



Mining software artefact variants for product line migration and analysis

Jabier Martinez

► To cite this version:

Jabier Martinez. Mining software artefact variants for product line migration and analysis. Software Engineering [cs.SE]. Université Pierre et Marie Curie - Paris VI; Université du Luxembourg, 2016. English. NNT : 2016PA066344 . tel-01477423

HAL Id: tel-01477423

<https://theses.hal.science/tel-01477423>

Submitted on 27 Feb 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE DE DOCTORAT DE
L'UNIVERSITÉ PIERRE ET MARIE CURIE
et
L'UNIVERSITÉ DU LUXEMBOURG

Spécialité
Informatique
EDITE de Paris
École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par
M. Jabier Martinez

Pour obtenir le grade de
DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE
et **DOCTEUR de l'UNIVERSITÉ DU LUXEMBOURG**

Sujet de la thèse :

**Exploration des variantes d'artefacts logiciels pour
une analyse et une migration vers des lignes de produits**

soutenue le 18 Octobre 2016

devant le jury composé de :

Dr. Jean-Marc Jézéquel, *rapporteur*
Professor, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes, France

Dr. Klaus Schmid, *rapporteur*
Professor, University of Hildesheim, Department of Computer Science, Hildesheim, Germany

Dr. Jacques Klein, *examineur*
Université du Luxembourg, Luxembourg

Dr. Pascal Poizat, *examineur*
Professor, Université Paris Ouest Nanterre La Défense et UPMC, Paris, France

Dr. Yves Le Traon, *co-directeur*
Professor, Université du Luxembourg, Luxembourg

Dr. Mikal Ziane, *co-directeur*
Maître de conférences HDR, Université Paris Descartes et UPMC, Paris, France

Dr. Tewfik Ziadi, *encadrant*
Université Pierre and Marie Curie, Paris, France

Mining Software Artefact Variants for Product Line Migration and Analysis

Abstract

Software Product Lines (SPLs) enable the derivation of a family of products based on variability management techniques. Inspired by the manufacturing industry, SPLs use feature configurations to satisfy different customer needs, along with reusable assets associated to the features, to allow systematic and planned reuse. SPLs are reported to have numerous benefits such as time-to-market reduction, productivity increase or product quality improvement. However, the barriers to adopt an SPL are equally numerous requiring a high up-front investment in domain analysis and implementation. In this context, to create variants, companies more commonly rely on ad-hoc reuse techniques such as copy-paste-modify.

Capitalizing on existing variants by extracting the common and varying elements is referred to as extractive approaches for SPL adoption. Extractive SPL adoption allows the migration from single-system development mentality to SPL practices. Several activities are involved to achieve this goal. Due to the complexity of artefact variants, feature identification is needed to analyse the domain variability. Also, to identify the associated implementation elements of the features, their location is needed as well. In addition, feature constraints should be identified to guarantee that customers are not able to select invalid feature combinations (e.g., one feature requires or excludes another). Then, the reusable assets associated to the feature should be constructed. And finally, to facilitate the communication among stakeholders, a comprehensive feature model need to be synthesized. While several approaches have been proposed for the above-mentioned activities, extractive SPL adoption remains challenging. A recurring barrier consists in the limitation of existing techniques to be used beyond the specific types of artefacts that they initially targeted, requiring inputs and providing outputs at different granularity levels and with different representations. Seamlessly address the activities within the same environment is a challenge by itself.

This dissertation presents a unified, generic and extensible framework for mining software artefact variants in the context of extractive SPL adoption. We describe both its principles and its realization in Bottom-Up Technologies for Reuse (BUT4Reuse). Special attention is paid to model-driven development scenarios. A unified process and representation would enable practitioners and researchers to empirically analyse and compare different techniques. Therefore, we also focus on benchmarks and in the analysis of variants, in particular, in benchmarking feature location techniques and in identifying families of variants in the wild for experimenting with feature identification techniques. We also present visualisation paradigms to support domain experts on feature naming during feature identification and to support on feature constraints discovery. Finally, we investigate and discuss the mining of artefact variants for SPL analysis once the SPL is already operational. Concretely, we present an approach to find relevant variants within the SPL configuration space guided by end user assessments.

ACKNOWLEDGEMENTS

Thanks. Thanks to my parents Chuchi and Josefa. *Gracias por el amor y apoyo incondicional, por vuestro trabajo duro que me dio la oportunidad y la libertad de elegir mi camino. No podré agradecer suficiente todo lo que habeis hecho por mi.*

Thanks to my brother Josu, I do not need to mention to which extent I appreciate the way you are. Thanks to my extended family and friends who, despite of the distance from my hometown Bilbao, were always close to me. Thanks Julia, for your encouragement and love.

Thanks Tewfik. Thanks for your never ending enthusiasm and vision. I gladly remember my visits to your office when I was still working in industry to discuss about research ideas. We accomplished those ideas and many others too. Thanks Mikal, I am grateful for your availability and support. Thanks to LiP6 and the nice people there.

Thanks Yves. Thanks for your continued support and encouragement. Thanks for creating an environment that allowed me to give the best of myself. I remember when I first met you to apply for a PhD student position. I always felt comfortable at Luxembourg and I have grown as a person and as a researcher within a team of great human and professional quality. Thanks Jacques, Tegawendé, Mike and Chris. Discussing with you was very enjoyable and prolific. Thanks to everybody in the team, including my officemates Anestis and Girum. Thanks to all. We spent uncountable hours and coffees together.

Thanks to the Luxembourg National Research Fund for the PhD grant. Thanks to the administrative staff of both universities. Thanks to my collaborators at the Luxembourg Institute of Science and Technology. Thanks to Gabriele. Thanks to the European Software Institute (Tecnalia) and Thales, places where I first got inspired to follow this research line.

Thanks to the organizers of the conferences I attended. Thanks to the constructive criticism of the anonymous reviewers. Finally, special thanks to the jury members for their time.

Jabier Martínez

Paris, France, August 2016

CONTENTS

List of abbreviations	xi
List of figures	xiii
List of tables	xvii
List of algorithms	xix
Contents	xix
1 General introduction	1
1.1 Context	2
1.1.1 Software product lines and extractive adoption	2
1.2 Overview of challenges and contributions	4
1.2.1 Challenges	4
1.2.2 Summary of contributions	6
1.2.3 Organization of the dissertation	7
 I Background and state of the art	 9
2 Background and definitions	11
2.1 Software reuse	12
2.1.1 The history of software reuse	12
2.1.2 Software product lines	12
2.1.3 It is not only about source code	17
2.2 Approaches for software product line adoption	18
2.2.1 Adopting a software product line	18
2.2.2 Extractive software product line adoption	18
2.3 Software product lines and end users	20
 3 Related work	 21
3.1 Mining artefact variants in extractive SPL adoption	22
3.1.1 Towards feature identification	22
3.1.2 Feature naming during feature identification	23
3.1.3 Feature location	24
3.1.4 Constraints discovery	25
3.1.5 Feature model synthesis	25
3.1.6 Reusable assets construction	26
3.2 The model variants scenario	26
3.3 Visualisation	28
3.4 Benchmarks and case studies	29
3.5 Generic and extensible frameworks in SPLE	30

3.6	SPL configuration spaces and end users	31
3.6.1	Interactive analysis of configuration spaces	31
3.6.2	Dealing with high subjectivity	32
II	Mining artefact variants for extractive SPL adoption	35
4	Generic and extensible extractive adoption of software product lines	37
4.1	Introduction	38
4.2	A Framework for extractive SPL adoption	39
4.2.1	Principles	39
4.2.2	Designing an adapter to benefit from the framework	41
4.2.3	Activities for extractive SPL adoption within the framework	43
4.3	Framework realization in BUT4Reuse	45
4.3.1	Genericity in BUT4Reuse through the adapters	46
4.3.2	Block identification	48
4.3.3	Feature identification	49
4.3.4	Feature location	50
4.3.5	Constraints discovery	53
4.3.6	Feature model synthesis	54
4.3.7	Reusable assets construction	55
4.3.8	Visualisations	55
4.4	Experiences and evaluation with the Eclipse case study	56
4.4.1	Design of the Eclipse adapter	57
4.4.2	Results and discussions	58
4.5	Limitations	61
4.6	Conclusions	62
5	Extraction of Model-based software product lines from model variants	63
5.1	Introduction	64
5.2	Extraction of Model-based Product Lines	64
5.3	Designing the Model Adapter	66
5.3.1	Elements identification: A meta-model independent approach	66
5.3.2	Structural dependencies identification	68
5.3.3	Similarity metric definition: Relying on extensible techniques	69
5.3.4	Reusable assets construction: Generating a CVL model	70
5.4	Experimental Assessment	73
5.4.1	BUT4Reuse settings for the case studies	73
5.4.2	ArgoUML case study	75
5.4.3	In-Flight Entertainment Systems case study	78
5.4.4	Discussions about MoVa2PL	80
5.5	Conclusion	81

III	Collecting artefact variants for study and benchmarking	83
6	Benchmark for feature location techniques using Eclipse variants	85
6.1	Introduction	86
6.2	The Eclipse family of integrated development environments	87
6.2.1	Tailored Eclipses for different development needs	87
6.2.2	Reasons to consider Eclipse for benchmarking	90
6.3	EFLBench: Eclipse Feature Location Benchmarking framework	91
6.3.1	Benchmark construction	91
6.3.2	Benchmark usage	92
6.4	Examples of EFLBench usage in Eclipse releases	93
6.5	Automatic and parametrizable generator of Eclipse variants	95
6.5.1	Strategies for the automatic selection of configurations	96
6.5.2	Results using automatic generation of variants	98
6.6	Conclusions	100
7	Feature identification in mined families of android applications	101
7.1	Introduction	102
7.2	Families of Android applications	103
7.2.1	An example of feature identification in an Android app family	103
7.3	Identifying families of Android apps in app markets	105
7.3.1	Implementation and preliminary results	106
7.4	Feature identification in selected families using BUT4Reuse	108
7.4.1	BUT4Reuse adapter for Android apps	108
7.4.2	Categorization of Android families after applying feature identification	109
7.5	Conclusions	112
IV	Assistance and visualisation in artefact variants analysis	113
8	Naming during feature identification with word clouds	115
8.1	Introduction	116
8.2	VariClouds approach	117
8.2.1	Word clouds and weighting factors	117
8.2.2	Retrieving the words through the adapters	117
8.3	Using VariClouds: Phases for the domain experts	119
8.3.1	Phase 1: Preparation of the word clouds	120
8.3.2	Phase 2: Block naming	123
8.4	Case studies and evaluation	125
8.4.1	Requirements of selected case studies	125
8.4.2	Quality of the word clouds	127
8.4.3	The benefits of summarization	129
8.5	Threats to validity	130
8.6	Conclusion	131

9	Feature constraints analysis with feature relations graphs	133
9.1	Introduction	134
9.2	Manual feature constraints discovery	135
9.3	Mining configurations to create feature relations graphs	136
9.3.1	Data abstraction and formulas	136
9.3.2	Graphical aspects	138
9.3.3	Interaction possibilities	142
9.3.4	Rejected alternatives	143
9.4	Electric Parking Brake case study	145
9.4.1	Introduction to the domain	145
9.4.2	Visualising the feature relations	146
9.4.3	Extension considering feature conditions	149
9.5	Conclusions	149
V	Configuration space analysis for estimating user assessments	151
10	End user assessments in software product lines: The HSPLRank approach	153
10.1	Introduction	154
10.2	Motivating scenarios for estimation and prediction	155
10.2.1	Subjectivity in SPL-based computer-generated art	155
10.2.2	An SPL for digital landscape paintings	156
10.2.3	Variability and user experience in UI design	158
10.3	The HSPLRank approach	159
10.3.1	Variant reduction	160
10.3.2	Variant assessment	161
10.3.3	Ranking computation	163
10.3.4	Confidence levels for ranking items	166
10.4	Conclusions	168
11	Software Product Line case studies for estimation and prediction	169
11.1	Introduction	170
11.2	Paintings case study	170
11.2.1	HSPLRank realization	171
11.2.2	Evaluation	175
11.3	Contact List case study	176
11.3.1	An SPL for Contact List design alternatives	176
11.3.2	HSPLRank realization	180
11.3.3	Evaluation	184
11.4	Discussion	188
11.5	Threats to validity	190
11.6	Conclusion	191

VI	Conclusions	193
12	Conclusions	195
12.1	Summary	196
12.2	Open research directions	197
	List of papers, tools & services	199
	Bibliography	203

LIST OF ABBREVIATIONS

AME	Atomic Model Element
AST	Abstract Syntax Tree
CVL	Common Variability Language
DSL	Domain-Specific Language
FM	Feature Model
HCI	Human-Computer Interaction
HSPL	Human-centered Software Product Line
IGA	Interactive Genetic Algorithm
IR	Information Retrieval
LOC	Lines Of Code
MDE	Model-Driven Engineering
MSPL	Model-based Software Product Line
SPL	Software Product Line
SPLE	Software Product Line Engineering
UI	User Interface

LIST OF FIGURES

1	General introduction	2
1.1	Organizational culture shift after extractive SPL adoption.	2
1.2	Software product line engineering processes: Domain and application engineering.	3
1.3	Overview of the context and challenges faced in this dissertation.	4
2	Background and definitions	12
2.1	Feature model example, feature diagram notation and example of configuration.	13
2.2	Negative and positive variability.	14
2.3	Preprocessor directives to implement negative variability using Antenna.	15
2.4	Compositional approach to implement positive variability using FeatureHouse.	16
2.5	Example of a car feature model	16
2.6	Relevant activities during extractive SPL adoption for leveraging artefact variants.	19
3	Related work	22
3.1	Feature model synthesis.	25
3.2	Common Variability Language.	27
4	Generic and extensible extractive adoption of software product lines	38
4.1	Artefact types and elements representation creation through the adapters.	40
4.2	Abstract Syntax Trees.	40
4.3	Design decisions for the images adapter.	42
4.4	Image variants and the result of applying the images adapter.	43
4.5	Extractive SPL adoption activities within our framework.	43
4.6	Block identification techniques.	48
4.7	Feature location results using two feature location techniques.	50
4.8	Feature location using latent semantic indexing.	51
4.9	Three different feature location techniques using SFS and term frequency.	52
4.10	Structural constraints discovery	53
4.11	Bars visualisation showing the blocks and the artefacts.	55
4.12	Heat map visualisation with the relation of blocks and features	56
4.13	Eclipse package decomposed in plugin and file elements.	59
5	Extraction of Model-based software product lines from model variants	64
5.1	UML model variants and the manual actions for their creation.	65
5.2	Bank UML model and excerpt of its atomic model elements.	67
5.3	Visualisation of the decomposition of three model variants.	69
5.4	Excerpts of the CVL models implementing the banking systems MSPL.	71
5.5	Excerpt of the CVL resolution model.	72
5.6	Identified blocks in the Bank system variants.	74
5.7	Structural constraints discovered among the features of the banking system.	74
5.8	Structurally invalid model as result of composing two features.	75
5.9	Structural constraints discovered for the ArgoUML case study.	77

5.10	Excerpt of the CVL Realization layer for the ArgoUML case study.	78
5.11	Operational analysis diagram of an In-Flight Entertainment system variant. .	79
5.12	In-Flight Entertainment system variant decomposition in elements.	79
6	Benchmark for feature location techniques using Eclipse variants	86
6.1	Eclipse Kepler SR2 packages and a mapping to their 437 features.	89
6.2	Feature dependencies in the Eclipse Kepler SR2 packages.	89
6.3	Plugin dependencies of the four plugins of the Eclipse CVS Client feature. . .	89
6.4	EFLBench: Eclipse package variants as benchmark for feature location. . . .	91
6.5	Automatic and parametrizable generation of Eclipse package variants. . . .	96
6.6	Different settings of the three strategies for selecting configurations.	97
7	Feature identification in mined families of android applications	102
7.1	Screenshots of six app variants of the 8684 family.	104
7.2	Pruned concept hierarchy visualisation of the 8684 family.	104
7.3	Steps of AppVariants app families discovery approach.	105
7.4	Meta-data of app variants of com.baidu	107
7.5	Boxplot on the number of variants per each family.	107
7.6	Distribution of the similarity of the variant pairs.	108
7.7	Pre-processing of the apk file	109
7.8	Screenshots of applications generated with Andromo.	110
7.9	Screenshots of two different apps with content-driven variability.	111
7.10	Screenshots of two apps with device-driven variability.	111
7.11	Screenshots of a mined app family that is only reusing libraries.	112
8	Naming during feature identification with word clouds	116
8.1	Examples of words retrieved from two excerpts of artefact variants.	118
8.2	VariClouds process with the preparation and block naming phases.	119
8.3	Vending machine variants and statechart diagrams of two variants.	120
8.4	Word cloud visualisations of vending machine variants.	121
8.5	Creation of a word cloud with VariClouds to summarize a set of elements. . .	121
8.6	Word clouds of all vending machine statechart variants.	122
8.7	Word cloud visualisations of identified blocks.	123
9	Feature constraints analysis with feature relations graphs	134
9.1	Configurations obtained as part of the extractive SPL adoption process. . . .	135
9.2	FRoGs visualisation for the Manual feature of the Car example.	139
9.3	Zones of a feature relations graph.	139
9.4	Graphical notation for formalized constraints.	141
9.5	Default color scheme for the nominal data of stakeholder perspectives. . . .	141
9.6	FRoGs visualisation for the PBSAutomatic feature	142
9.7	Rejected color scheme for the nominal data of stakeholder perspectives. . . .	144
9.8	Different relations regarding presence or absence of features.	144
9.9	Feature model for physical and logical components	145
9.10	FRoGs visualisation for the EffortSensor feature	148

9.11 FROGs visualisation for the PullerCable feature with activated filters	148
9.12 Visualising the impact of the selection of two features.	149
10 End user assessments in software product lines: The HSPLRank approach	154
10.1 Feature model of the landscape paintings.	157
10.2 Painting derivation process and examples.	158
10.3 Overview of HSPLRank.	160
10.4 Configuration space and viable space.	161
10.5 Variant assessment driven by the interactive genetic algorithm.	162
10.6 Similarity radius of a given configuration.	164
10.7 Different options to calculate the weight.	165
10.8 Neighbors similarity confidence.	166
10.9 Neighbors density confidence.	167
11 Software Product Line case studies for estimation and prediction	170
11.1 Displayed painting and voting device with a five points scale.	171
11.2 Mean absolute error for different radius values and weighting approaches. . .	173
11.3 Top 10 ranking items of the paintings case study.	174
11.4 Bottom 5 ranking items of the paintings case study.	174
11.5 Key paintings with high global confidence.	174
11.6 Contact List feature model.	177
11.7 Configuration and screenshot of its associated Contact List UI variant.	178
11.8 Screenshots of derived variants from the Contact List SPL.	178
11.9 Chromosome of an individual of the Contact List SPL.	181
11.10 LIST: Mean absolute error for radius values and weighting approaches. . . .	183
11.11 SnT: Mean absolute error for radius values and weighting approaches. . . .	183
11.12 Screenshots of relevant variants found using HSPLRank.	183
11.13 Results of the genetic algorithm evolution in the two organizations.	184
11.14 Generations progress in terms of mean score and diversity.	185
11.15 Comparing variant selection based on genetic algorithm and random.	185
11.16 Genotype population diversity.	186
11.17 Usability issues reported by users.	189
11.18 Relating middle parts with number of data set instances and scores.	189

LIST OF TABLES

2	Background and definitions	12
2.1	Examples of configurations for a feature model.	13
4	Generic and extensible extractive adoption of software product lines	38
4.1	List of available adapters for framework genericity assessment.	47
5	Extraction of Model-based software product lines from model variants	64
5.1	Elements of ArgoUML UML model variants.	76
5.2	Elements of the blocks identified in the ArgoUML case study.	76
5.3	Elements of IFE system model variants.	79
5.4	Elements of the features identified in the IFE system model variants.	80
6	Benchmark for feature location techniques using Eclipse variants	86
6.1	Eclipse CVS Client feature and its associated plugins.	88
6.2	Eclipse releases and their number of packages, features and plugins.	88
6.3	Precision and recall of the different feature location techniques.	94
6.4	Time performance in milliseconds for feature location.	95
6.5	FCA+SFS in generated packages using the percentage-based random strategy.	98
6.6	FCA+SFS in generated packages using the random strategy.	99
7	Feature identification in mined families of android applications	102
7.1	Top 10 families in number of variants.	108
8	Naming during feature identification with word clouds	116
8.1	Case studies for VariClouds evaluation	126
8.2	Evaluation of the quality of the word clouds.	127
9	Feature constraints analysis with feature relations graphs	134
9.1	Configuration space for the Car example.	136
9.2	Constraints identification while using feature relations graphs	137
9.3	Number of the identified constraints using FRoGs	147
11	Software Product Line case studies for estimation and prediction	170
11.1	Controlled assessment results.	175
11.2	Global Score Mean evaluation.	186

LIST OF ALGORITHMS

8	Naming during feature identification with word clouds	116
1	Automatic renaming of blocks.	124
11	Software Product Line case studies for estimation and prediction	170
2	Interactive genetic algorithm for data set creation in the Paintings case study.	172
3	Distance function between two painting configurations.	173
4	Interactive genetic algorithm for data set creation in the Contact List case study.	182

1

GENERAL INTRODUCTION

Contents

1.1	Context	2
1.1.1	Software product lines and extractive adoption	2
1.2	Overview of challenges and contributions	4
1.2.1	Challenges	4
1.2.2	Summary of contributions	6
1.2.3	Organization of the dissertation	7

1.1 Context

This dissertation focus on Software Product Lines (SPLs). Special emphasis is given to the challenges lying in the process of their adoption. Concretely, we concentrate in the case where a company can leverage existing variants to help during the adoption process that migrates from single-system development mentality to systematic and planned reuse [Nor04].

1.1.1 Software product lines and extractive adoption

SPLs are a mature paradigm for variability management in software engineering [NC⁺09, PBL05, ABKS13, vdLSR07]. They enable to define a family of product configurations and to later systematically generate the associated product variants. The inspiration for proposing this engineering practice is commonly attributed to the manufacturing industry where different predefined reusable components are usually combined to satisfy different customer needs. An SPL is formally defined as *“a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission, and that are developed from a common set of core assets in a prescribed way”* [NC⁺09].

Achieving large-scale productivity gains and improving time-to-market and product quality are some of the claimed benefits of SPLE [NC⁺09, vdLSR07]. Some acknowledged examples can be found in the SPL hall of fame [WCKK06] which reports commercially successful implementation of the SPL paradigm in companies from different domains ranging from avionics and automotive software, to printers, mobile phones or web-based systems.

Despite of these benefits, the barriers for SPL adoption are numerous [Nor04, Kru01]. Firstly, compared to single-system development, variability management implies a methodology that highly affects the life-cycle of the products as well as the processes and roles inside the company. Secondly, it has to be assumed that SPL adoption is a mid-long-term strategy given that preparing the reusable assets will be, at least in the first product deliveries, more costly than single-system development. It is often the case — around 50% according to a survey with industrial practitioners [BRN⁺13] — that companies already have existing products that were implemented using opportunistic ad-hoc reuse to quickly respond to different customer needs. We illustrate this case in the left side of Figure 1.1 where the goal is to change from next-contract vision to a strategic view of a business field through the adoption of an SPL [vdLSR07], as shown in the right side of the figure.

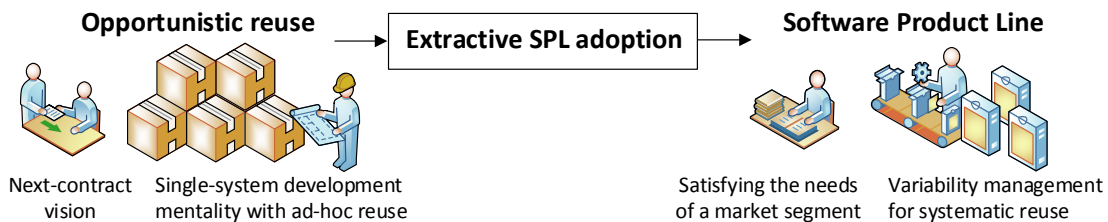


Figure 1.1: Organizational culture shift after extractive SPL adoption.

In SPL Engineering (SPLE), we can distinguish two main processes, *domain* and *application* engineering, illustrated in the horizontal layers of Figure 1.2. Domain engineering consists in analysing the domain for managing a feature model with the identified features in the product family and the constraints among them [KCH⁺90]. Then, a solution for this domain is implemented with the preparation of the reusable assets related to the features. In application engineering, customer requirements are analysed to create product configurations through the selection of features. Later, configurations are used to automatically derive the products with a mechanism that makes use of the reusable assets. To highlight the difference between domain and application engineering, the former is commonly called *development for reuse* while, the latter, *development with reuse*.

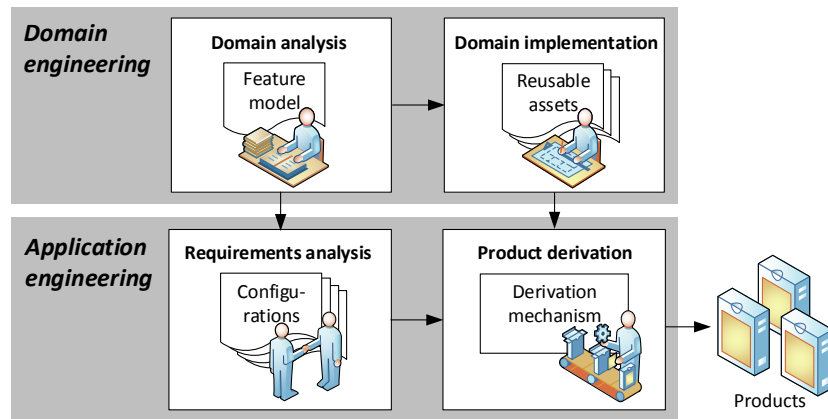


Figure 1.2: Software product line engineering processes: Domain and application engineering.

