 14, 16, 32).
- [178] Ștefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wařowski. “Concepts, Operations, and Feasibility of a Projection-Based Variation Control System.” In: *IEEE International Conference on Software Maintenance and Evolution*. IEEE Computer Society, 2016, pp. 323–333 (cit. on pp. 3, 28, 68, 74, 162).
- [179] Thomas A Standish. “An Essay on Software Reuse.” In: *IEEE Transactions on Software Engineering* 10.5 (1984), pp. 494–497 (cit. on p. 31).



- [180] Hongyu Sun, Miriam Hauptman, and Robyn Lutz. "Integrating Product-Line Fault Tree Analysis into AADL Models." In: *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 15–22 (cit. on p. 52).
- [181] Tamás Szabó, Simon Alperovich, Markus Voelter, and Sebastian Erdweg. "An Extensible Framework for Variable-precision Data-flow Analyses in MPS." In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2016, pp. 870–875 (cit. on p. 152).
- [182] Peri Tarr, Harold Ossher, William Harrison, and Stanley M Sutton. "N Degrees of Separation: Multi-dimensional Separation of Concerns." In: *Proceedings of the 21st International Conference on Software Engineering*. New York, NY, USA: ACM, 1999, pp. 107–119 (cit. on p. 22).
- [183] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Dead or Alive: Finding Zombie Features in the Linux Kernel." In: *Proceedings of the First International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2009, pp. 81–86 (cit. on p. 32).
- [184] Sahil Thaker, Don Batory, David Kitchin, and William Cook. "Safe Composition of Product Lines." In: *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*. New York, NY, USA: ACM, 2007, pp. 95–104 (cit. on p. 63).
- [185] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. "Abstract Features in Feature Modeling." In: *Proceedings of the 15th International Software Product Line Conference*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 191–200 (cit. on p. 13).
- [186] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schäfer, 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 (cit. on pp. 63, 65, 168).
- [187] 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 (cit. on p. 164).
- [188] Rebecca Tiarks. "What Programmers Really Do - An Observational Study." In: *Softwaretechnik-Trends* 31.2 (2011), pp. 36–37 (cit. on p. 31).

- [189] Michael VanHilst and David Notkin. “Decoupling Change from Design.” In: *ACM SIGSOFT Software Engineering Notes* 21.6 (1996), pp. 58–69 (cit. on p. 36).
- [190] Michael VanHilst and David Notkin. “Using Role Components in Implement Collaboration-based Designs.” In: *ACM SIGPLAN Notices* 31.10 (1996), pp. 359–369 (cit. on p. 27).
- [191] W. Vesely, F. Goldberg, N. Roberts, and D. Haasl. *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission, NUREG-0492, 1981 (cit. on pp. 52, 53, 54).
- [192] Markus Voelter. “Generic Tools, Specific Languages.” PhD thesis. Delft University of Technology, 2014. ISBN: 978-94-6203-586-7 (cit. on p. 99).
- [193] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. “mbeddr: Instantiating a Language Workbench in the Embedded Software Domain.” In: *Automated Software Engineering* 20.3 (2013), pp. 339–390 (cit. on p. 68).
- [194] Markus Voelter, Arie van Deursen, Bernd Kolb, and Stephan Eberle. “Using C Language Extensions for Developing Embedded Software: A Case Study.” In: *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2015, pp. 655–674 (cit. on p. 68).
- [195] Markus Voelter, Tamás Szabó, Sascha Lisson, Bernd Kolb, Sebastian Erdweg, and Thorsten Berger. “Efficient Development of Consistent Projectional Editors Using Grammar Cells.” In: *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering*. New York, NY, USA: ACM, 2016, pp. 28–40 (cit. on pp. 39, 99, 101, 123, 158, 161).
- [196] Janis Voigtländer. “Bidirectionalization for free! (Pearl).” In: *ACM SIGPLAN Notices* 44.1 (2009), pp. 165–176 (cit. on p. 28).
- [197] Markus Völter. “Implementing Feature Variability for Models and Code with Projectional Language Workbenches.” In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*. New York, NY, USA: ACM, 2010, pp. 41–48 (cit. on pp. 22, 68, 164).
- [198] Markus Völter and Iris Groher. “Product Line Implementation using Aspect-Oriented and Model-Driven Software Development.” In: *Proceedings of the 11th International Software Product Line Conference*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 233–242 (cit. on pp. 2, 15).
- [199] Markus Völter and Sascha Lisson. “Supporting Diverse Notations in MPS’ Projectional Editor.” In: *Proceedings of the 2nd International Workshop on The Globalization of Modeling Languages*. 2014, pp. 7–16 (cit. on p. 50).

- [200] Markus Völter and Eelco Visser. “Product Line Engineering Using Domain-Specific Languages.” In: *Proceedings of the 15th International Software Product Line Conference*. 2011, pp. 70–79 (cit. on p. 164).
- [201] Markus Völter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. “mbeddr: An Extensible C-based Programming Language and IDE for Embedded Systems.” In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. New York, NY, USA: ACM, 2012 (cit. on pp. 22, 32, 39, 97, 164).
- [202] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelman, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Wachsmuth Guido. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. ISBN: 978-1-4812-1858-0 (cit. on pp. 4, 99).
- [203] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. “Towards User-Friendly Projectional Editors.” In: *Proceedings of the 7th International Conference on Software Language Engineering*. Springer, 2014, pp. 41–61 (cit. on pp. 38, 39, 69).
- [204] Eric Walkingshaw and Martin Erwig. “A Calculus for Modeling and Implementing Variation.” In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*. New York, NY, USA: ACM, 2012, pp. 132–140 (cit. on pp. 15, 162).
- [205] Eric Walkingshaw and Klaus Ostermann. “Projectional Editing of Variational Software.” In: *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*. New York, NY, USA: ACM, 2014, pp. 29–38 (cit. on pp. 3, 28, 162).
- [206] Bernhard Westfechtel, Bjørn P. Munch, and Reidar Conradi. “A Layered Architecture for Uniform Version Management.” In: *IEEE Transactions on Software Engineering* 27.12 (2001), pp. 1111–1133 (cit. on pp. 3, 28).
- [207] Tim Winkelmann, Jonathan Koscielny, Christoph Seidl, Sven Schuster, Ferruccio Damiani, and Ina Schäfer. “Parametric DeltaJ 1.5: Propagating Feature Attributes into Implementation Artifacts.” In: *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering*. CEUR-WS.org, 2016, pp. 40–54 (cit. on p. 74).
- [208] Hongyu Zhang and Stanislaw Jarzabek. “XVCL: A Mechanism for Handling Variants in Software Product Lines.” In: *Science of Computer Programming* 53.3 (2004), pp. 381–407 (cit. on p. 163).