For the adoption of these SPLE practices, having legacy product variants can be seen as an enabler for a quick adoption using an extractive approach [Kru01]. However, in practice, extractive SPL adoption is still challenging. Mining the artefact variants to extract the feature model and the reusable assets gives rise to technical issues and decisions which need to be made by domain experts. Several activities are identified while mining artefact variants. If there is no complete knowledge of the common, alternative, and optional features within the product family, then the artefact variants should be leveraged for feature identification and naming. Also, we will need to identify the implementation elements associated to each feature. If features are known, feature location techniques can directly be used to obtain these associations. In addition, feature constraints discovery should be performed to guarantee the validity of certain feature combinations, for example, to avoid structurally invalid artefacts by combining incompatible features. Finally, the feature model conceptual structure need to be defined for efficient communication among stakeholders, and, certainly, reusable assets should be extracted to obtain an operative SPL.

1.2 Overview of challenges and contributions

This section presents the faced challenges, the summary of contributions and the organization of the dissertation.

1.2.1 Challenges

We faced four challenges in the context of SPL extraction and analysis. Three of them fall under the umbrella of extractive SPL adoption and the other is related to SPL analysis. We illustrate them in Figure 1.3 and we describe them below.

Harmonization: The lack of unification of extractive SPL adoption techniques

Practitioners lack end-to-end support for chaining the different steps of extractive SPL adoption. Addressing, within the same environment, the different activities starting from feature identification and location up to actually extracting the SPL, is a challenge by itself. Each approach requires inputs and provides outputs at different granularity levels and with different formats complicating the integration of approaches in a unified process. In addition, the proposed approaches are often related to specific types of software artefacts. There is a large body of algorithms and techniques for achieving the different objectives in SPL adoption (e.g., [RC13a, RC12a, YPZ09, LAG⁺14, SSD13, ASH⁺13c, ZHP⁺14, RC13b, AV14, LMSM10, LHGB⁺12, CSW08, HLE13, DDH⁺13, TDAJ16, SRA⁺14, HLE13, BABBN15, BBNAB14, IRW16]). Unfortunately, because the implementation of such algorithms is often specific to a given artefact type, re-adapting it for another type of artefacts is often a barrier. There is however an opportunity to reuse the principles guiding these existing techniques for other artefacts. When such algorithms are underlying in a framework, they could be transparently used in different scenarios. If there exists a set of principles for building a framework to integrate various algorithms and to support different artefact types this challenge can be overcome.

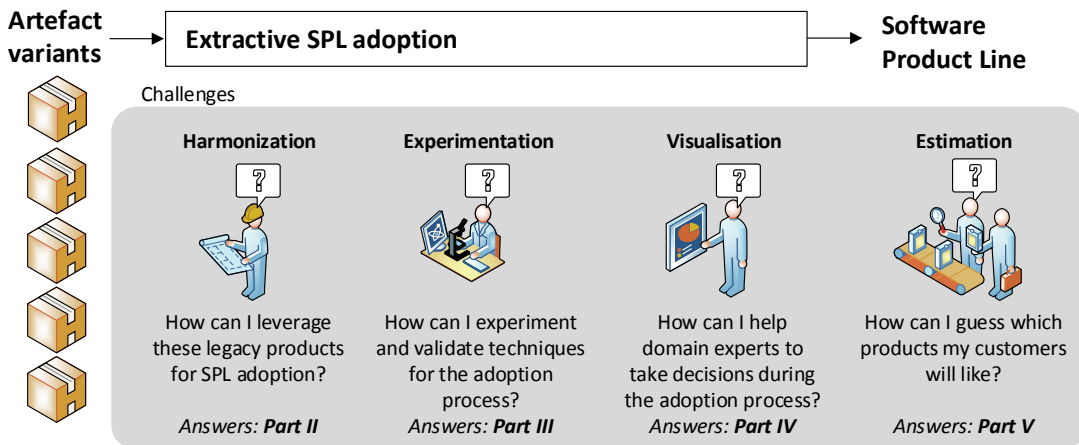


Figure 1.3: Overview of the context and challenges faced in this dissertation: Harmonization, experimentation and visualisation in extractive SPL adoption, and estimation of user assessments within the possible SPL products.

Experimentation: The need for case studies and benchmarks for extractive SPL adoption techniques

It is commonly accepted that high-impact research in software engineering requires assessment in realistic, non-trivial, comparable, and reproducible settings [SEH03]. In the extractive SPL adoption context, which shows a high proliferation of techniques, non-confidential feature-based artefact variants are usually unavailable. If available, establishing a representative ground truth is a challenging and debatable subject. Artefact variants created “in the lab”, such as variants directly derived from an SPL which is used as ground truth, might provide findings that are not generalisable to software development. The challenge is to continue the search of case studies “in the wild” to draw more representative conclusions about the different techniques. In this dissertation we have concentrated on enabling intensive experimentation on feature identification and location techniques.

Visualisation: The lack of support for domain experts during the extractive SPL adoption process

Visualisation reduces the complexity of comprehension tasks and helps to get insights and make decisions on a tackled problem [CMS99]. In extractive SPL adoption activities, the proposed approaches in the SPL literature are not focused on providing and validating appropriate visualisation support for domain experts. Mining artefact variants for SPL adoption is an arduous process and it is very optimistic to envision it as a fully automatic approach that does not require human stakeholders’ decision-making skills. Concretely, we tackle two visualisation challenges. The first one assists in naming during feature identification. Feature identification techniques that analyse artefact variants do not have a visualisation paradigm for suggesting feature names. The second challenge, once the features are identified, consists on designing a visualisation for feature constraints discovery in order to refine the envisaged feature model.

Estimation: The need to understand and foresee the end users’ expectations regarding the SPL products

Once the SPL is operational in a company, independently of whether the SPL was extracted or engineered from scratch, there is a need to satisfy the end users of the software products. This implies the exploration of different design alternatives. Exploring alternatives is specially relevant when considering products that have Human Computer Interaction (HCI) components. Variability management, as proposed in SPLE, enables to express and derive the different alternatives. In SPLs deriving human-centered products, there are important barriers to understand users’ perception of the products. First, the users cannot assess all possible products as the possibilities can be prohibitively large. Also, human assessments are subjective by nature. In addition, getting end user assessment (e.g., usability tests) is resource expensive. These facts make the selection of the more adequate products for the customers challenging.

1.2.2 Summary of contributions

The *harmonization*, *experimentation* and *visualisation* challenges are addressed in this dissertation by providing a framework and a process for mining and analysing existing products. That means that all of them are seamlessly integrated in the same conceptual and technical environment called BUT4Reuse.

Harmonization:

- **Bottom-Up Technologies for Reuse (BUT4Reuse):** A unified and extensible framework for extractive SPL adoption supporting several artefact types.
- **Model Variants to Product Line (MoVa2PL):** A complete example of using BUT4Reuse for extractive SPL adoption in the case of model variants.

Experimentation:

- **Eclipse Feature Location Benchmark (EFLBench):** A benchmark for intensive evaluation of feature location techniques in artefact variants.
- **Android Application families identification (AppVariants):** An approach for discovering families of Android applications in large repositories in order to experiment with feature identification techniques.

Visualisation:

- **Variability word Clouds (VariClouds):** A visualisation paradigm to support domain experts in feature naming during feature identification.
- **Feature Relations Graphs (FRoGs):** A visualisation paradigm to analyse feature relations and identify feature constraints.

The *estimation* challenge is addressed by providing an approach based on analysing the configuration space.

Estimation:

- **Human-centered SPL Rank (HSPLRank):** An approach to rank configurations based on the expected acceptance of the products by a profile of end users.

1.2.3 Organization of the dissertation

The dissertation is organized in six parts. The first part presents background information. The next four parts describe the contributions of this thesis, and the last part presents concluding remarks and outlines open research directions.

Part I: Background and state of the art

The technical background is presented in Chapter 2 and related work from the state of the art is presented in Chapter 3.

Part II: Mining artefact variants for extractive SPL adoption

Harmonization: Chapter 4 introduces the BUT4Reuse unifying framework for extractive SPL adoption. Its principles, genericity for supporting different artefact types and extensibility options are presented. This framework is the basis on which Part III and Part IV are built. Chapter 5 focuses on reporting the complete realization, usage and validation of the framework in model-driven development scenarios with model variants (MoVa2PL).

Part III: Collecting artefact variants for study and benchmarking

Experimentation: Chapter 6 presents the rationale and interest of using variants of the Eclipse plugin-based system to benchmark feature location techniques in artefact variants (EFLBench). Chapter 7 presents the mining of Android application repositories in the search for families of mobile applications to experiment with feature identification techniques (AppVariants).

Part IV: Assistance and visualisation in artefact variants analysis

Visualisation: This part presents two contributions regarding visualisation in extractive SPL adoption. Chapter 8 presents a visualisation paradigm to support feature naming during feature identification using word clouds (VariClouds). Chapter 9 presents a novel visualisation paradigm for feature relations analysis helping with constraints discovery (FRoGs).

Part V: Configuration space analysis for estimating user assessments

Estimation: Chapter 10 presents the principles and steps of a semi-automatic approach to rank the possible configurations of a given SPL regarding the expected acceptance of the products by a profile of end users (HSPLRank). Then, Chapter 11 presents and discusses two case studies conducted in two different scenarios.

Part VI: Conclusions

Chapter 12 presents the conclusions and outlines open research directions.

Part I

BACKGROUND
AND STATE OF THE ART

2

BACKGROUND AND DEFINITIONS

Contents

2.1	Software reuse	12
2.1.1	The history of software reuse	12
2.1.2	Software product lines	12
2.1.3	It is not only about source code	17
2.2	Approaches for software product line adoption	18
2.2.1	Adopting a software product line	18
2.2.2	Extractive software product line adoption	18
2.3	Software product lines and end users	20

2.1 Software reuse

Software reuse is the process of creating new software by reusing pieces of existing software rather than from scratch [Kru92]. It may be performed simply for convenience, as in the cases of the opportunistic *copy-paste-modify* approach and the project *clone-and-own* technique, or it may represent more thoughtful solutions to complex engineering problems, as in the case of Software Product Line Engineering (SPLE) [NC⁺09, PBL05, ABKS13, vdLSR07].

2.1.1 The history of software reuse

It is natural for humans to apply or adapt a known solution to similar problems. In software engineering, the analysis and continuous improvement of software reuse paradigms were of utmost importance for productivity and quality gains in the development of software. It is considered that this research started in the late 1960s mainly by proposing the reuse of subroutines and presenting the potential benefits of establishing a sub-industry and market for software components [McI68]. Nowadays, reusing methods, libraries or off-the-shelf components in software development is familiar to any software engineering practitioner and it had become part of our daily activities.

In the development of single systems, opportunistic copy-paste-modify is also an ad-hoc reuse technique massively used. This ad-hoc technique provides very short term productivity gains, even if potential clones introduced by this practice are known to have a cost in maintenance [JDHW09]. We are assisting also to an increasing practice of the clone-and-own reuse paradigm (also known as fork-and-own) mainly because of the vast proliferation of public software repositories. These repositories allow us to take entire software projects and modify them to fit our needs. Other advanced reuse techniques that we benefit are the services reuse (e.g., Software as a Service) or generators (e.g., transformations in Model-Driven Development). However, already in the mid 1970s the software engineering community started to discuss about program families for the realization of systematic, planned and strategic reuse [Par76]. Inspired by the manufacturing industry, mass customization started to be a reality in software engineering. This fact gave rise to the formalization in 1990 of a Feature-Oriented Domain Analysis method (FODA) which is considered an important milestone in the history of software reuse [KCH⁺90].

2.1.2 Software product lines

Software Product Lines (SPLs) were presented in Section 1.1 and illustrated in Figure 1.2. With an SPL, a company can benefit from an automatic derivation of a family of products through configurations based on customer requirements. This is possible after the analysis of the domain and its implementation through reusable assets. We provide in this section more details about these processes.

Feature models and configurations: Features are the prime entities used to distinguish the individual products of an SPL [BLR⁺15]. In the mentioned FODA method, the definition of feature was established: A feature is defined as “*a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems*” [KCH⁺90]. In this context, feature models (FM) are widely used in SPLE to describe both variability and commonalities in a family of product variants [BSRC10]. A given FM is a hierarchical decomposition of features including the constraints among them. Constraints are formalized in the FM to guarantee the correctness of valid products. Also, even if a product might be structurally valid, a feature combination can be semantically invalid in the targeted domain requiring the formalization of constraints to capture this domain knowledge.

The feature diagram notation is shown in Figure 2.1a regarding an illustrative and simplified example of an electronic shop. The **E-Shop** FM consists of a mandatory feature **Catalogue**, two possible **Payment** methods from which one or both could be selected, an alternative of two **Security** levels and an optional **Search** feature. Apart from the implicit constraints defined through the hierarchy (e.g., **BankTransfer** requires **Payment**), cross-tree constraints can be defined (e.g., **CreditCard** requires a **High** level of security) [BSRC10].

Given a FM, a configuration is a selection of features that satisfy all the constraints. Figure 2.1b shows a configuration example where **CreditCard** and **High** security are selected as well as the mandatory features. On the contrary, **BankTransfer**, **Standard** security and **Search** are not selected. Table 2.1 shows eight different configurations of the **E-Shop** FM. **Conf 1** in the table corresponds to the configuration in Figure 2.1b. If we consider the existence of an SPL

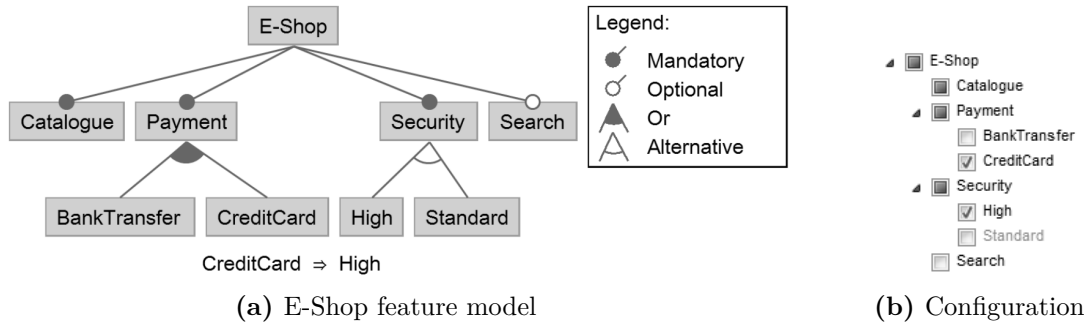


Figure 2.1: Feature model example of an electronic shop with feature diagram notation and a configuration example with selected features.

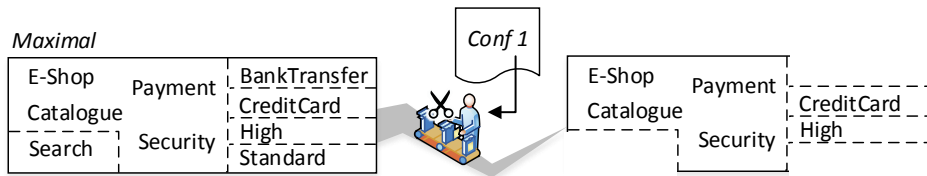
Table 2.1: Configurations of the electronic shop feature model. The eight configurations represent the configuration space given that any other configuration will not satisfy the feature constraints.

	Catalogue	Payment	Security	BankTransfer Payment	CreditCard Payment	High Security	Standard Security	Search
Conf 1	✓	✓	✓		✓	✓		
Conf 2	✓	✓	✓		✓	✓		✓
Conf 3	✓	✓	✓	✓			✓	
Conf 4	✓	✓	✓	✓			✓	✓
Conf 5	✓	✓	✓	✓		✓		
Conf 6	✓	✓	✓	✓		✓		✓
Conf 7	✓	✓	✓	✓	✓	✓		
Conf 8	✓	✓	✓	✓	✓	✓		✓

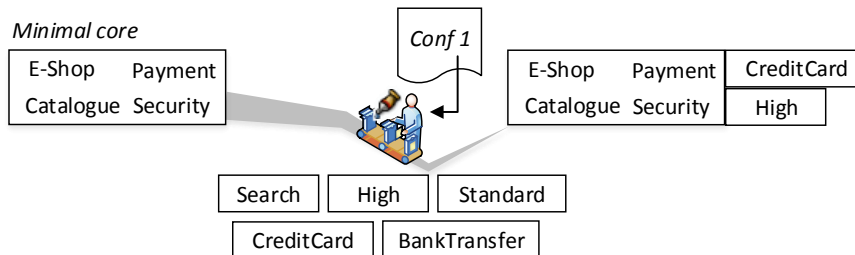
for the domain represented in this FM, the product variant derived from **Conf 1** will allow the end user to use the **CreditCard** and, for example, insert the credit card information. However, in the variant derived from **Conf 3**, the user will not have the possibility to pay by credit card.

The eight configurations presented in Table 2.1 purposely represent an exhaustive enumeration of all the possible valid configurations. This set containing all the possible configurations from a FM is referred to as the *configuration space*. The configuration space can be prohibitively large. For example, researchers have reported the challenges of dealing with the more than eight thousand features of the Linux kernel [STE⁺10], or the more than five hundred possible configurations of an SPL for elevators [CHSL11]. If we consider n optional and independent features, the configuration space consists of 2^n possible configurations. For instance, with n equal to 33 we could have a different configuration for every human alive today.

Product derivation: If we focus on the domain analysis, features are only symbols or labels [CA05], however, a feature is an abstraction that should usually have a counterpart or an impact in the SPL derivation process. Therefore, the reusable assets are the implementation of these abstractions. In order to derive products from feature configurations, *negative* and *positive* variability are the two main approaches conditioning the nature of the reusable assets [VG07]. Figure 2.2a illustrates negative variability using the **E-Shop** example. In negative variability, a product is derived by removing the parts associated to the non-selected features from a “maximal” system containing all the features (also known as 150%, single-copy or single-base SPL representation [RC12a, RC13a], or “overall” system [VG07]). In positive variability, illustrated in Figure 2.2b, the derivation process starts with a minimal core of the system, and then, a mechanism is able to add the parts associated to the selected features.



(a) Negative variability. A maximal product is used to remove the non-selected features of a configuration.



(b) Positive variability. A minimal core is used to add the selected features of a configuration.

Figure 2.2: Negative and positive variability are two approaches to implement SPL product derivation from feature configurations.

Examples of negative variability mechanisms are *annotative* approaches. For instance, preprocessor directives are used in source code artefacts to exclude source code fragments according to the selected features. Figure 2.3 shows an illustrative example of annotations using Antenna [PYW10] in a Java class managing the E-Shop user interface (UI). The source code is annotated between `#if` and `#endif` clauses. `#if Search` indicates the source code that should be included in case that **Search** is selected in the configuration. On the right side, we present the result of a derivation where **Search** is not selected and, therefore, the annotated source code that implemented **Search** is excluded. Another example of concrete annotative technique is Javapp which is used in the ArgoUML SPL [CVF11a] to annotate 31% of the total number of lines of code (LOC) of ArgoUML. Concretely, close to forty thousand LOC were annotated using preprocessor directives. Also, in the C code of the Linux kernel, KConfig annotations [SS08] are used to manage this huge SPL.

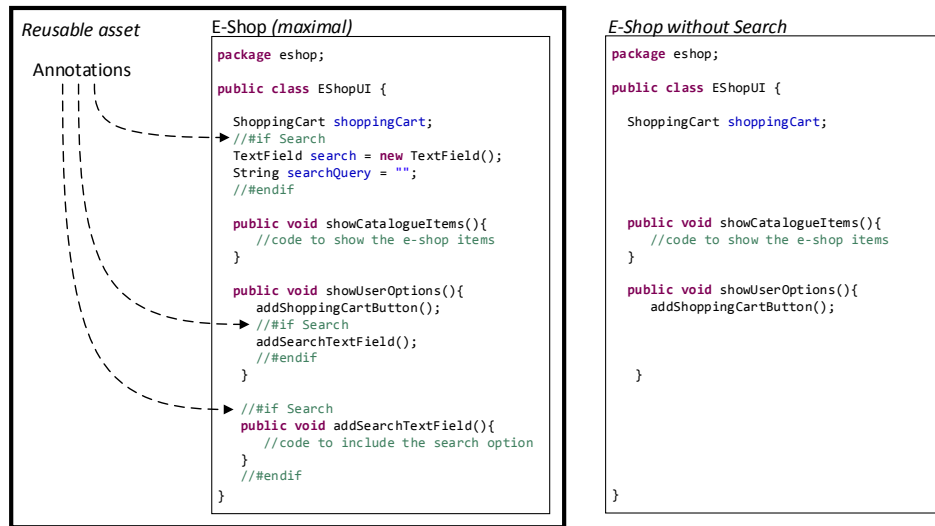


Figure 2.3: Example of preprocessor directives to implement negative variability using Antenna.

Regarding positive variability, *compositional* approaches enable the addition of implementation fragments in specified places of a system. A relevant example is FeatureHouse which is based on source code superimposition and merge [AKL09]. Figure 2.4 shows the same Java class regarding the E-Shop UI but using FeatureHouse instead of Antenna. With the compositional approach we define two separated reusable assets that are composed during derivation when **Search** is selected.

Many technical solutions have been proposed dealing with the SPL derivation mechanism. We can highlight generative programming [CE00], feature-oriented programming [Pre01, BSR04], aspect-oriented programming [KLM⁺97] or delta-oriented programming [SBB⁺10]. In addition, it is also possible to combine compositional with annotative techniques, therefore, hybrid approaches can support simultaneously negative and positive variability (e.g., implementations of the Common Variability Language (CVL) [HWC13, SIN15, HØ14]).

Soft constraints: Apart from the feature constraints presented before, there are other type known as *soft constraints* whose violation does not lead to invalid configurations [CSW08]. They are often used with the objective of warning the user, providing suggestions during the

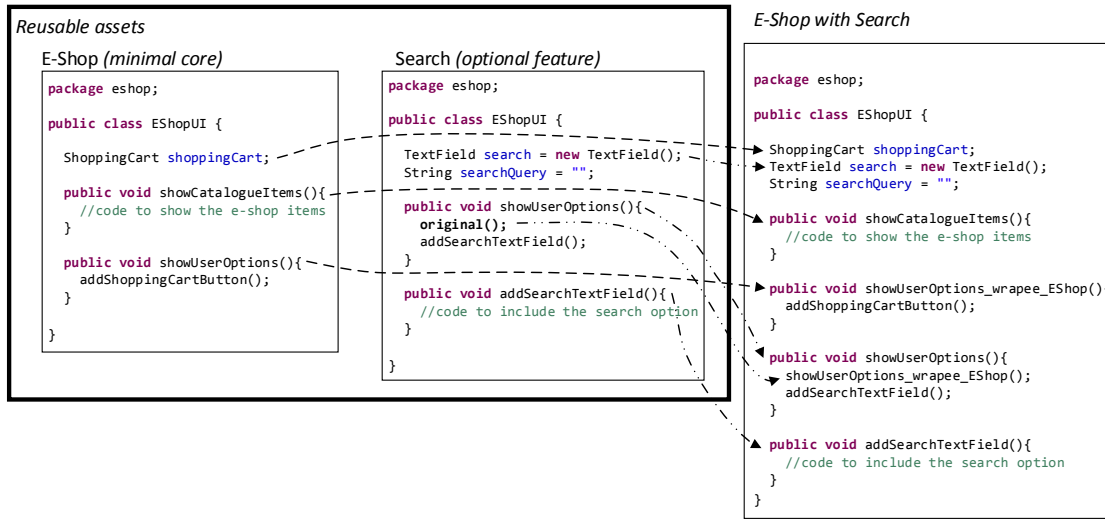


Figure 2.4: Example of compositional approach to implement positive variability using FeatureHouse.

configuration process [CSW08] or as a way to capture domain knowledge related to trends in the selection of features. Therefore, the existence of these soft constraints is motivated by the *suitability* of the selection of features in product configurations.

We can find many references regarding soft constraints in the literature. Soft constraints are introduced in feature modeling to help stakeholders in the configuration process providing advices in the selection of features or feature combinations [BM11]. In this way, *hints* constraints were introduced in order to suggest features during this decision-making process [SRP03, BHP03]. We can also find their opposite, the *hinders* constraints [BHP03]. They serve as a way to formalize the positive or negative influence of a given feature combination. The commercial feature modeling tool *pure::variants* has support for soft constraints through the *encourages* and *discourages* constraints [Beu10]. All these works indirectly introduced the intuition that soft constraints describe a certain level of uncertainty about the quality of the result of some feature combination.

Figure 2.5 presents an illustrative Car FM [CSW08]. The hierarchy of features includes mandatory features (Gear, Car), optional features (DriveByWire, ForNorthAmerica), alternative features (Manual, Automatic), and a cross-tree constraint ($\text{DriveByWire} \Rightarrow \text{Automatic}$). Based on domain expert knowledge or on previous experiences (e.g., existing configurations

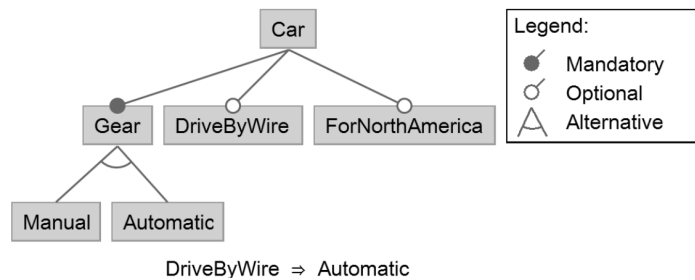



Figure 2.5: Example of a car feature model [CSW08].

from previous customers), selecting the feature `ForNorthAmerica` could suggest the selection of `Automatic` gear. In this dissertation we use a notation for soft constraints consisting in encapsulating the constraint inside a *soft* constructor. This way *encourages* and *discourages* can be formalized as follows:

- *encourages*: $\text{soft}(A \Rightarrow B)$. Feature A encourages the presence of feature B.
- *discourages*: $\text{soft}(A \Rightarrow \neg B)$. Feature A discourages the presence of feature B.

Using this notation, a domain expert could explicitly formalize the above-mentioned soft constraint (i.e., $\text{soft}(\text{ForNorthAmerica} \Rightarrow \text{Automatic})$). Therefore, soft constraints enable the capture of this relevant SPL domain knowledge during domain analysis.


 Soft constraints are relevant entities in Chapter 9 where we propose a visualisation for domain experts to discover constraints including soft constraints. Also, in Part V, they play an important role in a process to search optimal configurations where we suggest the formalization of soft constraints by domain experts to narrow the configuration space.

2.1.3 It is not only about source code

The potential of reuse is not only restricted to source code, as already suggested in the 1980s [Fre83]. Most of the attention is paid to source code reuse but documentation, designs, models or components are examples of software artefacts that are being reused too.

Many works targeting different artefact types have been proposed in software reuse research to identify similar fragments by comparing them. For example, clone detection has been proposed not only for source code [RC07], but also for other artefact types such as models [DHJ⁺08], graphs [PNN⁺09], dataflow languages [GKHB10] or DSLs implemented under the executable metamodeling paradigm [MGC⁺16] to name a few.

Derivation mechanisms for SPLE have been also proposed supporting different artefact types. In the modeling domain, we can find examples of annotative approaches such as in UML class models [ZJ06, CA05], sequence models [ZJ06], activity models [CA05] or statechart models [RC12a]. Another generic approach for dealing with models is CVL [HWC13] where a base model of a given domain-specific language (DSL) can be modified by removing or adding model fragments. There are plenty of specific techniques such as for model transformations reuse [CFSS16].

 Part II deals with the challenge of proposing a generic approach in mining artefact variants that could support different artefact types. Within this part, Chapter 4 discusses the principles guiding this generic approach and presents how diverse artefact types are supported. Chapter 5 presents all the details of the use of this generic approach in the case of models.

2.2 Approaches for software product line adoption

As mentioned in the introduction, SPL adoption is defined as the process for migrating from some form of developing software-intensive systems with a single-system mentality to developing them as an SPL [Nor04]. This section presents the approaches of SPL adoption.

2.2.1 Adopting a software product line

The SPL adoption strategy highly depends on the company scenario [Nor04, Kru01]. The *adoption factory pattern* is a generic adoption roadmap describing several practice areas [Nor04]. However, it is intended to be instantiated and customized for each case. Despite that there is no predefined path that works in any situation, Krueger distinguished three categories of SPL adoption strategies [Kru01]:

- *Proactive*: An SPL is designed taking into account the current and future customer needs. The organization proactively identifies and analyses the complete set of envisioned features and create their corresponding reusable assets.
- *Reactive*: In contrast to the proactive approach where the whole SPL aims to be created before starting the production, the reactive approach aims to create a minimal operative SPL, and later incrementally extend it to adapt to new customer needs. In terms of effort, the reactive approach requires less initial effort than the proactive one.
- *Extractive*: Existing product variants are leveraged to identify the features and the reusable assets to create the SPL. Mining existing assets is a practice area for establishing SPL production capability and operating the SPL [Nor04]. Notably, existing assets are project entry points that can be reused to seed, enrich, or augment the different building blocks needed to adopt an SPL [BFK⁺99].

Given that this dissertation is mainly focused on mining artefact variants, we pay special attention to the extractive approach .

2.2.2 Extractive software product line adoption

The extractive SPL adoption approach is compatible with both proactive and reactive approaches. For the proactive approach, the features in the existing variants can be leveraged while the envisioned ones will have to be created from scratch. For the reactive approach, the company can use an extractive approach to create an SPL that focuses on exactly the same products than the existing artefact variants, or they can only consider a subset of the features of the variants (e.g., those with higher payoff).

Figure 2.6 illustrates important activities in extractive SPL adoption. It is worth mentioning that we have concentrated in technical activities relevant for this dissertation, therefore we

are not considering other important factors in SPL adoption such as organizational change [Bos01], economic models [ABS09] or advanced scoping [Sch00].

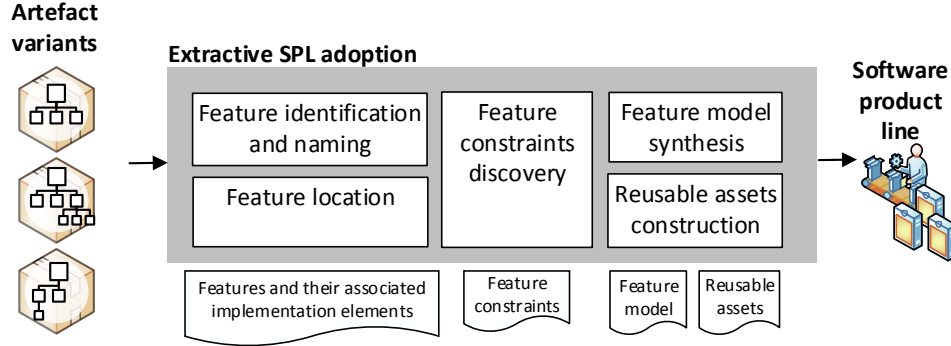


Figure 2.6: Relevant activities during extractive SPL adoption for leveraging artefact variants.

Given the complexity of the products, a complete upfront knowledge of the existing features throughout the artefact variants is not always available. Several domains of expertise are often required to build a product and different stakeholders are responsible for different functionalities. In this context, we can assume that domain knowledge about the features of legacy variants is scattered across the organization. In some cases it may not be properly documented or in a worse scenario, part of the knowledge could even be lost.

In an scenario of incomplete information, **feature identification** should be performed. As discussed by Northrop [Nor04], “*if an organization will rely heavily on legacy assets, it should begin the inventorying part of the mining existing assets*”, a practice area which consists in “*creating an inventory of candidate legacy components together with a list of the relevant characteristics of those components*” [NC⁺09]. In feature identification, we aim to obtain the list of features within the scope of the input variants as well as the implementation elements related to each feature. During the identification of features there is the **feature naming** sub-activity to define the names that will be used in the FM. In another scenario, if the features are known in advance, **feature location** can be performed directly to identify the implementation elements associated to each feature.


The activity **feature constraints discovery** is important to guarantee the validity of configurations in the envisioned FM. Then, the activity **feature model synthesis** consists in defining the structure of the FM to be as comprehensible for the SPL stakeholders as possible. This is because the same configuration space can be expressed in very different ways in a FM (e.g., features hierarchy). Then, the implementation elements associated to each feature obtained during feature identification or location, are important for the **reusable assets construction** which prepares the assets targeting a predefined derivation mechanism.

✎ Part II, III and IV deal with extractive SPL adoption proposing respectively: a generic and extensible framework, support for the experimentation of techniques in this research domain, and assistance to domain experts through visualisation and interaction paradigms.

2.3 Software product lines and end users

In many application domains, and regardless of the approach used for SPL adoption, SPLs derive products that end users should interact with. In SPLE scenarios intensively based on Human Computer Interaction (HCI) components, end users are key stakeholders. In general, HCI plays an increasingly important role in the success of software products. Historically, there are several milestones that have drastically changed the software engineering field regarding its relation with users: We can go back to the arrival of personal computers in the 1960s, the huge expansion of the Internet in the mid-1990s, the mass adoption of smartphones in the 2000s and, more recently, the promises of a globalization of ambient intelligence, internet of things, cloud-based workspaces, human-robot interaction or cyber-physical systems with humans in the loop. Computing has progressively evolved towards personalized customization and initiating a software engineering project increasingly requires to account for the human aspects of the potential customers.

Human centered SPLs (HSPL) are the inevitable confluence of the HCI and SPL engineering fields. From a human centered development perspective, there is the need to satisfy the end user of a software product, and this implies the exploration of different design alternatives. Variability management, as proposed in SPLE, enables the definition and derivation of the different alternatives. In HSPLs, three important barriers for understanding users perception of the product family variants are found: 1) It is not feasible that users assess all the possible variants in large configuration spaces, 2) human assessments are subjective by nature and 3) getting user assessment is resource expensive so there is an interest in minimizing it.

 Part V focuses on this challenge by proposing an approach to rank and identify the most appropriate configurations in an HSPL within the boundaries of its configuration space.

3

RELATED WORK

Contents

3.1	Mining artefact variants in extractive SPL adoption	22
3.1.1	Towards feature identification	22
3.1.2	Feature naming during feature identification	23
3.1.3	Feature location	24
3.1.4	Constraints discovery	25
3.1.5	Feature model synthesis	25
3.1.6	Reusable assets construction	26
3.2	The model variants scenario	26
3.3	Visualisation	28
3.4	Benchmarks and case studies	29
3.5	Generic and extensible frameworks in SPLE	30
3.6	SPL configuration spaces and end users	31
3.6.1	Interactive analysis of configuration spaces	31
3.6.2	Dealing with high subjectivity	32

3.1 Mining artefact variants in extractive SPL adoption


We present related work about the extractive SPL adoption activities described in Section 2.2.2: Feature identification, naming, feature location, constraints discovery, feature model synthesis and reusable assets construction.

3.1.1 Towards feature identification

The assumption that no domain knowledge is available about the features is perhaps too pessimistic and is thus limited to a few scenarios. However, it is also too idealistic to assume that an exhaustive list of the features and their associated implementation elements can be easily elicited from domain knowledge. Liu *et al.* and Kästner *et al.* among others proposed to identify feature information from a single product [LBL06, KDO14]. There are SPL adoption scenarios where the SPL wants to be extracted from a single product by separating its features. However, in this dissertation we concentrate on the case of several artefact variants. To distinguish features and their associated elements, researchers have proposed to analyse and compare artefact variants for the identification of their common and variable parts [YPZ09, ASH⁺13c, ZHP⁺14, RC12a, FLLE14]. We refer to each of such distinguishable parts as a *block*. A block is a set of implementation elements of the artefact variants that are relevant for the targeted mining task. Examples of existing techniques to identify blocks are based on static analysis, dynamic analysis or information retrieval techniques [AV14]. Independently of the technique or artefact type, a block is an intermediary abstraction representing a candidate set of elements that might implement a feature.

In this dissertation we will use the term block but, in the literature, we can find the same concept with different names. Rubin *et al.* call them *parts*, *regions*, or *diff-sets* alluding to the technique used to retrieve them [RC12a]. Other example of generic names are *modules* by Méndez-Acuña *et al.* [MGC⁺16] or *clusters* by Yang *et al.* [YPZ09] and Araar *et al.* [AS16]. Other employed terminology is less generic and they specifically refers to the concrete artefact types that they are dealing with. Linsbauer *et al.* [LAG⁺14] and Salman *et al.* [SSD13] refer to blocks as potential *feature-to-code mappings* or *traces*. AL-msie'deen *et al.* call them *object-oriented building elements sets* [ASH⁺13c] and *atomic blocks* [AM13].

Each calculated block cannot be directly considered the implementation of a feature. We distinguish the block and the feature identification activities considering that blocks help in feature identification but they still need to be refined with domain expertise.

 In Chapter 4, we present how blocks are important entities of our generic and extensible framework, and how existing feature identification techniques are complementary to this framework.

3.1.2 Feature naming during feature identification

Domain knowledge, which is characterized by a set of concepts and terminology understood by practitioners in the area of expertise is a sensible subject [NC⁺09]. During the SPL adoption process, the organization must unify a vocabulary that will enable the stakeholders to share a common vision of the SPL. The vocabulary related to the product characteristics is formalized in the FM which is key in engineering SPLs. The feature naming step during feature identification is an overlooked topic in the literature of extractive SPL adoption. Current automated blocks identification processes are not focused on the naming problem neither on supporting final users via visualisation paradigms.

Davril *et al.* presented a feature naming approach as part of their automatic feature model extraction method [DDH⁺13]. As input they used large sets of product descriptions in natural language. Itzik *et al.* presented a similar automatic approach but using product requirements [IRW16]. While these approaches are automatic, in Chapter 8 we propose a visualisation paradigm to include the domain experts early in the naming process, which could be used together with an automatic process if desired. We present relevant excerpts from the literature in extractive SPL adoption which acknowledge that current approaches still implement feature naming as a manual task that might be costly:

Yang *et al.* let “the analysts examine each of the generated candidate features (concept clusters) to evaluate its business meanings [...] After this manual examination and naming, all the domain features are determined with significant names denoting their business functions” [YPZ09].

AL-msie'deen *et al.* “manually associated feature names to atomic blocks, based on the study of the content of each block and on our knowledge on software” [ASH⁺13c] before suggesting feature naming as a research direction in order “to automatically propose feature names for the atomic blocks” [ASH⁺13b].

Méndez-Acuña *et al.* propose the discovery of reusable blocks (modules in their terminology) [MGC⁺16]. As in previous examples, their contribution is not about naming. In that work, they assigned numbers to the different identified blocks. In the same direction, Ballarin *et al.* stated that “Current techniques [...] do not provide meaningful names, only synthetic names (*F1*, *F2*, etc.)” And then, they “decided to add more meaningful names in order to improve understanding” by manually assigning the names of the identified features [BLC16].

If naming should be done manually, a lack of support in state-of-the-art approaches for feature identification will have to be faced. This is an important threat to their efficiency and challenges their end-to-end usage. Ziadi *et al.* further admitted that checking the mapping of blocks to actual features “is a manual step and thus it requires a lot of effort to be accomplished” [ZHP⁺14]. In fact, migration scenarios can vary in number of blocks, features, stakeholders and in the degree of availability of the domain knowledge.

✎ In Chapter 8 we present a visualisation paradigm and a support tool for assistance during feature naming in feature identification. Practitioners having some knowledge about the domain will quickly validate/modify the propositions of our approach, thus speeding up the naming process. Our visualisation paradigm is motivated by the necessity to 1) close the gap for providing support for the manual task of naming during feature identification by leveraging legacy variants and 2) to speed up and improve the quality of feature identification.

3.1.3 Feature location

Compared to the feature identification process, the assumption in feature location is that the features are known upfront. Therefore, feature location focus on mapping features to their concrete implementation elements in the artefact variants. Feature location techniques in software families also use to assume that feature presence or absence in the product variants is known upfront [FLLE14]. Rubin *et al.* [RC13b] and Wesley *et al.* [AV14] conducted surveys about the state-of-the-art in this domain. They showed the variety of techniques and application domains.

Feature location has been investigated in other software engineering fields such as in maintenance (e.g., determining relevant elements for a modification task [Rob05, RM02]). These techniques have been also used in extractive SPL adoption. Alves *et al.*, in a case study of commercial mobile game variants [AMC⁺07], instead of using static comparison techniques, located the implementation elements of the known features through concern graphs [RM02]. Kästner *et al.* proposed a semi-automatic approach for feature location in single systems where, as input, the domain expert manually needs to point the system to relevant fragments of an artefact with respect to a feature [KDO14]. Then, the approach automatically expands this user selection using information about element dependencies.

Depending on the type of the artefacts, feature location can focus on code fragments in the case of source code [RC12b, FLLE14, ZHP⁺14, ASH⁺13a], model fragments in the context of models [FBHC15] or software components in software architectures [ACC⁺14, GRDL09]. Therefore, existing techniques are composed of two phases: An *abstraction* phase, where the different artefact variants are abstracted, and the *location* phase where algorithms analyse or compare the different product variants to obtain the implementation elements associated to each feature. Despite these two phases, the existing works differ in:

- *The way the product variants are abstracted and represented.* Indeed, each approach uses a specific formalism to represent product variants. For example, AST nodes for source code [FLLE14], model elements to represent model variants [RC13b] or plugins in software architectures [ACC⁺14]. In addition, the granularity of the sought implementation elements may vary from coarse to fine [KAK08]. Some use fine granularity using AST nodes that cover all source code statements while others use purposely a little bit bigger granularity using object-oriented building elements [ASH⁺13a] like Salman *et al.* that only consider classes [SSD14].

- *The proposed algorithms.* Each approach proposes its own algorithm to analyse product variants and identify the groups of elements that are related to features. For instance, Fischer *et al.* used a static analysis algorithm [FLLE14]. Other approaches use techniques from the field of Information Retrieval (IR). Xue *et al.* [XXJ12] and Salman *et al.* [SSD13] proposed the use of Formal Concept Analysis (FCA) to group implementation elements in blocks and then, in a second step, the IR technique Latent Semantic Indexing (LSI) [DDL⁺90] to map between these blocks and the features. Salman *et al.* used hierarchical clustering to perform this second step [SSD14].

✎ In Chapter 4, we propose a unified representation of product variants in order to use existing feature location techniques for different artefact types. In Chapter 6, we propose a benchmarking framework for intensive experimentation of feature location techniques.

3.1.4 Constraints discovery

After identifying features from a set of variants, we need to infer feature constraints to build a FM that accurately defines the validity boundaries of the configuration space. The literature proposes approaches for mining feature constraints from existing artefacts, although they focus on specific artefact types, such as C source code [NBKC14] or Java [RPK11, AmHS⁺14, SSS16]. Some approaches do not rely on the internal elements of the artefact variants. For example, they rely on the existing feature configurations from the initial artefact variants to reason on the constraints [LMSM10, LHGB⁺12, HLE13, CSW08, HLE13]. Other approaches use existing documentation [DDH⁺13] or machine learning techniques along with an oracle that dictates when a given configuration is valid [TDAJ16].

✎ In Chapter 4 we propose an approach for constraints discovery that can leverage these specific techniques. Thus, they can be generalized to other artefact types. In addition, they can be simultaneously used to aggregate their results. In Chapter 9 we present a visualisation to assist domain experts in reasoning about feature relations to discover feature constraints.

3.1.5 Feature model synthesis

Once the features and the constraints are identified, the objective of feature model synthesis is to create comprehensive feature diagrams for the domain experts. Figure 3.1 presents an illustrative example of two different feature diagrams representing the same configuration space. A feature diagram contains a hierarchy of features, enriched by cross-tree feature

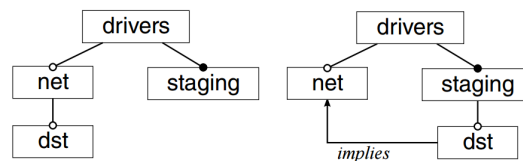


Figure 3.1: Two feature diagrams with the same configuration semantics [SRA⁺14].


constraints. Its structure should be heuristically defined and feature model synthesis has been proven to be an NP-hard problem [SRA⁺14].

Some approaches rely on the constraints defined in propositional formulas [SRA⁺14, HLE13]. Bécan *et al.*, trying to approximate more to the domain experts' expectations, embed ontological information of the domain [BABBN15]. WebFML is a framework specialized in feature model synthesis [BBNAB14]. Itzik *et al.* combine semantic and ontological considerations via mining requirement documents [IRW16].

 In Chapter 4 we consider this activity in extractive SPL adoption.

3.1.6 Reusable assets construction

Once the features are identified from the artefact variants, the final step towards SPL adoption is to construct the reusable assets that are associated to the features. This construction should enable the creation of new artefacts by composing or manipulating the associated reusable assets of the features. As mentioned before, approaches for identifying and extracting features from single software systems has been proposed [KDO14]. In the case of artefact variants, other approaches constructed reusable assets based on source code abstract syntax trees [FLLE14, ZHP⁺14]. Méndez-Acuña *et al.* constructed reusable language modules [MGC⁺16]. Finally, other approaches focused on defining a framework of n-way merging of models to create SPL representations [RC13a].

 The extensible framework and the notion of adapters explained in Chapter 4 offer the possibility of leveraging and integrating different approaches.

3.2 The model variants scenario

Several approaches have been proposed to study the mining of model variants for extracting Model-based SPLs (MSPL). We consider an MSPL as an SPL whose final products are models [FFBA⁺14]. In this model-driven engineering (MDE) scenario, CVL is a modeling approach specialized in implementing MSPLs [HWC13]. Figure 3.2 shows the CVL process for enabling a language-independent solution. CVL provides the necessary expressiveness to support systematic reuse for the derivation of models [CGR⁺12]. Concretely, the Variability model of CVL consists of two layers:

- **Variability definition layer:** This layer defines the variability of the domain in a very similar fashion as feature modeling does [KCH⁺90], i.e., this layer represents the features of a product family and the constraints among them.
- **Product realization layer:** This layer defines the modifications to be performed in a *Base Model* according to the features of the variability specification (e.g., adding or removing model fragments if a specified feature is selected).

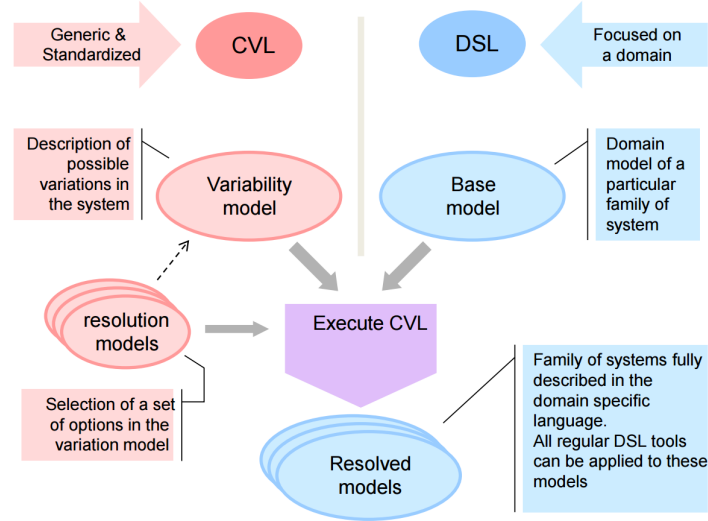


Figure 3.2: Common Variability Language [HWC13].

CVL was thus designed to simplify a top-down approach to the adoption of MSPL, where practitioners directly define and implement the assets for defining the MSPL. Tools for variability management (e.g., implementations of CVL [SIN15, HØ14]) are not focused on extractive SPL adoption.

For extractive MSPL adoption, Zhang *et al.* propose the CVL Compare process [ZHM11]. Concretely, they use EMF Compare [Ecl16b] to analyse a set of model variants and CVL to create a preliminary MSPL model. This approach is similar to the one that we present in Chapter 5, however, there are three differences: 1) while we automatically calculate the Base Model, CVL Compare relies on the SPL developer to choose it, 2) CVL Compare is based on EMF Compare two-way comparison mechanism, i.e., model variants are only compared with each other but not simultaneously, 3) CVL Compare does not consider the identification of constraints. The structural constraints are important to construct valid MSPLs.


Model comparison techniques have received a lot attention in the MDE community. However, this is often limited to a comparison between two versions of a model. Koschke *et al.* propose an automated approach for comparing specific model variants that only concern the static architectural view [KFBA09]. Stephan *et al.* [SC13] and Kolovos *et al.* [KDRPP09] analysed model comparison approaches and provided a classification for them. Model splitting can be used to reduce the complexity of maximal models for human comprehension and team collaboration by hiding part of it [SRTC14]. These approaches provide heuristics to modularize the model before splitting. However, they focus on single models, not model variants. Heuristics for model splitting can be complementary in the feature identification and location processes.

Rubin *et al.* propose the *merge-refactoring* framework [RC12a, Rub14]. *Merge-refactoring* is a formal framework to compare UML model variants using what is referred as n-way model merging [RC13a]. Their merge-in operator creates a maximal model merging the input variants. Their work is mostly focused in the challenge to merge and they do not tackle the problem of trying to analyse the identified blocks nor that of finding comprehensive feature

names. In the realization of their approach, the merge-in produces a feature for each input variant and uses it to annotate all elements which originated from that variant. Therefore, they extract a maximal model with annotations that can be used on MSPL based on an annotative derivation mechanism.

Regarding feature identification, Ryssel *et al.* compare model variants that are represented as function-blocks [RPK10]. Ziadi *et al.* propose an approach to analyse the source code through the use of UML class diagrams of a set of software variants and identify commonality and variability among them [ZFdSZ12]. Apart from not covering the reusable assets construction activity, these approaches are not generic to any MOF-based scenario. The tool FeatureMapper enables manual and automatic mapping of model elements to features [HKW08]. However, their automatic mapping is based on monitoring the developers when they are implementing a feature.

Regarding feature constraints discovery, Bosco *et al.* targeted the challenge of generating invalid models from MSPL to be used as counter-examples to refine variability models [FFBA⁺14]. Blanc *et al.* propose an approach for validating sub-models of a maximal model [BMMM08]. Czarnecki *et al.* propose feature-based model templates verification against OCL constraints [CP06].

 In Chapter 5 we focus on providing an extractive approach for MSPL adoption.


3.3 Visualisation

As reported by Nestor *et al.*, “*the use of visualisation techniques in SPL can radically help stakeholders in supporting essential work tasks and in enhancing their understanding of large and complex SPLs*” [NTB⁺08]. One popular visualisation in SPLE is the use of colors to associate features to implementation elements. Kästner *et al.* propose CIDE (Colored Integrated Development Environment) to visualise the mapping of features to source code by extending source code editors with color highlighting [KTA08]. Concretely, each feature is associated with a color. In a similar way, Heidenreich *et al.* propose in FeatureMapper to use color highlighting inside model diagrams to visualise the traceability between features and model elements [HKW08].

Apart from coloring, related work in SPL visualisation mainly consider assistance to domain experts or customers during product configuration (i.e., selecting the features that fit the customer needs). Pleuss *et al.* conducted a survey on the use of visualisation techniques for this purpose [PRB11]. Specific examples are VISIT-FC (Visual and Interactive Tool for Feature Configuration) from Nestor *et al.* [NTB⁺08]. Botterweck *et al.* propose another visual configurator based on tree visualisations [BJS09]. Other approaches, such as the one proposed by Schlee and Vanderdonckt, are based on transformation patterns from feature modeling concepts to traditional user interfaces (e.g., windows with checkboxes, buttons and dialogs) [SV04].

Other visualisations provide assistance for other specific tasks. For instance, Lopez-Herrejon *et al.* propose the use of visualisation techniques to display features for pairwise testing [LHE13]. However, few visualisation paradigms have been proposed that help analysing the artefact variants during the extractive SPL adoption process. In this dissertation we tackle the assistance during feature naming and constraints discovery activities. Regarding feature naming, we already discussed in Section 3.1.2 the lack of visualisation for this activity, so we present related work regarding feature constraints visualisation.

Understanding constraints among features is challenging for human comprehension given that real-world SPLs yield FMs that can be large and complex. VISIT-FC [NTB⁺08] or the tool by Pleuss *et al.* use arcs between features [PB12]. Trinidad *et al.* proposed the cone tree visualisation paradigm introducing a 3D representation of FMs [TCBS08]. However, this visualisation represent the hierarchical information of features without considering constraint visualisation. Understanding feature relations is specially relevant in extractive SPL adoption where feature constraints needs to be identified. Techniques to extract constraints from existing configurations, as the ones presented in Section 3.1.4, focused on automation and they do not propose visualisation support for the involvement of domain experts.

 In Chapter 8 we present a visualisation for feature naming during feature identification. In Chapter 9 we present a visualisation paradigm for supporting domain experts in manual analysis of feature relations that enable free exploration for constraints discovery.


3.4 Benchmarks and case studies

In SPL engineering several benchmarks and common test subjects have been proposed. Lopez-Herrejon *et al.* proposed evaluating SPL technologies on a common artefact, a Graph Product Line [LB01], whose variability features are familiar to any computer engineer. The same authors proposed a benchmark for combinatorial interaction testing techniques for SPLs [LFC⁺14]. Also, automated FM analysis has a long history in SPLE research [BSC10]. FAMA is a tool for feature model analysis that allows to include new reasoners and new reasoning operators [TBC⁺08]. Taking as input these reasoners, the BeTTY framework [SGB⁺12], built on top of FAMA, is able to benchmark the reasoners to highlight the advantages and shortcomings of different analysis approaches.

Feature location on software families is also becoming more mature with a relevant proliferation of techniques. Therefore, benchmarking frameworks to support the evolution of this field are in need. Different case studies have been used for evaluating feature location in software families [AV14]. For instance, ArgoUML variants have been extensively used [CVF11a]. However, none of the presented case studies have been proposed as a benchmark except the variants of the Linux kernel by Xing *et al.* [XXJ13]. This benchmark considers twelve variants of the Linux kernel from which a ground truth is extracted with the traceability of more than two thousands features to code parts. This benchmark does not provide a framework to support experimentation by easily integrating feature location techniques.

Also, case studies for experimenting with feature identification techniques are needed. Github is trending in the mining software repositories community. However, its expansion as public repository has degraded or complicated its practical exploitation in research [KGB⁺14], mainly because the results may not be generalizable to software engineering practices in industry. In Chapter 7 we focus on Android application (app) markets to try to identify families of apps. Without detracting from the validity of research using Github, we consider that Android apps at the app markets have more guarantees of not being merely personal projects.

Reuse practices in Android markets have been studied [LBKLT16, RAN⁺14] and the research field on malware detection is creating advanced and scalable methods for mining app markets [LBP⁺16]. Their objective is to heuristically define if a legitimate app has a malware counterpart. The Diversify project has conducted several large-scale studies regarding software libraries usages in different contexts such as JavaScript websites, WordPress or Mavenⁱ. Their objective is to analyse the diversity as an enabler to adapt a software product during its evolution.

 In Chapter 6 we present a benchmark for feature location based on variants of the Eclipse integrated development environment (EFLBench). The Linux kernel benchmark can be considered as complementary to advance feature location research because EFLBench a) maps to a project that is plugin-based, while Linux considers C code, and b) the characteristics of the natural language terminology is different from the Linux kernel terminology. This last point is important because techniques based on information retrieval techniques should be evaluated in different case studies. EFLBench is integrated with BUT4Reuse which is extensible for feature location techniques making easier to control and reproduce the settings of the studied techniques.

3.5 Generic and extensible frameworks in SPLE

SPLE is a maturing field that has witnessed a number of contributions, in particular in the form of frameworks for dealing with different aspects of variability management. Unfortunately, general approaches for mining existing artefacts are still not mature enough, delaying real-world SPL adoption. A contribution of this thesis is a unified, generic and extensible framework for extractive SPL adoption that harmonizes the adoption process (presented in Part II) but we can highlight other similar efforts for harmonizing other activities in SPLE.

Tools such as Pure::Variants [Beu10] and Gears [KC] provide variability management functionalities targeting not only source code but also other artefacts including DOORS requirements, Microsoft Rational or Microsoft Excel spreadsheets to name a few. Handling new artefact types while sharing most of their core functionality for variability management is possible through *extensions* in Pure::Variants and *bridges* in Gears. The Feature IDE tool [TKB⁺14] also provides extensibility for including *composers* dealing with different artefact types. Among the default composers, this tool includes FeatureHouse [AKL09] which by itself is already a generic and extensible composition framework providing support for several programming

ⁱ<http://diversify-project.eu/data/>

languages. As another example, we already presented CVL in Section 3.2. CVL defines the variability on models as well as the composition of model elements in a meta-model-independent way [HWC13]. That means that any DSL can be enriched with variability management and variants derivation functionalities.

All the mentioned tools do not tackle the extractive SPL adoption by themselves. The feature model and the reusable assets are designed and developed inside the tool from scratch without support for mining existing artefact variants. On the contrary, our framework is focused on helping domain experts in the extractive SPL adoption process. Of course, the mentioned tools can be used to leverage and manage the mined variability and the extracted reusable assets to create an operational SPL [JDB07, RCC13].

ECCO (Extraction and Composition for Clone-and-Own) [FLLE14] or VariantSync [PTS⁺16] are proposed as variability-aware configuration management and version control systems. As part of their functionalities, they perform feature location on artefact variants and they were evaluated in source code case studies. The objectives of all these frameworks focus on specific activities of extractive SPL adoption.

Clone detection has been also used as an approach to identify features in a set of artefact variants. Most clone detection techniques are focused on the peculiarities of a targeted programming language [RC07] or to a specific meta-model in the case of models (e.g., simulink models [SASC12, DHJ⁺08, Pet12] or graph-based models [PNN⁺09]). The clone analysis workflows of ConQAT [JDH09] or JCCD [BD10] support several languages and provide extensibility for adding new types of artefacts while reusing visualisations and other analysis workflow elements. Using ConQAT, we introduced the cross-product clone detection approach to deal with source code artefact variants [MT12]. However, the tool was still not adapted to extractive SPL adoption since this was part of feature identification which is only an activity of the process. MoDisco [BCDM14] is another example of an extensible framework to develop model-driven tools to support software modernization. However, these tools do not specifically target sets of artefact variants and focus only on single systems.

3.6 SPL configuration spaces and end users

In this section we present related work to Part V that assumes the existence of an SPL with Human-Computer Interaction (HCI) components.

3.6.1 Interactive analysis of configuration spaces


Instead of developing a software platform representing a solution for a wide range of user profiles, it is interesting to perform feature-based analysis to identify the most adapted solutions for specific needs [SMA13]. To achieve this objective, we should consider end user assessments as a driver for SPLE processes. Ali *et al.* presented the vision of *social* SPLs where the main goal is to leverage user feedback over time to derive products that maximise

the satisfaction of a given user profile or collective [ASD⁺11]. In this context, it seems mandatory to evaluate how users respond to the proposed artefact variants (e.g., usability evaluation). Therefore, there is a need to analyse the SPL and its configuration space.

Selecting optimal SPL product variants based on some criteria has been studied in SPLE. To achieve this, it is a common practice to enhance the variability model by what is referred to as quality attributes [BMAC05, SMA13, SRK⁺13, CDFP97, dCMMdA12, HPP⁺13a, JHF⁺12]. Quality attribute annotations in these approaches are fixed independently of user feedback (e.g., *cost* quality attribute annotations defined as a fixed value per-feature [HPP⁺13a, SMA13]). In this context, Benavides *et al.* consider the selection problem as a Constraint Satisfaction Problem (CSP) and solve it using CSP solvers [BMAC05]. Sayyad *et al.* use search-based algorithms [SMA13]. Also, in the SPL testing community, several approaches have been proposed to select the best configurations to be tested guided by a set of predefined testing objectives [CDFP97, dCMMdA12, HPP⁺13a, JHF⁺12]. These testing objectives can be considered as quality attributes.

Unfortunately, if we consider end users, they provide subjective assessments which require feedback aggregation mechanisms (e.g., combining many assessments to draw some form of conclusion). Also, the assessments are performed for a product as a whole, indeed, users are not necessarily aware of the variability mechanisms nor of the specific features behind the assessed variants.

Genetic algorithms have been used for guiding the analysis of the configuration space [EBG12, SMA13]. A key operator for evolutionary genetic algorithms is the *fitness function* representing the requirements to adapt to. In other words, it forms the basis for selection and it defines what improvement means [ES03]. In our case, the fitness function is based on user feedback which is a manual process in opposition to automatically calculated fitness functions (e.g., the sum of the cost of the features). Because we deal with these user assessments as part of the search in the configuration space, we leverage Interactive Genetic Algorithms (IGA) where humans are responsible to interactively set the fitness [ES03, Tak01]. This technique has been already used, specially in cases dealing with high subjectivity as we discuss in next subsection.

 In Chapter 10 we present an approach to analyse the configuration space using an interactive evolutionary approach followed by a data mining technique.


3.6.2 Dealing with high subjectivity

Most probably, the highest subjectivity levels are obtained when dealing with human perceptions. For example, deciding if an artwork is interesting or not uses to be a debatable subject. Concretely, the challenges for leveraging user feedback to improve the results of the derivation of products have been already tackled in the computer-generated art community. Concretely, evolutionary computing applied to computer-generated art is the main technique used to leverage user feedback. In this way, IGAs have been used for aesthetics evaluation [RM08, Dor13].

Eiben presented two works on computer generated art that takes into account user feedback: the Mondriaan and Escher evolvers [Eib08]. His objective was to mimic the artist styles by asking people how close they consider that a generated painting conforms to the painter style. The configuration space was explored through an IGA. In our case, the objective is not necessarily to approach to “style fidelity”. The objective is to maximize the changes of collective acceptance of the derived products, therefore, the result can be far from the expectations of the artist.

The most relevant difference of our work compared with evolutionary art approaches is that they only rely on the evolution phase. Their assumption is that the most adapted art products found at the end of the evolution should be the best ones. However, in a standard situation, not all the possible configurations were assessed by the users given time and resource limitations. We present a second phase in order to predict the expected user feedback of the non-assessed products and the creation of a ranking.

Regarding previous work on SPLs with artistic content that might deal with subjectivity, we find that they did not made use of user assessments to reach their objectives. For example, Alves *et al.* presented an approach for managing variability in images of mobile games using SPL techniques [ASC⁺06]. They implemented image decomposition in several features (e.g., they decompose a character in arms, head, torso and legs and they introduced automatic transformations and location of images as features). The benefits targeted by them were to improve the general performance of a game by reducing the necessary images to load each frame. Acher *et al.* presented a tool to manage the variability on the characteristics of video sequences to test video processing algorithms [AAG⁺14]. Acher *et al.* also presented the synergies between SPLs and the variability management techniques used in 3D printing communities [ABBJ14].

 In Chapter 11 we present a case study dealing with an SPL-based computer generated art system.

Part II

MINING ARTEFACT VARIANTS FOR
EXTRACTIVE SPL ADOPTION

GENERIC AND EXTENSIBLE EXTRACTIVE ADOPTION OF SOFTWARE PRODUCT LINES

This chapter is based on the work that has been published in the following paper:

- Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Bottom-up adoption of software product lines: a generic and extensible approach. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 101–110. ACM, 2015

Contents

4.1	Introduction	38
4.2	A Framework for extractive SPL adoption	39
4.2.1	Principles	39
4.2.2	Designing an adapter to benefit from the framework	41
4.2.3	Activities for extractive SPL adoption within the framework	43
4.3	Framework realization in BUT4Reuse	45
4.3.1	Genericity in BUT4Reuse through the adapters	46
4.3.2	Block identification	48
4.3.3	Feature identification	49
4.3.4	Feature location	50
4.3.5	Constraints discovery	53
4.3.6	Feature model synthesis	54
4.3.7	Reusable assets construction	55
4.3.8	Visualisations	55
4.4	Experiences and evaluation with the Eclipse case study	56
4.4.1	Design of the Eclipse adapter	57
4.4.2	Results and discussions	58
4.5	Limitations	61
4.6	Conclusions	62

4.1 Introduction

There are still research challenges that go beyond the specific techniques for the extractive SPL adoption activities presented in Section 2.2.2. Without detracting from the importance of improving and proposing techniques for feature identification, location, constraints discovery, feature model synthesis or reusable assets construction, this chapter faces the challenge of designing a common framework to unify the extractive SPL adoption process.

Causes for the lack of unification of extractive SPL adoption techniques.

- *The proposed approaches are often related to specific kinds of software artefacts.* As we presented in Section 3.1, there is a large body of techniques for achieving the objectives of SPL adoption. Unfortunately, their design and implementation are often specific to a given artefact type. There is, however, an opportunity to reuse the principles guiding these techniques for other artefacts.
- *The absence of a unified process:* From feature identification and location to feature model synthesis and reusable assets construction, addressing within the same environment the different objectives of these activities is challenging. Each approach requires inputs and provides outputs at different granularity levels and with different formats complicating their integration in a unified process. If such techniques are underlying in a framework, they could be seamlessly used in different scenarios.
- *The need for benchmarking:* The existing approaches and techniques for extractive SPL adoption differ in the way they analyse and manipulate the artefact variants. Thus, they also differ in their performance and quality of the results. In this context, their assessment and comparison become challenging.

Contributions of this chapter.

- **A unifying framework to support extractive SPL adoption.** The process proposed in the framework is built upon an analysis of the steps from the state-of-the-art on mining artefact variants. We use an intermediate model allowing to generalize and integrate existing specific techniques.
- **A framework that allows domain experts to experiment with existing techniques** for each of the activities of the process. This framework provides a common ground for assessing and comparing different algorithms as well as comparing different ways to chain them during the activities of extractive SPL adoption.
- **A framework with extensibility for visualisations** to support domain experts in manual tasks during extractive SPL adoption.
- **BUT4Reuse (Bottom-Up Technologies for Reuse):** A realization of the framework for extractive SPL adoption specially designed for genericity and extensibility.

This chapter is structured as follows: Section 4.2 presents the framework principles and the promoted process. Section 4.3 presents the realization of the framework. Section 4.4 presents an empirical evaluation and Section 4.5 discusses the limitation of the framework. Finally, Section 4.6 presents the conclusions outlining future research directions.

4.2 A Framework for extractive SPL adoption

This section presents the framework principles and the notion of adapters to support different artefact types. Then we present the design of an adapters using an illustrative example. Finally, we detail the activities of the framework for extractive SPL adoption.

4.2.1 Principles

In order to be *generic* in the support of various artefact types, our framework is built upon the following three principles:

- (1) A typical software artefact can be decomposed into distinct elements, hereafter referred to as *elements*.
- (2) Given a pair of elements in a specific artefact type, a similarity metric can be computed for comparison purposes.
- (3) Given a set of elements recovered from existing artefacts, a new artefact, or at least a part of it (which would be a reusable asset), can be constructed.

On the one hand, principles (1) and (2) make possible to reason on a set of software artefacts for identifying commonality and variability, which in turn will be exploited in feature identification and location activities. Principle (3), on the other hand, promises to enable the construction of the reusable assets based on the elements found in existing artefacts.

Because our approach aims at supporting different types of artefacts, and to allow extensibility, we propose to rely on *adapters* for the different artefact types. These adapters are implemented as the main components of the framework. The adapter is responsible for decomposing each artefact type into the constituting elements, and for defining how a set of elements should be constructed to create a reusable asset. Figure 4.1 illustrates examples of adapters dealing with different artefact types. We can see how an adapter allows two operations, 1) to adapt the artefact to elements and 2) to take elements as input to construct a reusable asset for this artefact type.

Source code can be adapted to Abstract Syntax Tree (AST) elements which capture the modular structure of source code. Figure 4.2 presents two ASTs of two different variants presented in Section 2.1.2 regarding the example of an EShop user interface. The root of the tree (*eshop*) is the Java package, the second level are classes, and the third are methods and fields. Concretely, Figure 4.2a shows the AST elements of an EShopUI without the **Search** feature, and Figure 4.2b shows the AST elements obtained from decomposing another variant

with **Search**. Apart from source code, a Text file can be decomposed in Line elements, or EMF Models [Ecl16a] can be decomposed in Meta-Object Facility Elements [OMG06] such as Class, Attribute and References. More details about the available adapters are presented in Section 4.3.1.

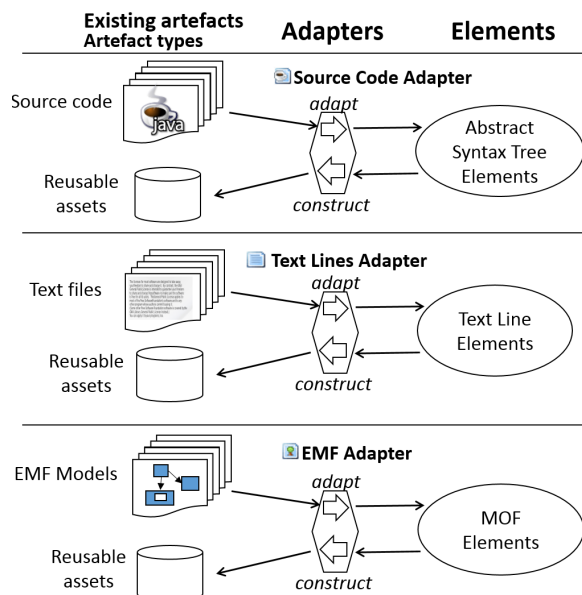


Figure 4.1: Artefact type examples and elements representation creation through the adapters. The adapter can also construct reusable assets taking a set of elements as input.

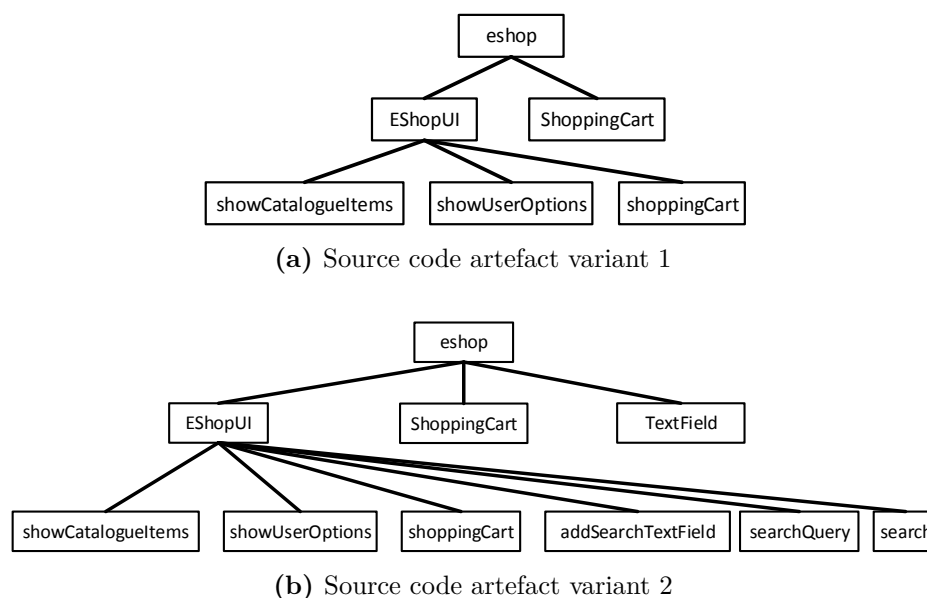


Figure 4.2: Two Abstract Syntax Trees (AST) to illustrate how a source code adapter can decompose the structure of different source code artefacts. In this example, variant 2 has four AST elements that are not present in variant 1.

4.2.2 Designing an adapter to benefit from the framework

Designing an adapter for a given artefact type requires four tasks. Throughout this dissertation, we refer to these tasks to explain the design decisions in the implementation of various adapters.

Task 1: Element identification. *Identifying the Elements that compose an Artefact.* This will define the granularity of the elements in a given artefact type. For the same artefact type we can select from coarse to fine granularity. For example, we can design an adapter for source code that only takes into account whole components (e.g., packages), or an adapter that decomposes in the AST elements presented in Figure 4.2, or we can decide to use a finer granularity and decompose statement elements inside the methods.

Task 2: Structural constraints definition. *Identifying Structural Dependencies for the Elements.* When the artefact type is structured, the elements will have containment relations. For example, in ASTs we have the relation to the parent. In model artefacts there is usually a root and containment references as well. However, there can be other types of structural dependencies. In the case of source code, this information is usually captured in program dependence graphs. For example, a class can *extend* another class producing a structural dependency, or a class, inside one of its methods, can *instantiate* another class. In addition to identifying all the dependencies among elements, in this task, it is also important to identify the cardinality of these element relations. For example, following with source code, if the programming language does not allow multi-inheritance, the inheritance dependency between two classes should have a maximum of one.

Task 3: Similarity metric definition. *Defining a Similarity metric between any pair of Elements.* An element should compare its similarity with another element getting a value ranging from zero (completely different) to one (identical). The similarity function is defined as $\sigma : E \times E \rightarrow [0, 1]$. There is no general rule to calculate similarity. In some cases the elements provide an id, a unique signature or, in the case of a tree structure, a unique path to the root element. In other cases, it can be more complicated and heuristics must be integrated considering different structural characteristics, semantic relationships of retrieved words from the elements, or other kind of semantic similarity.

Task 4: Reusable assets construction. *Defining how to use a set of elements to construct reusable assets.* In order to decide how to construct, it is important to decide how the reusable asset will be composed with other reusable assets. If we are using a compositional approach, we will need to construct a fragment that conforms to this compositional approach. In general, if we decompose an artefact in a set of elements and we use the same set of elements for the construction, we should be able to obtain the same artefact.

To illustrate the design of an adapter, we will consider the following scenario:

Extractive SPL adoption for images: An illustrative example

A graphic artist wants to adopt an SPL from image variants that were initially created following the *copy-paste-modify* reuse paradigm. On the left side of Figure 4.3 we present the set of existing image variantsⁱ and on the right side we illustrate the design decisions for the images adapter.

- *Task 1: Element identification.* It was decided to adapt image variants in an intermediate representation based on Pixel elements. The decomposition of an image into Pixel elements consists in loading the whole pixel matrix from the image file and obtaining the non-transparent pixels as Pixel elements.
- *Task 2: Structural constraints.* Each Pixel element has a structural dependency with its position (Cartesian coordinates). It was decided that pixel overlapping is not allowed so only one pixel is permitted for each position. This will allow to automatically discover structural constraints between features, e.g., two shirts cannot be used in the same image because at least one pixel position will be shared and the pixels in that position cannot overlap.
- *Task 3: Similarity metric.* The similarity metric consists in checking that the Pixel element positions are exactly the same, and then calculating the similarity of the color (red, green and blue values) and alpha channel (transparency).
- *Task 4: Reusable assets construction.* The construction for a set of Pixel elements is based on including the pixels at their corresponding positions in a transparent image. The targeted compositional approach is the image composer of the FeatureIDE variability management tool [TKB⁺14]. This composer will be able to use the constructed assets as reusable assets in a derivation mechanism based on superimposition of images.

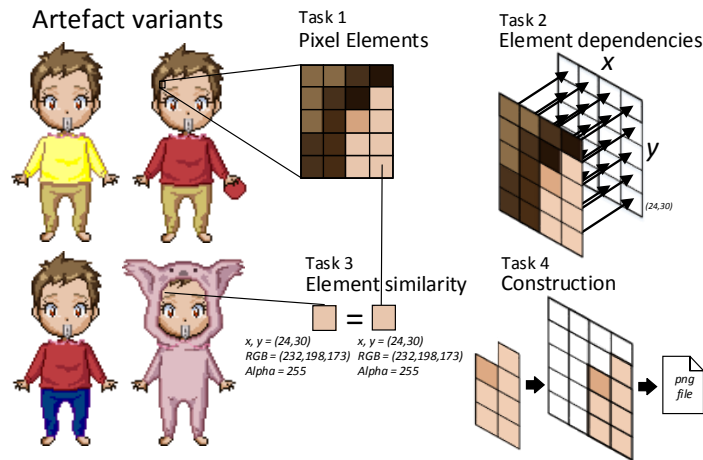


Figure 4.3: Design decisions for the images adapter.

ⁱImages obtained from:

<http://deco-kun.deviantart.com/art/Hikari-Yagami-all-outfits-in-Pixel-304142452>

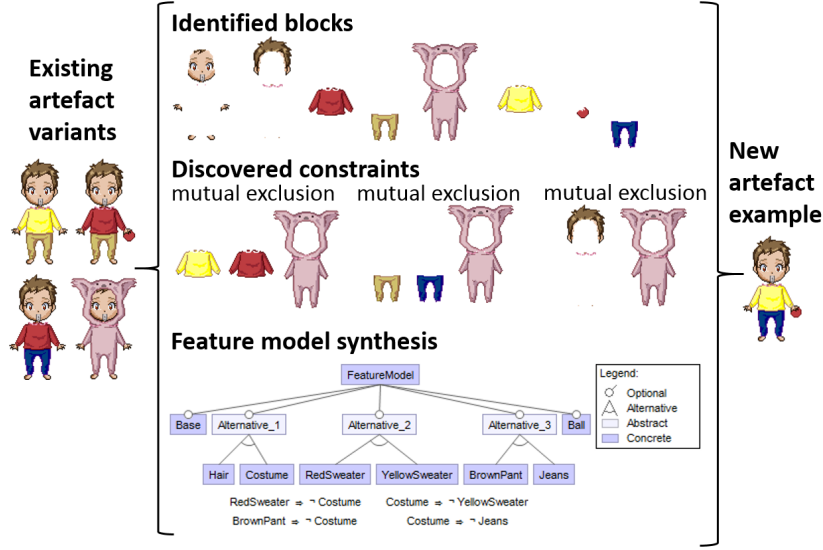


Figure 4.4: Example of image variants and the result of applying the images adapter.

Figure 4.4 illustrates how the images adapter enables mining artefact variants of images. The adapter is the “piece” that need to be implemented to chain extractive SPL adoption activities towards the creation of an images SPL. The extracted SPL can be used to create new variants, as the one shown on the right side of the figure.

4.2.3 Activities for extractive SPL adoption within the framework

In Section 2.2.2, we presented relevant activities in extractive SPL adoption. Figure 4.5 presents the concepts and complementary activities introduced by our framework. Regarding the role of the adapter, in the *decomposition* layer, on the left side of Figure 4.5, we illustrate how the adapter represents the strategy for the decomposition of the artefact variants into elements. Also, on the right side of the figure, we illustrate how the adapter and its construction method helps in preparing the reusable assets.

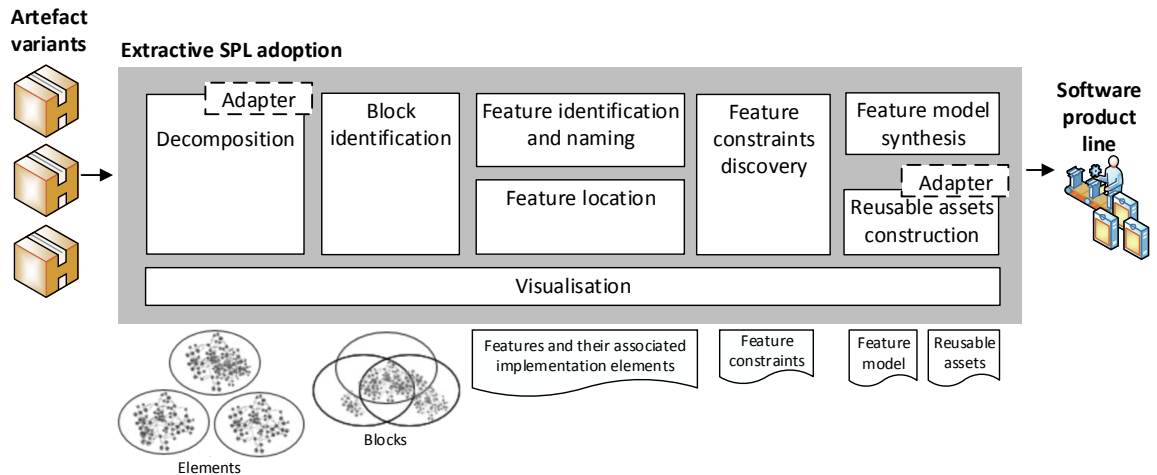


Figure 4.5: Relevant activities during extractive SPL adoption within our framework.

The framework builds on an architecture composed of various extensible layers explained below. After the decomposition of all artefact variants in elements, the elements are grouped via block identification. Then, with the abstraction of elements and blocks, we can chain the activities for extractive SPL adoption. Moreover, in Figure 4.5 we present the visualisation layer for domain experts which is an horizontal layer that covers the whole process.

Decomposition: The first layer is the decomposition layer using the *adapters*. Therefore, this layer is extensible by providing support for different artefact types. It enables the creation of the *elements representation* which provides a common internal representation.

Block identification: As presented in Section 3.1.1, a block is a set of elements obtained by analysing the artefact variants (e.g., comparing the variants to identify common and variable parts). Blocks permit to increase the granularity of the analysis by the domain experts so that we do not have to reason at element level. This is specially relevant when trying to identify or locate features, and also during constraints discovery. In the context of feature identification, block identification represents an initial step before reasoning at feature level. In our running example, the identified blocks shown in Figure 4.4 correspond to the sets of Pixel elements of the different “parts” of the images. The employed technique, to be explained later, is the *interdependent elements* technique [ZFdSZ12] that calculated the intersections among the Pixel elements of the variants.

Feature identification: Feature identification consists in analysing the blocks to map them to features. For instance, blocks identified as feature-specific can be converted into a feature of the envisioned FM. In other cases, a block can be merged with another block, or we can split the elements of one block in two blocks, to adjust it to identified features. To illustrate the usefulness of this, we present a recurrent issue when dealing with copy-paste-modified source code artefacts. This issue is the situation where a modification (e.g., a bug fix) was released in a set of the artefacts but not in all of them. In this case, the block identification technique could identify three different blocks: One containing the modified elements for the bug fix, one containing the “buggy” elements, and, finally, the one that contains the shared elements that were not part of the modified elements. In this case, we would like to remove the “buggy” block and merge the bug fix block with the shared block.

Also, some features are intended to *interact* with other features. A feature interaction is defined as *some way in which a feature or features modify or influence another feature in defining overall system behavior* [Zav03]. **Logging** feature is a common example to illustrate features that must interact with others because of its crosscutting behavior [CVF11a] (i.e., other features might need to log if the **Logging** feature is present). In some cases, the implementation of the interaction needs *coordination elements* (also known as *derivatives* [LBL06], *junctions* [ZFdSZ12] or *glue code* among others). It is also important to consider the negations of the features, i.e., some elements can be needed when a feature is not present. In our context of extractive SPL adoption, feature interactions, and the coordination elements, must be taken into account.

As part of feature identification, block names should be modified by domain experts to have representative names. In our running example, the graphic artist decided that the identified blocks would be directly assigned to features and a meaningful name to each feature was given.

Feature location: For feature location, we aim to locate the features reasoning on the list of known features and the identified blocks. Concretely, feature location techniques in our framework create mappings between features and blocks with a defined confidence. The confidence is a value from zero to one. For example, if the feature location technique concludes that a given feature is located in a given block with a confidence of one, it means that the technique is almost certain that the elements of this block are implementing this feature.

Constraints Discovery: The identification of constraints by mining existing assets has been identified as an important challenge for research on the SPL domain [AV14]. For this purpose, our framework is extensible to contribute constraints discovery approaches. In our running example, the Pixel elements cannot overlap as discussed before, so mutual exclusion constraints can be found, as shown in Figure 4.4, through the *Structural constraints* technique explained below.

Feature model synthesis: Feature model creation demands a feature model synthesis technique to obtain comprehensible feature diagrams. In our running example, the identified features and constraints can be used to automatically create the feature diagram of Figure 4.4 by including the alternative features notation. As in the previous layers, different techniques can be integrated in this layer.

Reusable assets construction: The framework supports the step of creating the reusable assets from a set of elements using the adapter. Depending on the needs, the set of elements could correspond to a block, an identified feature, or the elements that corresponded to a located feature. In the running example, the reusable assets were constructed and they corresponded to the images of the identified blocks as shown in Figure 4.4.

Visualisation: The visualisation layer is orthogonal to the other layers. It is intended to present to the domain expert relevant information yielded by other layers. Visualisations can be used, not only to display information, but also to interact with the results.

4.3 Framework realization in BUT4Reuse

Bottom-Up Technologies for Reuse (BUT4Reuse) is our realization of the presented framework. Significant efforts have been dedicated to engineering a complete tool-supported approachⁱⁱ. The assessment of the realization of the framework consists of two parts: In the first place, we assess its genericity by presenting the available adapters. Secondly, we assess its extensibility by presenting the different techniques integrated in each of the framework layers. Most of the techniques that we integrated were previously used in extractive SPL adoption. However, given

ⁱⁱBUT4Reuse source code, tests, documentation, user manual and tutorials: <http://but4reuse.github.io>

that they were proposed for specific artefact types, our contribution consists in generalizing them to support the *elements* abstraction.

4.3.1 Genericity in BUT4Reuse through the adapters

Currently, BUT4Reuse integrates 15 adapters dealing with different artefact types to be directly used. Table 4.1 presents detailed characteristics of each of them. Also, it is worthy to mention that one can build on top of these adapters to develop tailored or improved ones. Despite of the different types of artefacts, similarity metrics and construction mechanisms, they can all benefit from the unified framework to perform extractive SPL adoptions.
















From a technical perspective, based on our experience, the development burden for a typical adapter is small in terms of LOC. This is because 1) BUT4Reuse Core implementation provides the dedicated extension points to ease the work of the adapter developer and 2) the adapters can rely on off-the-shelf libraries for the manipulation of the targeted artefact types including its decomposition, similarity calculation or construction. For example, the Text Lines, File Structure, CSV, Graphs, and Images adapters are respectively made of only 177, 207, 210, 274 and 230 LOC. Besides, each of them has been implemented in less than one day by an experienced developer.

The C and Java Source Code adapters have been realized by integrating ExtractorPL [ZHP⁺14]. The integration of this adapter took about one work-day and consists of 930 LOC. The short time to integrate ExtratorPL can be justified because it is also based on the principle of decomposing the artefacts into elements (AST elements in this case). After this integration, we reproduced the same case studies dealing with source code artefact variants presented in previous works [ZFdSZ12, ZHP⁺14]. We further extended ExtractorPL in the case of Java source code by including more information about structural dependencies among elements. ExtractorPL uses ASTs as the ones presented in Figure 4.2 which only capture containment dependencies, thus, omitting other structural dependencies present in source code (for example Java methods instantiating classes from other packages will have a structural dependency with these classes). For this purpose, we use source code dependency graphs created with the Puck tool [GBZ16] to add extra dependencies to the AST elements.

The development and integration of the EMF Models adapter took eight days and consists of 499 LOC. It should be noted that the complexity and development time of all the above-mentioned adapters must be put in perspective with the benefits in the complex analyses that they enable once integrated in BUT4Reuse.

Regarding the similarity function of the adapters to compare elements, we presented that the results ranged from zero to one. By default the threshold is one, meaning that they need to be identical to be considered the same element during the automatic techniques (e.g., block identification). However, we allow a user-specified threshold that can be useful when using heuristics for similarity that are not using ids or signatures of the elements. We also allow another optional user-specified threshold for requiring the domain expert to manually decide if two elements are equal.

Table 4.1: List of available adapters for framework genericity assessment.

	Java Source Code Adapter [ZHP ⁺ 14]: A folder containing source code in Java. Elements: FSTNonTerminalNode and FSTTerminalNode. FeatureHouse [AKL09] Java source code visitor. Similarity: Feature Structure Tree (FST) [AKL09] positions and names comparison Dependencies: FST nodes dependencies and source code dependencies detected with Puck [GBZ16] Construct: FeatureHouse extraction creating code fragments
	C Source Code Adapter [ZHP ⁺ 14]: Same as Java Source Code Adapter but for the C language
	EMF Models Adapter [MZB ⁺ 15a]: MOF compliant model [OMG06]. Chapter 5 presents all the details.
	File Structure Adapter: Any folder. Elements: File Element and Folder Element. Pre-Order tree traversal of its structure. Similarity: Name and relative path to the initial folder. Optionally file contents based on MD5 hashing Dependencies: Containment dependency Construct: Copy the resources in a given destination
	Text Lines Adapter: Any file that is not a folder. Elements: Line Element. The file is read line by line. Similarity: Levenshtein distance between strings [Lev66] Dependencies: None defined Construct: Append the line strings to an empty file
	Natural language text Adapter: A text file Elements: SentenceElement. Sentence splitter using OpenNLP [Apa10] Similarity: WUP natural language comparison technique [WP94] as implemented in WordNet for Java [fJ15] Dependencies: None defined Construct: A file with the selected sentences
	Requirements Adapter: Requirements Interchange Format (ReqIF) file [OMG13] Elements: RequirementElement. ReqIF model visitor ProR [Ecl14a] The rest is the same as the Natural language text adapter.
	CSV Adapter: Comma-separated values file. Elements: CellElement. Cells' matrix visitor. Similarity: String comparison Dependencies: A cell depends on its row and column Construct: A csv file the corresponding cells and empty cells in the non existing cells
	JSON Adapter: A Json file Elements: ObjectElement, KeyElement, ValueElement, ArrayElement, IndexArrayElement Similarity: The path to the root including array indexes Dependencies: The parent element Construct: A JSON file
	Scratch Adapter: A Scratch script. This adapter is an extension of the JSON adapter Construct: A complete sb2 file with the project.json and all the needed resources (images, sounds etc.)
	Images Adapter: An image file in jpg, bmp, png, gif or ico format. See Section 4.2.1
	Music score Adapter: A MusicXML file [Goo01] Elements: NoteElement. MusicXML parser Similarity: Same note position in the measure, same pitch and duration Dependencies: Position Construct: A MusicXML file
	Graphs Adapter: A GraphML or GML file. Elements: VertexElement, EdgeElement. Blueprints Tinkerpop graph visitor [Tin15] Similarity: Label similarity Dependencies: Vertex dependency based on the edges Construct: Subgraphs creation in GraphML format
	Eclipse Adapter: The folder of an Eclipse package. See Section 4.4.1
	Android Adapter: An Android application (apk file). See Chapter 7

4.3.2 Block identification

Figure 4.6 illustrates the block identification techniques integrated in BUT4Reuse. All of them share a similarity analysis phase using the similarity function defined by the adapter to compare elements. We use gray backgrounds to show the different identified blocks using these techniques.

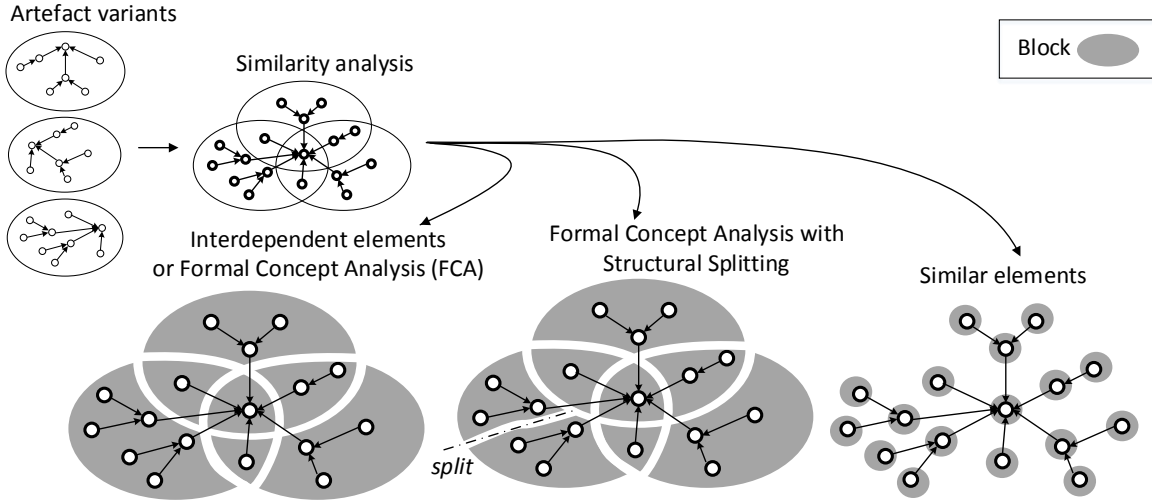


Figure 4.6: Block identification techniques.

Interdependent elements: This block identification technique is borrowed from ExtractorPL [ZFdSZ12]. The idea behind the algorithm is to separate the different intersections of the artefact variants. In Figure 4.6, we illustrate it on the left side of the figure, obtaining one block for each of the intersections of the artefact variants. This technique was used in the illustrative example of image variants in Section 4.2.2.

This technique is based on a formal definition of a block that uses the notion of interdependent elements. Interdependent elements are defined as follows: Given a set A of artefacts that we want to compare, two elements (of artefacts of A) e_1 and e_2 are *interdependent* if and only if they belong to exactly the same artefacts of A . Therefore, e_1 and e_2 are interdependent if the two conditions in Equation 4.1 are fulfilled.

$$\begin{aligned} \exists a \in A \quad e_1 \in a \wedge e_2 \in a \\ \forall a \in A \quad e_1 \in a \Leftrightarrow e_2 \in a \end{aligned} \tag{4.1}$$

Since interdependence is an equivalence relation on the set of elements of A , when using this algorithm, the following definition can be provided for a block: Given A a set of artefacts, a block of A is an equivalence class of the interdependence relation of the elements of A .

Formal Concept Analysis (FCA): FCA groups elements that share common attributes. A detailed explanation about FCA formalism in the same context of extractive SPL adoption can be found in Al-Msie'deen *et al.* [ASH⁺13a] and Shatnawi *et al.* [SSS16]. FCA uses a

formal context as input. In our case, the entities of the formal context are the artefact variants and the attributes (binary attributes) are the presence or absence of each of the elements. With this input, FCA discovers a set of *concepts*. The concepts containing at least one element (“non empty concept intent” in FCA terminology) are considered as a block. At technical level, we implemented FCA for block identification using Galatea.ⁱⁱⁱ

It has been suggested that the interdependent elements technique presented before uses an algorithm similar to Formal Concept Analysis [GW97]. Our experimental results confirm that the same blocks are obtained using both techniques. However, the interdependent elements algorithm orders the blocks by frequency in the artefact variants, while with FCA a post-processing of the retrieved blocks is needed to achieve this.

FCA and structural splitting: This technique extends FCA by further splitting the obtained blocks based on information regarding the structural dependencies of the elements. Concretely, it detects groups of elements within a block that do not depend on other elements of the block. In Figure 4.6, we show how the block in the bottom left is split into two blocks because the two groups are not connected through dependencies within the block. This technique can be useful in distinguishing potentially unrelated groups of elements that could belong to different features (specially when the artefact elements are strongly connected and the blocks are large in number of elements). On the contrary, if the elements are not highly connected, we risk obtaining a lot of blocks with few elements.

Similar Elements: This technique provides the finest granularity where each element, after the similarity analysis phase, corresponds to one block. In Figure 4.6 we show how each block is associated to one element. By using this algorithm we create large amount of blocks to be used for testing or for delegating the analysis to the techniques of the following extractive SPL adoption activities.

4.3.3 Feature identification

For feature identification, the domain experts analyse the elements of each of the identified blocks to try to map them to features. This means that, currently, it is a manual process only supported by the visualisations. In some cases, feature identification could be a trivial activity if block identification techniques could end up with blocks directly associated to features. However, in other cases, the identified blocks could be related to feature interactions or noise introduced by independent evolutions of the artefact variants (e.g., bug fixes) requiring further analysis.

Constraints discovery techniques could help to spot blocks containing coordination elements in case of feature interactions. These techniques can discover relations among the features involved in the feature interaction. For example, structural dependencies can be discovered among the block of coordination elements and the features. Also, by mining the existing configurations we can find that a block only appears when some features are present.

ⁱⁱⁱGalatea Formal Concept Analysis library: <https://github.com/jrfaller/galatea>

4.3.4 Feature location

For feature location, the current implementation consists in trying to map blocks to features. Apart from the core of each feature, we are also interested in locating coordination elements belonging to feature interactions. Before applying feature location techniques, in order to locate these coordination elements, we implemented a method for pre-processing the feature list. This method can be optionally used to automatically include “artificial” features related to pairwise interactions, 3-wise interactions or feature negations.

Given that we introduced the notion of the *confidence* of the located feature with a value from zero to one, we allow the definition of a user-specified threshold. By default, the threshold is one. We integrated the following set of feature location techniques:

Feature-Specific: This heuristic is based on the idea that, for a given feature, the relevant blocks are those that are *always* present when the feature is present. Equation 4.2 shows how it is calculated. For each feature and block pair, we calculate, from the artefact variants implementing the feature, the percentage that it also contains the block. A percentage of 100% for a given block and feature means that the block *always* appear when the feature is present in the artefacts.

$$located(f_i, b_i) = \frac{| artefacts(f_i) \cap artefacts(b_i) |}{| artefacts(f_i) |} \quad (4.2)$$

Figure 4.7a shows an example assuming that FCA was applied for block identification and that we are trying to locate a given feature (F1) that we know it is implemented in two artefacts (A1 and A1). We can observe how we only reach 100% in the blocks intersecting A1 and A2.

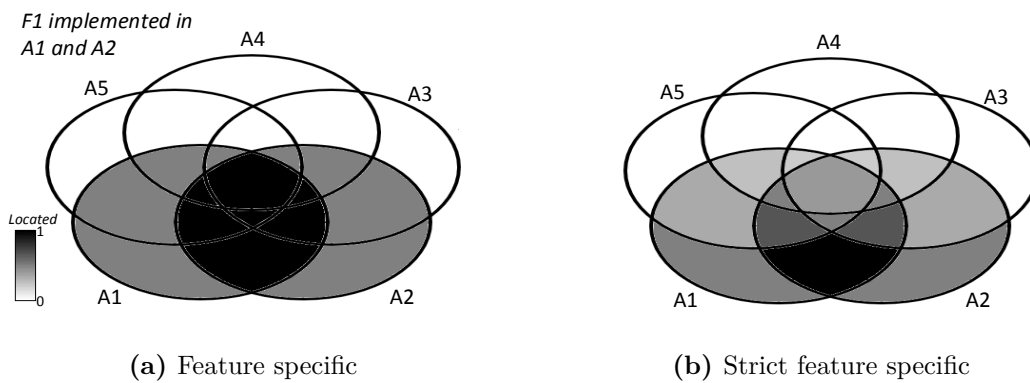


Figure 4.7: Feature location results of a feature present in A1 and A2 using two feature location techniques.

Strict Feature-Specific (SFS): SFS is more restrictive than feature-specific by following two assumptions: A feature is located in a block when 1) the block *always* appears in the artefacts that implements this feature and 2) the block *never* appears in any artefact that does not implement this feature. The principles of this feature location technique are similar

to locating distinguishing features using diff sets [RC12b]. Equation 4.3 shows how SFS is calculated. Figure 4.7b shows an example of this technique where we can observe that this technique “penalizes” the blocks that are included in artefacts which do not include this block (i.e., A3, A4 and A5). 100% is only obtained in the block that intersects A1 and A2 and no other artefact.

$$located(f_i, b_i) = \frac{| artefacts(f_i) \cap artefacts(b_i) |}{| artefacts(f_i) \cup artefacts(b_i) |} \quad (4.3)$$

The next following feature location techniques use information retrieval (IR) techniques. The intuition behind these techniques is based on the fact that feature names and descriptions contain useful information that can be used to associate features to elements. Therefore, words (also known as terms) are extracted from feature names and descriptions as well as from the elements. Several techniques are proposed to filter and pre-process these words to reduce the amount of words (e.g., synonyms). Some of these filtering techniques are explained in Chapter 8.

Latent Semantic Indexing (LSI): LSI is a technique for analyzing relationships among a set of documents [DDL⁺90] that has been used in feature location in the context of extractive SPL adoption [ASH⁺13c, SSD13, XXJ12]. The name and description of a feature are extracted to create a term query. Then, a term-document matrix is created and the cosine similarity is calculated with the words extracted from the elements of the block. Given that the amount of words can be large, LSI techniques often rely on parameters to reduce this amount. In our implementation, this can be configured for using a user-specified percentage of the words. Alternatively, the user can select a fixed number of words ignoring the least frequent ones. Figure 4.8 illustrates the LSI technique by showing how a feature has associated words (the tag symbol) and how the blocks also have associated words retrieved from their elements. Then, depending on the similarity among these words using LSI, some blocks will seem to be more related to this feature.

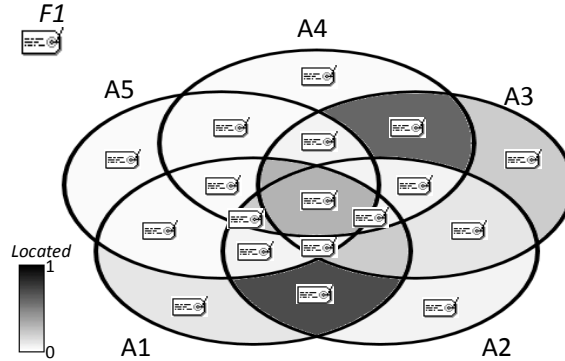


Figure 4.8: Feature location using latent semantic indexing.

SFS and Shared term: The intuition behind this technique is first to group features and blocks with SFS and then apply a “search” of the feature words within the elements of the block to discard elements that may be completely unrelated. For each association between feature and block, we keep, for this feature, only the elements of the block that have at least one meaningful word shared with the feature. That means that we keep the elements whose *term frequency* (tf) between feature and element (*featureElementTF*) is greater than zero. For clarification, *featureElementTF* is defined in Equation 4.4 being f the feature, e the element and **tf** a method that just counts the number of times a given term appears in a given list of terms.

$$featureElementTF(f, e) = \sum_{term_i \in e.terms} tf(term_i, f.terms) \quad (4.4)$$

Figure 4.9 illustrates, on the left side, how for a given feature, we have associated words and how, from a block obtained with SFS, we discard elements that do not share any word with the feature.

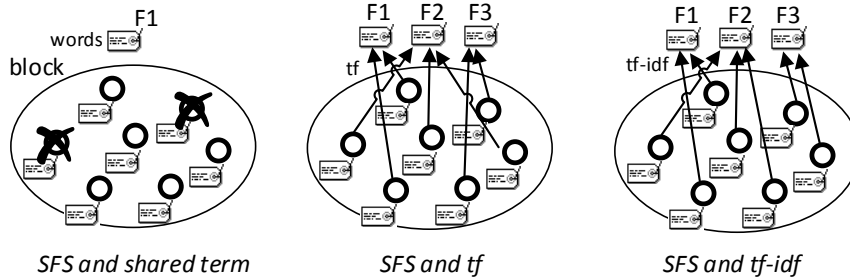


Figure 4.9: Three different feature location techniques using SFS and term frequency.

SFS and Term frequency: After employing SFS, this technique is based on the idea that all the features assigned to a block compete for the block elements. The feature (or features in case of drawback) with higher *featureElementTF* will keep the elements while the other features will not consider this element as part of it. Figure 4.9 illustrates this technique in the center of the figure. Three features compete for the elements of a block obtained with SFS, and the assignment is made by calculating the **tf** between each element and the features.

SFS and tf-idf: Figure 4.9, on the right side, illustrates this technique. SFS is applied and then the features also compete, in this case, for the elements of the block but a different weight is used for each word of the feature. This weight (or score) is calculated through the *term frequency - inverse document frequency* (**tf-idf**) value of the set of features that are competing. **tf-idf** is a well known technique in IR [SWY75]. In our context, the idea is that words appearing more frequently through the features may not be as important as less frequent words. A more detailed explanation and the equations are presented in Section 8.2.1.

4.3.5 Constraints discovery

In BUT4Reuse, several techniques for discovering constraints can be simultaneously applied. We provide three techniques for discovering constraints among features or blocks.

Structural Constraints Discovery: It consists in analysing the structural dependencies between pairs of features or blocks. Specifically, we identify *requires* ($A \Rightarrow B$) and *mutual exclusion* ($\neg(A \wedge B)$) structural constraints by analysing the structural dependencies defined in the elements. The *requires* constraint is defined when at least one element from one side has a structural dependency with an element of the other side. Formally, and being the same for features (replacing B by F), the definition for the requires constraint is presented in Equation 4.5. *do* stands for dependency object and a dependency object can be an element or an external entity (e.g., in the images example the dependency objects are cartesian coordinates).

$$B_1 \text{ requires } B_2 \iff \exists e \in B_1 : \exists do \in e.\text{dependencies} : do \in B_2 \wedge do \notin B_1 \quad (4.5)$$

Figure 4.10 illustrates, on the left side, an example when an element from a block (B1) has a dependency with an element from another block (B2) and thus a requires constrain exists between B1 and B2.

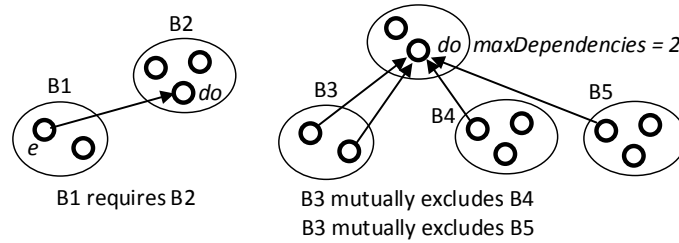


Figure 4.10: Examples of structural constraints discovery.

The *mutual exclusion* constraint discovery is also defined at block and feature levels. The rationale is that, in some cases, a *do* can only tolerate a maximum number of elements depending on it. Figure 4.10 illustrates, on the right side, three blocks (B3, B4 and B5) which have elements that depends on an element *do*. If *do* can only tolerate two elements depending on it then B3 and B4 cannot co-occur, nor B3 and B5. Therefore, B3 excludes B4 and B5. However, B4 and B5 can co-occur because they do not exceed the maximum number of dependencies allowed by *do*.

In our running example, the pixel position dependency object can only tolerate one Pixel element depending on it. Other example are containment references in Classes of EMF Models where an upper bound can be defined according to the cardinality of the reference.

We present how to identify these mutual constraints: Given DOs the set of dependency objects where $do \in DO$, and $dependencyIDs$ the different types of structural dependencies where $id \in dependencyIDs$, the function $nRef$ (number of references) represents the set cardinality of the subgroup of elements in the block that has a structural dependency with a given do . Equation 4.6 presents how $nRef$ is calculated and the formula for mutual exclusion at block level when the set of elements of the blocks are disjoint.

$$\begin{aligned}
 nRef(B_i, do, id) &= |\{e : e \in B_i \wedge do \in e.getDependencies(id)\}| \\
 B_1 \text{ excludes } B_2 &\iff \exists do \in DO, \exists id \in dependencyIDs : \\
 nRef(B_1, do, id) + nRef(B_2, do, id) &> do.getMaxDependencies(id)
 \end{aligned} \tag{4.6}$$

A-priory association rule: This technique uses association rule learning. Concretely we integrated the *A-Priory algorithm* as previously evaluated for this purpose in the SPLE literature [LMSM10]. In comparison to the structural constraints technique that suggests constraints after internally analysing the elements and their structural dependencies, this approach mines the relationships of the presence or absence of the blocks in the artefact variants (i.e., the configurations of the existing variants). It is worth mentioning that using this technique, the discovered constraints will restrict the configuration space to the feature configurations of the initial artefact variants. In other words, any feature combination that was not previously present in the mined artefact variants will be considered as invalid. Our implementation of this technique uses the Weka data mining library [HFH⁺09].

FCA-based constraints discovery: When applying FCA, concepts are related through concept lattices which represent potential relations among the blocks. This technique has been presented and evaluated in research on extractive SPL adoption [RPK11, AmHS⁺14, SSS16]. We implemented the requires dependency when there is a sub-concept that requires a super-concept according to the identified lattices.

4.3.6 Feature model synthesis

This activity is covered by BUT4Reuse with two simple implementations. Currently, the synthesized feature models are exported to FeatureIDE [TKB⁺14].

Flat feature diagram: This technique creates a FM without any hierarchical information among features. It is based on the creation of an abstract root feature where all the features are added as subfeatures. Regarding the constraints, all of them are added as cross-tree constraints.

Alternatives before Hierarchy: This heuristic is based on calculating first the Alternative constructions from the *mutual exclusion* constraints, and then creating the hierarchy using the *requires* constraints. The constraints that were not included in the hierarchy are added as cross-tree constraints. Figure 4.4 showed the result of this heuristic in the case of the identified mutual exclusions in the images example.

4.3.7 Reusable assets construction

The construction is the responsibility of the adapter that will use the elements associated to features to create the reusable assets. Extractive SPL adoption approaches are said to be semantically correct when they can generate exactly the original products [RC12a]. Once the features are located, BUT4Reuse has a functionality to re-generate the initial products to check this property.

4.3.8 Visualisations

BUT4Reuse provides a set of visualisations.

Bars Visualisation: Bars visualisations are used for visualising crosscutting concerns in aspect oriented software development [Ecl14b] or for visualising source code clones [TGB06]. In our context, bars are used for displaying several types of information. Concretely, it can display how elements are distributed on the artefacts, how blocks are distributed on the artefacts, how features span in the blocks and how blocks map the features. Figure 4.11 presents a bars visualisation example showing how blocks are distributed across the mined artefacts. On the left side there is a bar for each of the variants, and on the right side, there is the list of identified blocks. The height of each bar is proportional to the number of elements in the artefact and each stripe is colored accordingly to the associated block. In this example, they are vending machines EMF Models [IBK11, MZKT14] and the order of the stripes represents the sequence of elements as returned by the adapter, which for the EMF Models adapter is a tree-traversal of the model.

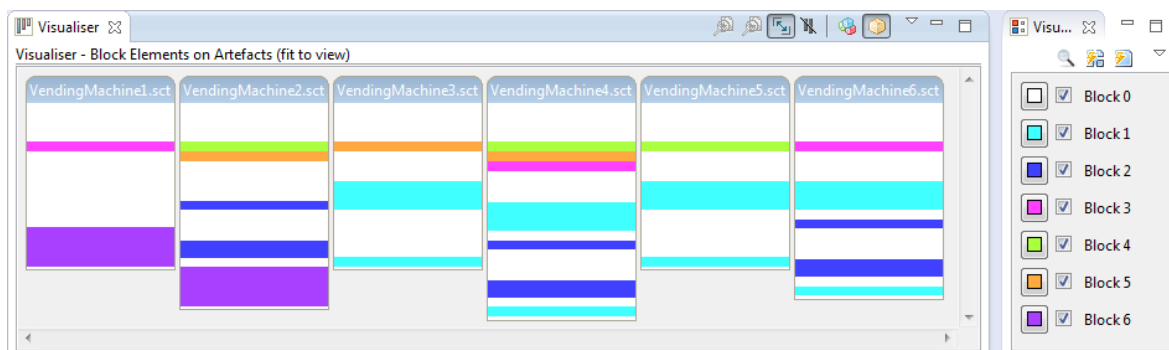


Figure 4.11: Visualisation showing the Blocks (colors) on the artefacts (bars).

Feature location heat-map: Other currently available visualisations rely on *Heat Maps* where larger values in a matrix are represented by dark squares and smaller values by lighter squares. Feature location heat-map visualises the relations among features and blocks to help in feature location. Figure 4.12 presents a matrix that relates known features on Vending Machines artefacts to identified blocks on this set of artefacts. The matrix values that define their color is based on the calculation from the feature-specific feature location technique. For example, for feature Coffee, block 0 and block 4 have 100% because these blocks are present in all the artefacts where we have Coffee. The heat-map is enriched with red location marks

to show the results of the selected feature technique that can be other than feature-specific. In this case, the red marks correspond to SFS meaning that, in the case of Coffee, block 0 appears in other variants that do not have Coffee, and block 3 does not appear in any variant does not have Coffee. Therefore, the red mark appears only for block 3.

	Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6
Base	1.0	0.66	0.5	0.5	0.5	0.5	0.33
Coffee	1.0	0.66	0.33	1.0	0.66	0.66	0.33
Tea	1.0	0.66	0.33	0.66	1.0	0.66	0.33
Soda	1.0	0.66	1.0	0.33	0.33	0.66	0.33
Coins	1.0	1.0	0.5	0.5	0.5	0.5	0.0
Card	1.0	0.0	0.5	0.5	0.5	0.5	1.0
RingTone	1.0	0.66	0.66	0.66	0.66	1.0	0.33

Figure 4.12: Relation of Blocks and Features regarding their presence in the artefact variants displayed using a Heat Map visualisation.

Graph visualisations: BUT4Reuse provides three kinds of graph-based visualisations. The first of them is a graph where the nodes are all the elements and the edges are the dependency relations among them. An example of this graph is presented in Section 4.4.1 at Figure 4.13. The second one is a graph where the nodes are the identified blocks and the edges are the identified constraints among them. An example is shown in next chapter in Figure 5.7. The third one is a graph where the nodes are the features wanted to be located and the edges are the identified constraints between them. In all these graphs, nodes and edges are labelled with different attributes for easy graph manipulation.

Pruned Concept Hierarchy: We provide a visualisation of the pruned concept hierarchy [Pet01], also known as AOC-poset. The hierarchy is calculated through the concept lattice obtained through FCA and it has been used during extractive SPL adoption [XXJ12, ASH⁺13a]. Figure 7.2 shows an example further discussed in Chapter 7.

Feature Relations Graphs (FRoGs): FRoGs is detailed in Chapter 9 and it is used for constraints discovery.

VariClouds: This visualisation, based on word clouds, is used during the feature identification activity to suggest feature names to domain experts. It is described in Chapter 8.


4.4 Experiences and evaluation with the Eclipse case study

In this section we discuss the practical usage of the realization of the framework in BUT4Reuse. First, we quantitatively evaluate the effort for integrating new techniques and adapters. Then, we detail an SPL adoption scenario building on the case study of Eclipse variants. We perform a qualitative evaluation in a controlled experiment scenario with master students that designed and developed the Eclipse Adapter. We also present the results of the usage of this Adapter to discuss the benefits of an extensible framework.

4.4.1 Design of the Eclipse adapter

Eclipse [Ecl16a] is an integrated development environment providing tool-sets for a wide range of software development needs. Each of this tool-sets is called an Eclipse *package*. The official Kepler release of Eclipse is a family of twelve default packages: *Standard*, *Java EE*, *Java*, *C/C++*, *Scout*, *Java and DSL*, *Modeling Tools*, *RCP and RAP*, *Testing*, *Java and Reporting*, *Parallel Applications* and *Automotive Software*^{iv}. We targeted the extractive SPL adoption taking these packages as existing variants.

An Eclipse package is based on a folder that contains the executable and a set of folders and configuration files. Two relevant folders are the *plugins* folder (containing the installed plugins), and the *features* folder (containing information about the features present in the variant). We purposely ignored the information that the features folder could provide and we used it only for discussing the results of the extractive SPL adoption activities.

 Chapter 6 provides more details on Eclipse and Eclipse releases. The detailed information in that chapter is not necessary to understand this experiment which focus on the Kepler release.

Eclipse adapter design and implementation

A BUT4Reuse adapter for Eclipse was designed and implemented in three weeks of development by a group of eight master students that received a formation of six hours on BUT4Reuse principles. They followed the tasks presented in Section 4.2.1.

- Task 1: *Elements identification*. The elements that compose an Eclipse package are the Plugin elements (the plugins) and the File elements (all the resources of an Eclipse package that are not the plugins). In order to decompose an artefact, the adapter performs a tree traversal of the Eclipse root folder obtaining the Plugin and File elements.
- Task 2: *Structural dependencies identification*. A Plugin element may depend on other plugins. Each plugin has meta-data (bundle manifest declaration) which declares its required plugins. Concretely, we considered the *static non-optional* dependencies defined in the *Require-Bundle* set. Therefore the Plugin element will structurally depend on the Plugin elements of these plugins. The id assigned to this dependency type is *requiredBundle* and it has no upper bound because there is no limit in the number of plugins that can require a plugin. The File elements depend on their corresponding parent File element and the defined dependency type's id is *container*. There is no upper bound as the folder can contain any number of files or folders.
- Task 3: *Similarity metric definition*. The similarity between Plugin elements is implemented comparing the plugin identifiers (ids). For File elements, the similarity is implemented comparing their file path relative to the Eclipse root folder.

^{iv}<http://eclipse.org/downloads/packages/release/Kepler/SR2>

- Task 4: *Reusable assets construction*. The construction of a set of elements is implemented by replicating the plugins and files associated to these elements. Also, there is a configuration file named *bundles.info* that is adjusted if present during the construction. This final adjustment leads to completely functional Eclipses created through systematic reuse.

The Eclipse adapter, integrated in BUT4Reuse, consists of 471 LOC. Because of the presented design decisions, there are some known limitations: 1) we only identify Plugin elements that are in the Eclipse plugins folder. Other technical mechanisms exist like using the *dropins* folder or the *bundle pooling* mechanism. However, they are not used in the Eclipse official packages so it was not considered a relevant limitation, 2) the similarity metric among Plugin elements does not consider plugin versions, meaning that two plugins with the same ids but different versions are considered the same Plugin element. This situation happens with 23 plugins out of more than two thousand plugins present in the twelve variants and only in two of them they were major version changes. Finally, 3) other technical methods to define structural dependencies such as *Import-Package* or *x-friends* are not considered.

Lessons learned: We consider that the learning curve in the framework principles and the basic usage of BUT4Reuse (six hours) is acceptable. The students started to discuss in terms of elements and blocks quickly. We have also observed that the effort is bigger in the design of the Adapter than in the implementation itself. Before starting the implementation, the students needed to obtain an in-depth knowledge of many notions and terminology of Eclipse. This corroborated our experience with other adapters, such the JSON and Scratch adapter shown in Table 4.1 which were implemented by another student. Defining the granularity of the elements and the similarity metric, or identifying how to get the information of the dependencies, represented a difficult decision-making process with many trade-offs.

4.4.2 Results and discussions

We report the results of the different layers defined in Section 4.2.3 and we discuss the implication of the layers' extensibility. We used the twelve Eclipse Kepler SR2 Windows 64-bits packages as artefact variants. The average number of elements per artefact is 1483. The Eclipse adapter developed by the students takes eleven seconds to decompose the twelve Eclipse packages into Plugin and File elements. The reported performance in execution time is the average of ten executions calculated using a laptop Dell Latitude E6330 with a processor Intel(R) Core(TM) i7-3540M CPU @3.00GHz 3.00GHz, 8GB RAM, with Windows 7 64-bit.

Visualisation: Figure 4.13 shows an example of a visualisation that helped to understand the Eclipse structure in terms of the defined elements. Concretely, it presents the result of the Graph visualisation of the elements of the *Modeling Tools* package. The white nodes are Plugin elements and the grey nodes are File elements. The edges correspond to the defined structural dependencies. The size of the nodes is related to the number of elements that depends on this node (in-degree). The biggest grey File element node corresponds to the mentioned *plugins* folder containing all the plugins. The biggest white Plugin element

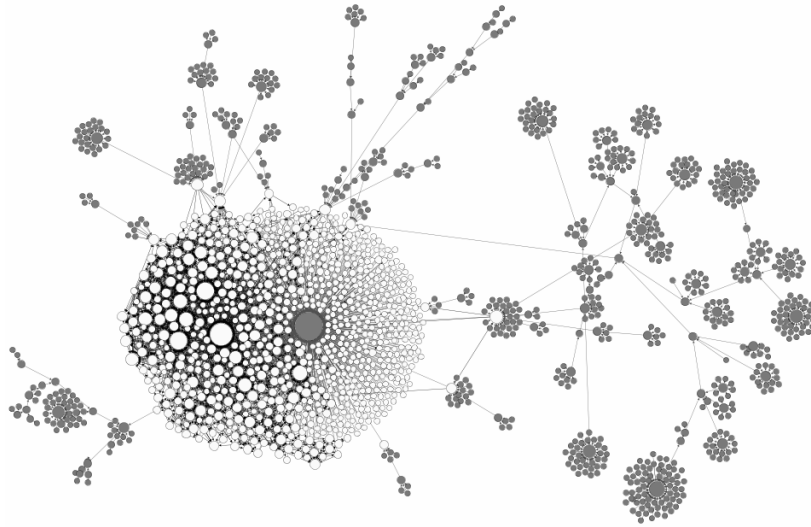


Figure 4.13: An Eclipse package decomposed in Plugin elements (white) and File elements (grey) with the structural dependencies between them.


node corresponds to the `org.eclipse.core.runtime` plugin which is the foundation of the Eclipse platform. Almost all plugins require this plugin to be functional. If we focus on the white nodes, on the left side of the biggest grey File element (*plugins* folder), we can see a set of highly interconnected plugins, and on the right side, a set of plugins with few or no dependencies to other plugins. If we focus on the grey nodes, we can observe the tree-like structures of the File elements.

Block identification: We used the Interdependent Elements technique for block identification as presented in Section 4.3.2. 61 blocks were identified. The average number of elements per block was 68, and the technique took 62 milliseconds.


Feature identification: For this step, we requested the expertise of three domain experts with more than ten years of experience on Eclipse development, who analysed, independently from each other, the textual representations of the elements of the 61 Blocks. They were able to manually identify features by guessing the functionality that the blocks could provide. We present this manual process through an explanation of the feature naming subactivity.

Feature naming during feature identification: The domain experts were able to select a name for each identified functionality. The feature identification process was possible with an average of 87% of the blocks assigned to a named feature. This manual task took an average of 51 minutes. We manually analysed the reported names of all the blocks and the experts' comments. Regarding the coincidences in the names, 56% of the blocks were equally named by the three domain experts. In 33% of the blocks there were coincidences in two of them. That ends up with an 11% where there was no coincidence. Also, not all the blocks were easy to name. In 18% of them, at least one domain expert was not able to set a name. According to their comments, the reasons were 1) completely ignoring the plugins or 2) the plugins inside a block had no evident relations among them. The first one is a limitation stemming from the experts' knowledge while the second one is a limitation of the block identification technique. Also, 8% of the blocks corresponded to plugins that are libraries. The three domain experts

mentioned that these blocks cannot be considered as features, but support for actual features. Another 5% of the blocks were considered irrelevant from a functional perspective given that they completely consist of source code plugins (non compiled) found in some distributions but not in others.

 We repeated this feature naming case study with the assistance of a visualisation paradigm for domain experts. The results are presented in Chapter 8.

Feature location: In this case study, feature location consists in locating the plugins associated to each Eclipse feature. For assessment purposes, we programatically mined the Eclipse features of all the Eclipse packages by getting the information from the *features* folder. This provide us the list of features to locate, as well as a ground truth for evaluation of the employed feature location technique.

 Chapter 6 explains the feature location activity in Eclipse packages and the results of four feature location techniques.

Constraints discovery: We used the Structural constraints discovery technique on the 61 blocks. 74790 structural constraints were discovered. That demonstrated how highly interconnected the Eclipse plugins are. The analysis took 88 seconds. We also used the A-Priory association rules (with a limit of 30000 rules to prevent memory issues in the algorithm). The analysis took 0.5 seconds. This technique discovered also *excludes* constraints that are not expected to be true in the context of Eclipse features analysis. This technique is conservative in the sense that it prevents block combinations that are not part of the existing variants. Again, there are trade-offs of using one technique or other. The A-Priory technique, which does not reason on the elements' structural dependencies, is more conservative against possible semantic constraints among the features. A user of BUT4Reuse may decide that selecting this technique is not appropriate for the Eclipse variants scenario because we are not expecting mutual exclusions constraints among features.

Reusable assets construction: Each of the 61 Blocks were constructed separately. The reusable asset consists of a set of files that, if integrated in an Eclipse, can provide some functionality. We evaluated the validity of the reusable assets by re-constructing the 12 Eclipse variants. We compared the file structure from the original and the re-constructed ones. They were the same except for a few cases because of the mentioned limitation of not considering different plugin versions. After manually solving these versioning problems, we manually checked that the Eclipse packages were executable and functional (i.e., the plugins could be started without dependency issues). We further generated non-existent variants from structurally valid configurations according to our discovered constraints. For example, we generated an Eclipse with only the Core block, i.e., the elements that are common to the twelve packages. Also, we generated one package with all the blocks providing, in a single package, the functionality of the twelve variants. We created other Eclipse with the core and CVS versioning system support, and another with the union of the blocks corresponding to the Java and the Testing Eclipse variants.

Feature model synthesis: As discussed before, the blocks were renamed during feature identification. After that, using the two available feature model synthesis approaches, the Flat feature diagram and the Alternatives before Hierarchy heuristic, we created two different feature diagrams. In the second one, the hierarchy was very limited because of the highly interconnected blocks. The presence of the cross-tree constraints were more prominent given that classical feature diagrams only support one parent feature.

4.5 Limitations

In this section we discuss two general limitations of the proposed framework for extractive SPL adoption.

The problem of feature interactions

Apart from planned feature interactions (e.g., the **Logging** feature mentioned in Section 4.2.3), feature interactions can also be the cause of undesired system behavior. In extractive SPL adoption this is an issue when, once the SPL is adopted, we want to derive new products which were not part of the initial configurations. As we mentioned in Section 4.3.7, the extractive SPL adoption is semantically correct when we can generate exactly the original products [RC12a]. After performing the feature constraints discovery activity, the generation of new products beyond the original ones is an issue mainly related to SPL testing [HPP⁺14].

The boundaries of semantic similarity

The third task for designing an adapter described in Section 4.2.2 is related to the definition of a similarity metric among elements. The similarity function can make use of all the information that the adapter can retrieve such as the properties of the element, information about the ancestor elements in case of structured elements, or information about element dependencies to other elements. However, in many cases, calculating the similarity requires to take into consideration the semantics of the elements.

This issue has been tackled in different research communities dealing with different artefact types. In the modeling community, they have categorized different similarity calculation approaches [KDRPP09]: The static identity-based matching assumes that the elements have a unique id. The signature-based matching calculates an id based on several properties. The similarity-based matching assign weights to properties and aggregate related elements (e.g., dependencies among elements). Finally, custom language-specific matching take into account the semantics of the elements. To achieve these semantic comparisons the similarity function is aware of the domain-specific semantics (e.g., UML class models [XS07]).

In the source code clone detection community, textual similarity is used to detect clones of Type I, II or III [RC07] which are, respectively: identical source code fragments, fragments with changes in the name of literals, variables, etc., or fragments where some modifications were made by adding or removing parts. In the Type IV, the fragments do not share textual similarity but they have functional similarity. These ones are called semantic clones.

While designing an adapter for BUT4Reuse requiring semantic comparisons, we will depend on the state-of-the-art of the available semantic comparison approaches for the targeted artefact type. The limitations of the similarity function will be also the limitations of the BUT4Reuse adapter. One example is the requirements and the natural language text adapter that we implemented, shown in Table 4.1. We use a similarity function to compare the meaning of two sentences based on a semantic analysis [WP94] using WordNet [fJ15] which is a database of cognitive synonyms. For analysing the variability in requirements, we used syntactic and semantic similarity while others also rely on parts of behaviors as manifested in the requirements [IRW16]. Semantic analysis requires advanced techniques which are out of the scope of this dissertation. However, we are aware that failing to correctly calculate the similarity can impact negatively the extractive SPL adoption activities supported by BUT4Reuse.

4.6 Conclusions

We introduced a generic and extensible framework for an extractive approach to SPLE adoption. We presented its principles with the objective to reduce the current high up-front investment required for an end-to-end adoption of systematic reuse. The framework can be easily adapted to different artefact types and can integrate state-of-the-art techniques and visualisation paradigms to help in this process. We have presented Bottom-Up Technologies for Reuse (BUT4Reuse) which is our realization of the framework. We demonstrated the generic and extensible characteristics of this realization by presenting a variety of fifteen adapters to deal with diverse artefact types. We also demonstrated the extensibility with techniques and visualisation paradigms that have already been integrated to provide a complete solution. We empirically evaluated the framework integration and development complexity, and its usage, in the scenario of adopting an SPL approach from existing Eclipse variants.

As further work, apart from improving or proposing concrete techniques, there are still many challenges on genericity and extensibility in extractive SPL adoption. Software does not rely on only one type of artefact. For example, a software project uses to contain requirements, design models, source code and test suites. Therefore, we should be able to take into account different artefact types simultaneously. We conducted experiments in this direction, for example, considering files and plugins in Eclipse variants, or source code, files and meta-data in the case of Android applications as we will see in Chapter 7. Also, extensibility in the layers of the framework creates the need of defining guidelines for different scenarios to select the most appropriate techniques and extensions.

EXTRACTION OF MODEL-BASED SOFTWARE PRODUCT LINES FROM MODEL VARIANTS

This chapter is based on the work that has been published in the following papers:

- Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Automating the extraction of model-based software product lines from model variants (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 396–406. IEEE Computer Society, 2015
- Jabier Martinez, Tewfik Ziadi, Jacques Klein, and Yves Le Traon. Identifying and visualising commonality and variability in model variants. In *Modelling Foundations and Applications - 10th European Conference, ECMFA 2014, Held as Part of STAF 2014, York, UK, July 21-25, 2014. Proceedings*, pages 117–131, 2014

Contents

5.1	Introduction	64
5.2	Extraction of Model-based Product Lines	64
5.3	Designing the Model Adapter	66
5.3.1	Elements identification: A meta-model independent approach	66
5.3.2	Structural dependencies identification	68
5.3.3	Similarity metric definition: Relying on extensible techniques	69
5.3.4	Reusable assets construction: Generating a CVL model	70
5.4	Experimental Assessment	73
5.4.1	BUT4Reuse settings for the case studies	73
5.4.2	ArgoUML case study	75
5.4.3	In-Flight Entertainment Systems case study	78
5.4.4	Discussions about MoVa2PL	80
5.5	Conclusion	81

5.1 Introduction

Using the principles of the BUT4Reuse framework described in Chapter 4, we present MoVa2PL (Model Variants to Product Line) where we address the requirements for extracting Model-based Software Product Lines (MSPLs) from model variants. Adopting an MSPL, or any other kind of SPL, will allow practitioners to easily and efficiently propagate changes in one feature to all existing variants by simply re-deriving them automatically. Moreover, the extracted MSPL can be relied upon to efficiently derive new products by combining features.

Challenges of extractive SPL adoption in the modeling scenario.

- *Dealing with model variants.* Analysing and comparing the existing model variants (i.e., models that are used using ad-hoc reuse techniques) to identify commonality and variability in terms of features is an important activity to extract an MSPL. Also, once features are identified and located, and the constraints among them are detected, we need to use this information to construct the operative MSPL.

Contributions of this chapter.

- **BUT4Reuse adapter for models enabling MoVa2PL:** We provide a meta-model independent approach for commonality and variability analysis through the design of a BUT4Reuse adapter. We assume that variants are not independently generated out of the same family of systems. We include information about the element dependencies for the automatic detection of constraints between the identified features. Finally, we propose a method to automatically construct the MSPL. In the realization of our approach, we used the Common Variability Language (CVL) [HWC13] to implement the MSPL.

The remainder of this chapter is structured as follows: Section 5.2 presents an example to illustrate the challenges in the modeling domain. Section 5.3 details the design decisions of the model adapter. Section 5.4 presents experiments based on case studies with large systems and we discuss the approach and limitations. Finally, Section 5.5 concludes this work and outlines future directions.

5.2 Extraction of Model-based Product Lines

In the realm of software engineering, models, which are high-level specifications of systems, have progressively gained importance for researchers and practitioners as the primary artefacts of development projects. Traditionally, modeling has been used in a descriptive way to represent systems by abstracting away some aspects of the systems and emphasizing others [Völ13]. Nonetheless, prescriptive modeling is now trending and is relied upon to automate the generation of products as well as their validations [Sch06]. In this context, models are often

extended, customized or simply reconfigured for use in particular system settings. Thus, an important challenge in Model-Driven Engineering (MDE) is to develop and maintain multiple variants, i.e., similar models, by exploiting the features that the models share (commonalities) and managing the features that vary among them (variabilities) [AK09].

We present in Figure 5.1 our running example illustrating a scenario of UML model variants for different banking systems. This banking system domain and artefacts were used in previous works [ZFdSZ12, ZJ06, ABB⁺02]. In this scenario we have created three models through ad-hoc reuse with variations on the limit of bank withdrawal, the consortium entity and currency conversion. The building of variants for such a simple running example aims to illustrate how time-consuming and error-prone the manual creation of variants can be in real-world complex scenarios.

The first created variant, **Bank 1** UML model, is implemented with information related to currency conversion and consortium. **Bank 2** UML model includes a new requested feature that is the support for a limit in the withdrawal. This new banking system, however, does not need consortium. Thus, to create it, we consider a copy of **Bank 1** where we added one UML property, two UML operations, modified the *name* attribute of a UML operation and removed the UML class Consortium and all related UML elements (one UML class and two UML operations). The needs of **Bank 3**, on the other hand, include limit in the withdraw and consortium, but no currency conversion. To create this variant, we build from a copy of **Bank 2** where we removed all UML elements related to currency conversion (one UML property, four UML operations and one UML class). However, we also selected and copied UML elements from **Bank 1** to complete the implementation of **Bank 3**.

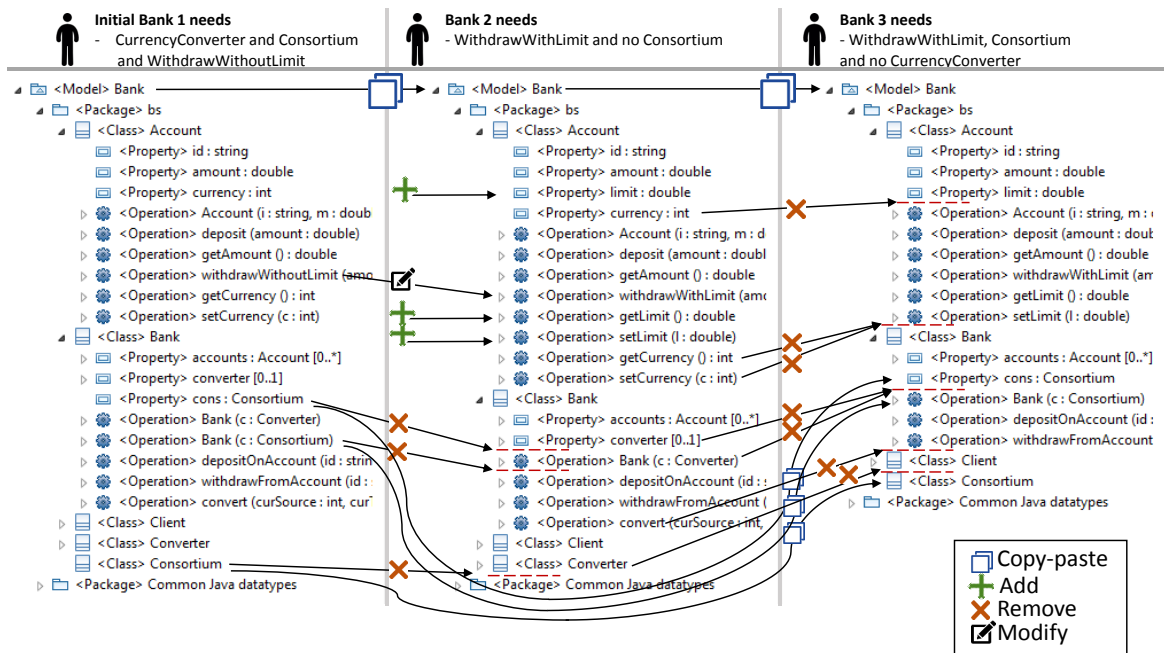


Figure 5.1: Three UML model variants fulfilling different bank system needs and the manual actions for their creation.

The presented manual process quickly ceases to be sustainable if we consider the possibility that continuously creating new variants requires even more effort in finding, selecting and reusing elements from other variants. Furthermore, because of a lack of an explicit formalization of feature constraints, potential inconsistencies in the requested feature configuration for a new variant will be found during or after the variant derivation.

To extract an MSPL from the variants of our running example, feature identification and location would consist in analysing the three UML variants shown in Figure 5.1 to identify the core elements of a banking system, as well as the different features related to currency conversion, consortium and limit support. In addition, we would need to identify the constraints among them. Then, we will exploit the identified features and the existing variants to build the necessary assets for the MSPL. In this case, these assets are the CVL layers presented in Section 3.2 which are the *variability definition* and *product realization* layers. We are concerned with the following research questions:

- RQ1: Based on existing model variants, can we automatically construct a variability definition layer that ensures the validity of the configuration space?
- RQ2: Can we automatically infer a product realization layer for variants of complex systems?

This chapter presents the design of the BUT4Reuse model adapter which aims to provide a solution for the previous questions enabling the extraction of MSPLs.

5.3 Designing the Model Adapter

We report the design decisions during the creation of the model adapter. The following four subsections corresponds to the design tasks described in Section 4.2.1 including technical details about modeling frameworks and CVL.

5.3.1 Elements identification: A meta-model independent approach

Models are artefacts that can be expressed as a sequence of elementary construction operations [BMMM08]. By using the Meta Object Facility (MOF) concepts [OMG06] we are able to decompose any model compliant with the Essential MOF which is a subset containing MOF's core. The Eclipse Modeling Framework (EMF) [Ecl16a] is a meta-modeling framework which is considered an implementation of Essential MOF. EMF is widely used to define domain-specific modeling languages (DSLs). The model adapter decomposes the model in elements, called Atomic Model Elements (AMEs) hereafter. The AMEs in our approach are:

- **Class:** Each DSL defines which types of meta-classes are available. Therefore, it is important not to strictly relate the term class with UML classes. In Figure 5.1 we can observe how, even for UML models, we have other classes such as Packages, Properties or Operations.

- **Attribute:** A class in a DSL can contain attributes relevant to this class. Each attribute will have a value. A typical attribute can be the name attribute of a class which expects a string value.
- **Reference:** Apart from attributes, references are important properties of the classes. Instead of the typed values of the attributes, references are “links” to other classes.

From a technical perspective, we implemented the decomposition of a model variant in AMEs using a pre-order tree traversal of the model by following the containment references. We add each Class and its Attributes and, after that, we add the References once all the Classes are retrieved.

The reflexivity capability of EMF models allows to get information from the meta-model. Concretely, we use this capability to provide a generic approach for dealing with models in MoVa2PL. During the decomposition, we also check, for both attributes and references, that they are not *derived*, nor *volatile*, nor *transient*. If this is the case, we ignore them in the decomposition. For example, if an attribute has the *derived* flag in a given meta-model, it means that its value is automatically calculated from other attributes or by some function, thus we do not add this as Attribute element during the decomposition of the model in AMEs. We also add the condition that their values must have been set before (i.e., non null values or empty lists of referenced elements).

By applying the presented method, Figure 5.2 shows the decomposition of a banking system UML model. This figure shows only an excerpt of the 164 AMEs of this model that contains 67 classes including UML classes, properties, operations, etc. We will show that these principles for decomposing a model are generic and apply for other DSLs such as Capella system engineering models [Pol15] (Section 5.4.3) or Yakindu statecharts [Ite14] (used as case study in Chapter 8).

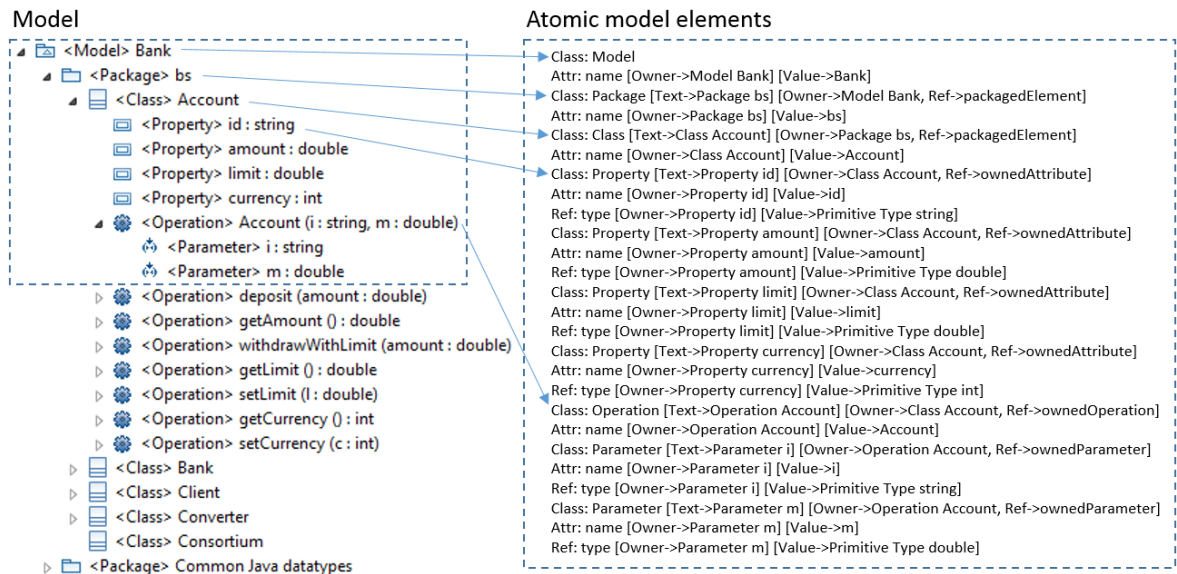


Figure 5.2: Bank UML model and excerpt of its atomic model elements.

5.3.2 Structural dependencies identification

In Section 4.3.5, we explained why the information about the structural dependencies among elements is important to identify structural constraints. A dependency involves a pair of AMEs and it has an id which corresponds to the dependency type. Each dependency type has an upper bound representing the maximum number of times an AME can be referenced with its dependent counterpart.

The AME dependencies are obtained as follows:

- *Class*: A class AME depends on its parent class AME. By parent we mean the container relation (not to be confused with inheritance in the UML sense). The dependency id is the containment reference id as it was given in the meta-model. The upper bound is the containment reference upper bound as indicated in the cardinality defined in the meta-model. For instance, from the running example, the class AME **Operation deposit** depends on the class AME **Class Account** (parent), its dependency id is **ownedOperation** and, in this case, there is no upper bound because the meta-model defines that a UML Class can contain unlimited owned operations.
- *Attribute*: An attribute AME depends on the class AME hosting the attribute. The dependency id is the attribute id which is defined in the meta-model. A class cannot have the same attribute twice, therefore, the upper bound is one. As an example of such dependency, any **Operation** has an attribute AME **name**. In **Operation deposit**, the attribute AME **name** depends on **Operation deposit** and there can only be one attribute **name**.
- *Reference*: A reference AME depends on the class AME hosting the reference. The dependency id is the reference id of the host class. The upper bound in a dependency with the host is one as it was the case for the attribute AME. However, a reference AME also depends on each of the class AMEs referenced. In this case, the hard-coded value **referenced** is used as id for the dependencies to the referenced class AMEs. Given that a class AME can be referenced as many times as desired, the upper bound in a dependency with the referenced classes is unlimited. From the running example, the reference AME **Type** of **Parameter amount** depends on **Parameter amount** (the host) and also depends on the class AME **Primitive Type double** as it is referenced.

Figure 5.3 shows the AMEs of **Bank 1**, **Bank 2** and **Bank 3** and their corresponding dependencies using a directed graph visualisation. In this case, these graphs are clockwise-based directed graphs. The direction of the edges between two elements is defined through the clockwise direction of the arc. For example, in Figure 5.3a, we show how **Package bs** depends on **Model Bank** (the root element of the model). We can see how the class AMEs (in black color) are surrounded by the attribute AMEs (light color) and reference AMEs (dark color) that depend on each class AME. We can also see, as black edges, how some class AMEs depend on other class AMEs through a containment relation (given that a class depends on its parent class AME). The graph shows how attribute AMEs only depend on their corresponding

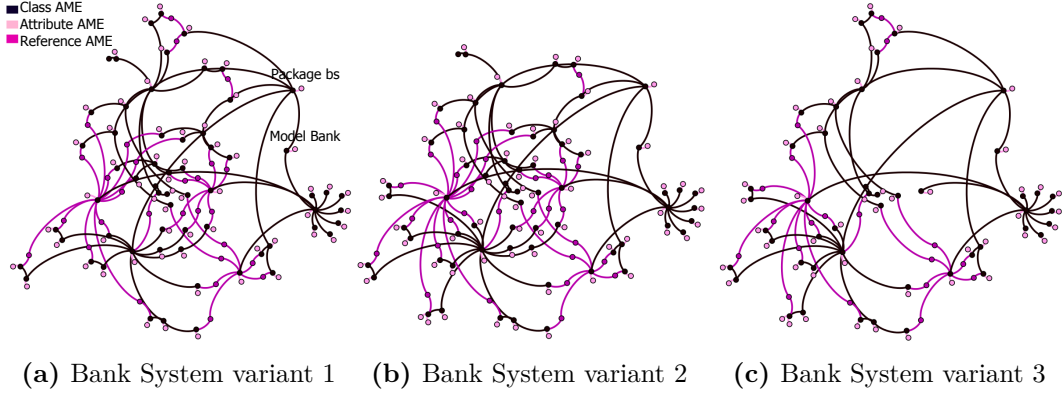


Figure 5.3: Visualisation of the decomposition of three model variants into Atomic Model Elements. The direction of the edges between two elements is defined through the clockwise direction of the arc.

host class AME. On the contrary, we can see the reference AMEs that, besides their host class AME, also depend on the referenced class AMEs. All the graphs shown in this chapter are automatically generated by BUT4Reuse (see Section 4.3.8 on visualisations).

5.3.3 Similarity metric definition: Relying on extensible techniques

In Section 4.5 we discussed the different ways to calculate similarity between elements. For the comparison between AMEs, we rely on existing techniques of model comparison that are highly extensible. We used EMF DiffMerge [Ecl16c] that enables the comparison of two model scopes using different match and diff policies. These techniques allow dealing with meta-model peculiarities or implementing specific comparison purposes. By using DiffMerge, we provide the means to integrate domain-specific similarity calculations if needed. MoVa2PL provides a default similarity metric for AMEs. More precisely, using the classification by *Kolovos et al.* [KDRPP09] MoVa2PL default model comparison behaviour is static identity-based matching. By using the mentioned extension mechanisms, it is possible to contribute signature based-matching or others if required.

We explain how we designed the *default* similarity metric for the model adapter, which is a boolean method (i.e., 0 for different and 1 for equal).

- *Class*: Two class AMEs are equal if we isolate each of the classes in a scope that contains only these elements and the default EMF DiffMerge policy returns no difference in the comparison. The Match policy used by default consists in trying to compare an attribute tagged as id for the meta-class of the class. In EMF meta-models, this information is sometimes included (e.g., an id attribute or a unique name attribute). If no id attribute is defined, it tries to infer an id by directly looking in the serialization mechanism of the artefact (e.g., ids in a XMI file). If no id is found, it calculates a URI as id for the comparison. Notice that this default policy ignores all the attributes and references of the class (except the id attribute if defined). This way it will not necessarily state that a class is different if they have different attributes or references.

- *Attribute*: In EMF, each defined attribute in the meta-model has an identifier (for example *Operation_Name* is the id for the attribute Name of the Operation meta-class). Two attribute AMEs will be the same if they deal with the same attribute id and if the owner classes of the attribute are the same. Finally, the diff policy is in charge of deciding whether the values of the attributes should be considered equal for this attribute id. The default implementation of the diff policy just performs an *equals* operation on the values.
- *Reference*: As well as with attributes, two reference AMEs are equal if they share the same reference id and if the owner classes are the same. Then we check that the referenced classes are the same. If it is an ordered reference, the referenced AMEs must appear in the same positions while if it is not ordered, it is only needed that all elements are present in the other reference AME.

5.3.4 Reusable assets construction: Generating a CVL model

As presented in Section 2.1.2, different strategies can be selected for implementing an SPL ranging from positive or negative variability (or hybrid approaches) [VG07]. These strategies have been already analysed in the context of MSPLs where positive variability is referred to as the additive approach and negative variability is referred to as the subtractive one [Zha14, PKGJ08]. The additive approach relies on a minimum base model composed only by the core of the family of models, i.e., the model elements that are common to all model variants. Then, library models are required containing the model fragments to be added to the base model. On the contrary, the subtractive strategy consists in constructing the maximum base model and then removing the model elements related to each of the non selected features. Hybrid approaches use a mix between subtractive and additive strategies by using library models and still leaving the possibility to subtract from the base model.

In MoVa2PL, we rely on a subtractive strategy. It is possible to construct a maximum base model even if the resulting base model violates cardinalities defined in the relations among meta-classes (i.e., cardinality upper bounds). Despite that the base model is structurally invalid, the resolution model will be responsible to operate in the base model to bring it to a valid state. We explain how the CVL layers, described in Section 5.2, are created. Once these layers are constructed, the used CVL tool [SIN15] provides an engine to automatically transform the base model in a resolved model with the selected features.

Variability model construction: The variability model is created using information from the identified features and the discovered constraints. Figure 5.4a shows the CVL variability model created from the three model variants of the running example. The four steps for its creation are as follows:

1. Identified features are added as well as their negations. Feature negations are needed as a technical mechanism to differentiate the actions of the resolution layer. The negations will trigger the removal of model elements from the base model. We use the flat feature model synthesis presented in Section 4.3.6.

2. Discovered structural constraints are added as propositional logic formulas.
3. Mutual exclusion constraints are added to avoid selecting a feature and its own negation.
4. Configurations, in terms of features in the existing model variants, are added. In CVL terminology, these configurations are called *resolution elements*.

In the running example, as shown at the bottom of Figure 5.4a, three configurations, corresponding to existing variants, are created in the fourth step. However, with the features and constraints identified and formalized as result of the first three steps, there are eight possible valid configurations. Therefore five additional models can be derived using different combinations of feature selections.

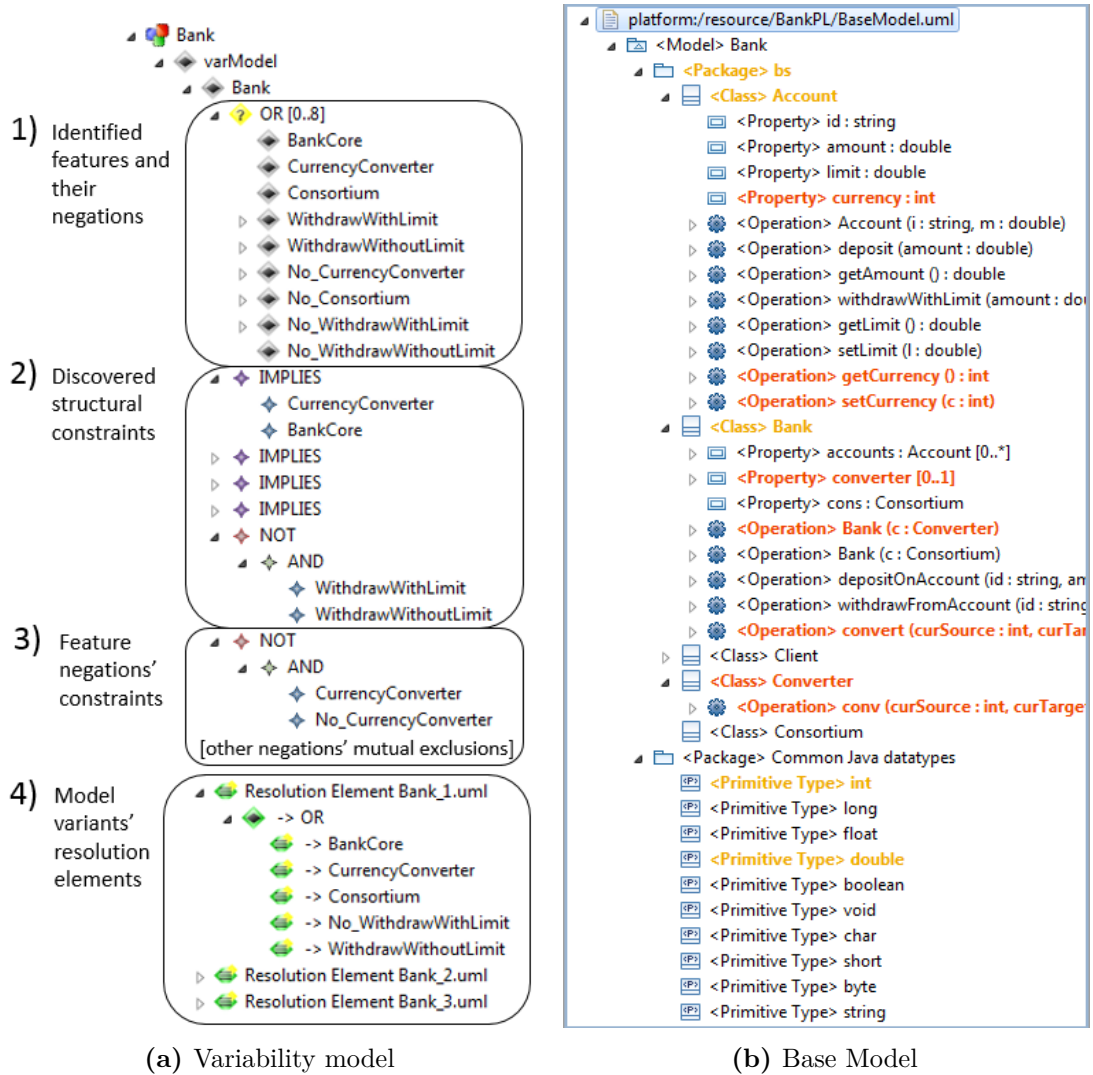


Figure 5.4: Excerpts of the CVL models implementing the banking systems MSPL.

Base model construction: The base model creation, which in our case is a maximal model, can be considered as a realization of an n-way model merge [RC13a]. We start from the class AME marked as the initial resource (i.e., the root) and we automatically construct the base model from scratch with the information contained in each AME using: an in-depth tree traversal of the containment dependencies creating the classes and setting their attributes,

and a second phase where the references are set to the corresponding classes in the base model. In this process, there is neither need to consider if upper bounds are being violated, nor if there are attributes in the base model that are already set. In these cases, as discussed before, the resolution model is responsible for providing the means to adjust it at derivation time. Figure 5.4b shows the base model obtained from the variants of our running example.

Resolution model construction: Given that we are following a subtractive strategy, each feature negation will be resolved by removing the classes associated to this feature negation. To do so, we create three CVL entities for each feature negation: A placement fragment, a replacement fragment and a fragment substitution element. The placement fragment defines the model elements from the base model that will be replaced with the replacement fragment. To implement deletion in our subtractive strategy, this replacement fragment consists of an empty fragment. Finally, the fragment substitution element is only a link to relate the placement with the replacement fragment.

In the CVL implementation that we use [SIN15], the resolution layer is defined inside the variability model itself so the resolution information is contained in each of the features presented before in the variability layer. Figure 5.5 shows an excerpt of the resolution model for our running example. In the case of `No_CurrencyConverter` we can see a Placement fragment encompassing all the classes related to `CurrencyConverter`. In the previous Figure 5.4b, the classes inside the placement fragment of `No_CurrencyConverter` are highlighted in dark color. The placement also includes the `FromPlacement`. In the `FromPlacement` we specify all the classes that are referenced from any class of the placement classes and from any class in the contents of the placement classes. These classes are highlighted in light color in Figure 5.4b. This placement information allows, at derivation time, the removal of the `CurrencyConverter` feature by its substitution with the empty replacement.

Regarding attribute and reference AMEs, if the class AME hosting the attribute or reference AME is not in the same feature, we need to add a placement, replacement and substitution elements to the resolution information of the feature. For example, in the `WithdrawWithoutLimit`

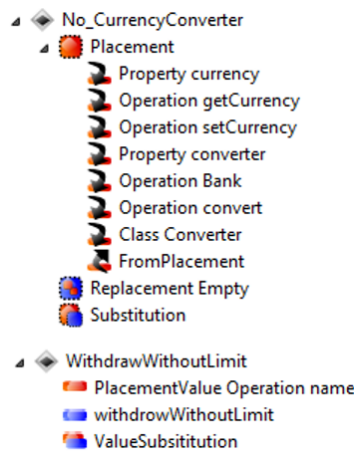


Figure 5.5: Excerpt of the CVL resolution model. When `No_CurrencyConverter` is selected, the model elements in its placement will be removed. If `WithdrawWithoutLimit` is selected, the attribute name of a class will be changed with a defined value.

feature, we add a *Placement value* in the **name** attribute of the corresponding **Operation class** and we add a *Replacement value*, as shown at the bottom of Figure 5.5, with the string of the attribute AME. At resolution time, if **WithdrawWithoutLimit** is selected, this attribute value will be assigned. For reference AMEs, the same approach is followed but Object placement, replacement and substitution are used.

5.4 Experimental Assessment

In this section we discuss the assessment of MoVa2PL in two case studies. First, we describe the BUT4Reuse settings for conducting the extractive SPL adoption. Then, we present the characteristics of the case studies and the extracted MSPL. Finally, we summarize the evaluation of the MoVa2PL where we checked its efficiency to extract an MSPL for large models, and its effectiveness to derive the initial models and new valid model variants.

5.4.1 BUT4Reuse settings for the case studies

Block and feature identification: Block identification is performed by computing the interdependence relations among AMEs as explained in the interdependent elements algorithm in Section 4.3.2. Feature identification is a process where domain experts will manually review the elements from the identified blocks to map them with the features of the system.

Feature constraints discovery: A structurally valid model is a model that does not violate any constraint defined in the meta-model such as cardinality of the model references or the non existence of dangling elements (i.e., an element without parent). Semantic validity of model variants, which is checked by domain experts, is out of the scope of the structural analysis. As described in Section 5.3.2, we augment the information of AMEs with the dependencies among AMEs in all variants. Thus, once the features are already identified, we can perform the structural constraints discovery activity which will allow a more reliable definition of the variability model for the MSPL. The binary structural constraints discovery technique is explained in Section 4.3.5. For the discovery of structural constraints among features we reason on the dependencies of the AMEs. We provide details, separately, on the requires and excludes constraints discovery in this context.

Requires constraints discovery: To avoid dangling elements in derived models after the extraction of the MSPL, we identify the *requires* constraints that will assure that all the classes (except the root) will have a parent. We also identify the *requires* constraints between any pair of features when one feature needs the other (i.e., the feature is referenced).

In Figure 5.6 we show the blocks that were identified by computing the interdependence between AMEs. The **BankCore** feature, which corresponds to the first block (Block 0), comprises most of the AMEs. Specifically, it comprises those common to all model variants. On the contrary, **WithdrawWithoutLimit** is based only on one AME (in the bottom left corner of the figure) that is the attribute AME of the **name** of an **Operation**. Each AME

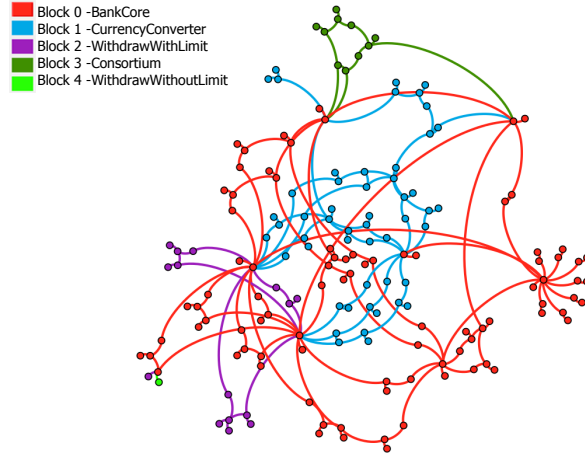


Figure 5.6: Identified blocks in the Bank system variants which are mapped to features. The dependencies among their associated Atomic Model Elements are used for constraints discovery.

shows also its dependencies with other AMEs through the edges (clockwise-based directed graph). We observe that most of these dependencies exist between AMEs corresponding to the same feature (intra-feature structural dependencies). However, we observe dependencies between AMEs of different features (inter-feature structural dependencies) that in this case they are all in the direction to the **Block 0** of the **BankCore** feature. As defined in the binary structural constraints discovery, we identify the *requires* constraint between two features when at least one AME from one feature has a structural dependency to an AME of the other feature. Figure 5.7 thus shows the discovered *requires* constraints in the running example.

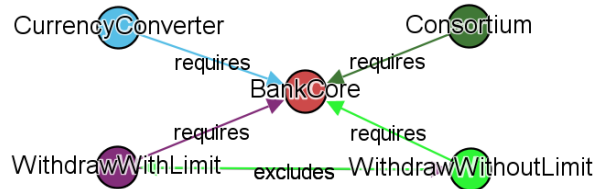


Figure 5.7: Structural constraints discovered among the features of the banking system. All the features require the bank core and there is a mutual exclusion between withdraw with limit and withdraw without limit.

Mutual exclusion constraints discovery: Cardinality in model references plays an important role in defining a DSL. These cardinalities define constraints in the domain that must not be violated to obtain valid models. To avoid violation of upper bound cardinalities we identify in which cases two features cannot coexist in the same model.

To illustrate upper bound cardinalities in real modeling scenarios, and following with the context of the running example, we discuss the cardinalities of the widely used UML meta-model. In UML meta-model, as implemented in Eclipse UML2 ecore, there are 242 classes with a total of 3113 non-volatile, non-transient, non-derived references. 67.7% of these references have no upper bound. 32.2% of them have an upper bound of one maximum referenced model classes. The remaining 0.1% corresponds to the **DurationObservation** meta-class which allows to model execution durations in UML. This class has the **event** reference with an upper bound of two UML **NamedElements**.

Taking the example of a containment reference with an upper bound of one, it is structurally invalid to reference two different containments at the same time. Figure 5.8 illustrates an example of two UML variants from which three features are identified: a **Core**, **Time1** and **Time2** features. We show how a structurally invalid model can be created by combining **Time1** and **Time2** features because they will violate constraints defined in the UML meta-model. We show an excerpt of the meta-model at the bottom of the figure: the **TimeEvent** class has the **when** containment reference with an upper bound of one to a **TimeExpression** class. Therefore, it is not valid to reference two **TimeExpression** classes using this reference. Given such information we can identify *mutual exclusion* constraints because having **Time1** and **Time2** will violate the upper bound. The specific formula to identify structural mutual constraints is presented in Section 4.3.5.

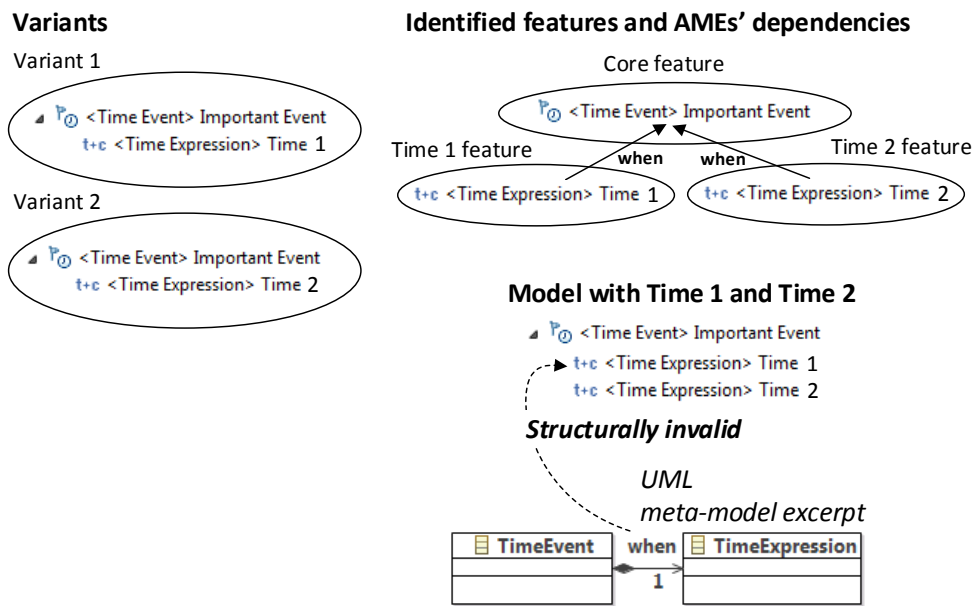


Figure 5.8: Illustrative example of structurally invalid model as result of composing two features.

This is applicable also for the reference AME and for the attribute AME. In our running example, we have two attribute AMEs that depend on the same **Operation** class AME with the **name** dependency id. In one of them the value is `withdrawWithLimit` and in the other is `withdrawWithoutLimit`. These attribute AMEs correspond to two different features and only one can be used in a derived model. This discovered constraint is shown in Figure 5.7 with an “excludes” link between the two features.

5.4.2 ArgoUML case study

ArgoUML is an open source tool for UML modeling. Variants of this tool were created from its Java codebase [CVF11b]. The features are mainly related to the tool support for the edition of different kind of UML diagrams (i.e., **Activity**, **Collaboration**, **Deployment**, **Sequence**, **State** and **UseCase** diagrams). We reverse-engineered the source-code of seven variants related to diagram edition support as UML models in order to apply MoVa2PL.

Table 5.1 presents these variants and the AMEs obtained after using the model adapter. The last column corresponds to the number of dependencies between AMEs. For example, the first one, **ActivityDisabled**, meaning that this model variant contains all the features related to UML diagrams except **Activity** diagram, contains more than hundred thousand AMEs. Reverse-engineering source code variants to models was also appropriate to evaluate the scalability of MoVaPL. These models contain more than fifty thousand classes.

Table 5.1: Number of Atomic Model Elements of ArgoUML UML model variants and number of dependencies between them.

Variant	AMEs	Class	Attr	Ref	Depend
ActivityDisabled	157,896	51,235	77,707	28,954	180,623
CollabDisabled	158,535	51,418	78,046	29,071	181,338
DeployDisabled	157,314	51,033	77,450	28,831	179,949
Original	159,771	51,820	78,667	29,284	182,738
SequenceDisabled	155,231	50,349	76,417	28,465	177,646
StateDisabled	156,193	50,699	76,805	28,689	178,785
UsecaseDisabled	157,504	51,056	77,547	28,901	180,184

The decomposition in AMEs for the seven model variants, including the dependencies, took an average of 15 seconds (i.e., around two seconds per variant) using a lap-top Dell Latitude E6330 with a processor Inter(R) Core(TM) i73540M CPU @3.00GHz 3.00GHz, 8GB RAM, with Windows 7 and 64-bit Operating System. The interdependent elements algorithm identified 41 blocks and took an average of seven minutes in ten runs. Table 5.2 shows the result of the identified blocks and their size in terms of AMEs.

Table 5.2: Number of Atomic Model Elements of the blocks identified in the ArgoUML case study.

Block	AMEs	Class	Attr	Ref
Block 0 -Core	143,894	46,724	70,696	26,474
Block 1 -UseCase	2,260	760	1,117	383
Block 2 -Sequence	4,509	1,461	2,233	815
Block 3 -Collaboration	1,204	392	604	208
Block 4 -State	3,499	1,095	1,818	586
Block 5 -Deployment	2,457	787	1,217	453
Block 6 -Activity	1,796	559	916	321
Block 7	1	0	0	1
Block 8	1	0	0	1
⋮	⋮	⋮	⋮	⋮
Block 40	4	2	2	0

We manually analysed the blocks in order to identify the features. The meaning of the blocks from zero to six was easily recognisable by using the visualisation presented in Chapter 8. Block 0 corresponds to the **Core** of ArgoUML, Block 1 to **UseCase** diagrams edition, Block 2 to **Sequence**, Block 3 to **Collaboration**, Block 4 to **State**, Block 5 to **Deployment** and

Block 6 to **Activity** diagrams. These blocks were the bigger ones in terms of number of AMEs given that the rest of the blocks only contain few of them as shown in the table (e.g., Block 7 only contains one reference AME).

Apart from the “big” blocks, we manually checked the small blocks and we realized that most of them contain reference AMEs that, in the UML meta-model, are defined as ordered (i.e., the order of the referenced elements is important). The applied matching method takes into consideration the ordering of the referenced elements and therefore considered them as different. This issue, probably introduced by the Java to UML reverse engineering tool, makes more difficult the work of the domain expert who needs to manipulate and analyse these blocks. However, in terms of size of model elements, the main part of the features were successfully identified by MoVa2PL (i.e., blocks zero to six).

Figure 5.9 shows the graph visualisation of the discovered structural constraints. Concretely, for the ArgoUML case study, 45 requires constraints and 13 mutual exclusion constraints were discovered. Figure 5.9 shows that the six identified features related to the diagrams requires the **Core** feature. The other nodes of the graph correspond to blocks that the domain expert needs to analyse. These automatically calculated relations among the blocks can help during this manual process. For example, those blocks that exclude each other should be related and the scope of analysis is narrowed for those blocks requiring another block that is not the **Core** feature.

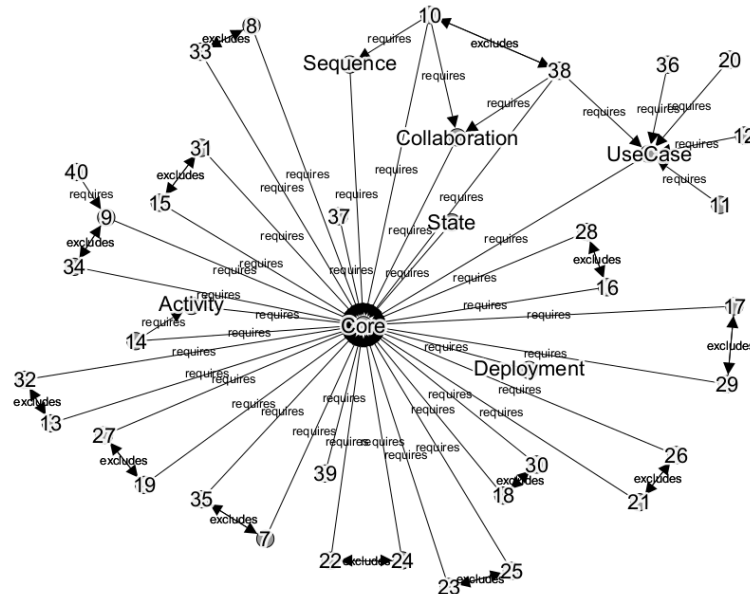


Figure 5.9: Structural constraints discovered for the ArgoUML case study.

Once the constraints are discovered, the CVL MSPL extraction step generates the variability model, the base model and the realization layer. Figure 5.10 shows an excerpt of the realization layer related to the **UseCase** and **Sequence** features. We can see the placement fragment with its related classes. The base model was created as the maximum model from the model variants and the time for its creation took an average of eight minutes while the variability and realization layer took three seconds.

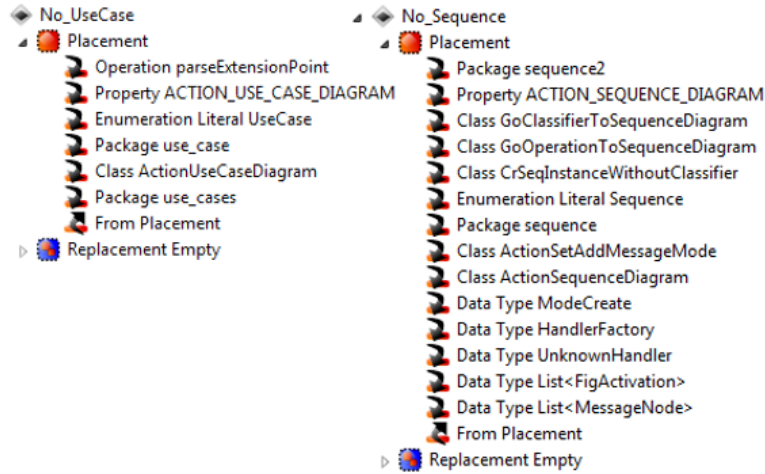


Figure 5.10: Excerpt of the CVL Realization layer for the ArgoUML case study.

In the beginning we had model variants that only considered disabling one type of diagram each. With the extracted MSPL, we can generate ArgoUML UML model variants with any combination of diagrams. For example, we can derive a UML model considering only Sequence diagrams.

The mentioned problem identified with the ordering of the references highlights the importance of the matching method. MoVa2PL is flexible to apply different matching methods as we presented in Section 5.3.3. By providing a matching method that ignores the ordering of the references, 18 blocks were identified in the ArgoUML case study from which the first seven blocks also corresponded to the Core and diagrams' features.

5.4.3 In-Flight Entertainment Systems case study

We consider the case study of an In-Flight Entertainment (IFE) System. This system is responsible for providing entertainment services for the passengers, including movies, music, internet connection or games during a flight. For our experiments we consider an IFE system from the Thales Group modelled in Capella [Pol15]. Capella is a system engineering modeling tool which implements the Arcadia method for system, software and hardware architectural design. A system modelled with Capella consists of five layers. The operational analysis layer captures the stakeholders, their needs, as well as general information of the system's domain. The system analysis layer formalizes the system requirements. The logical architecture layer defines how the system fulfils its requirements. The physical architecture layer defines how the system will be technically developed and built. Finally, the end-product breakdown structure layer formalizes the component requirements definition to facilitate component integration, validation, verification and qualification. The domain of Arcadia method is realized and tool-supported in Capella by defining a system engineering DSL consisting of 17 related meta-models with a total of 411 meta-classes.

We consider in our study three model variants of the IFE system. The first is the **Original** IFE system and the other two (**LowCost1** and **LowCost2**) are manually created taking this

variant as input. **LowCost1** is a variant that does not include the feature for **Wi-Fi** access for passengers. Figure 5.11 shows an operational analysis diagram with model elements related to Wi-Fi access. We can see how the aircraft provides connectivity and the personal device allows the passenger to browse the Internet during the flight. The capability of Wi-Fi access for passengers is propagated to the rest of the model layers such as the system analysis or the logical and physical architecture. **LowCost2** is an IFE system without support for **ExteriorVideo** which allows the passengers to watch, in their personal screens, the exterior of the plane at any time during the flight.

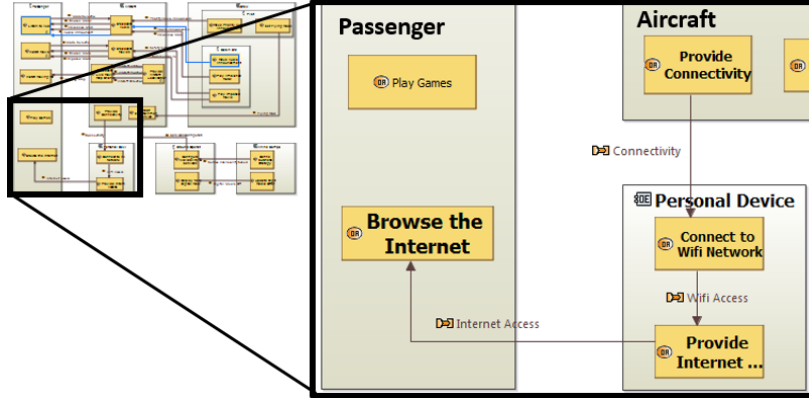


Figure 5.11: Operational analysis diagram of the In-Flight Entertainment system variant showing some model elements related to Wi-Fi access for passengers.

Following the MoVa2PL process, first the IFE model variants are decomposed in AMEs and the dependencies among AMEs are calculated. Figure 5.12 shows the decomposition of the **Original** IFE System variant to highlight the size of the artefact and the density of their dependencies. Table 5.3 presents the obtained number of AMEs for each variant.

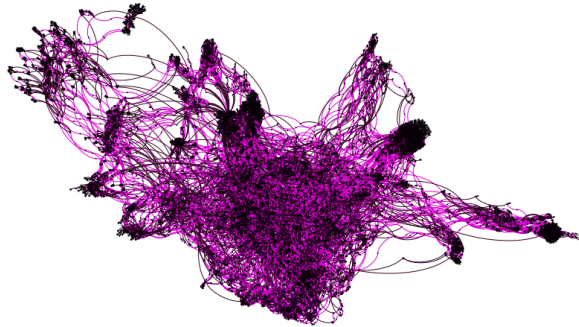


Figure 5.12: In-Flight Entertainment system variant decomposition in Atomic Model Elements.

Table 5.3: Number of Atomic Model Elements of IFE system model variants and number of dependencies between them.

Variant	AMEs	Class	Attr	Ref	Depend
Original	16,624	5,345	4,081	7,198	24,205
LowCost1	16,551	5,321	4,066	7,164	24,098
LowCost2	16,594	5,335	4,075	7,184	24,161

The interdependent AMEs algorithm leads to the identification of three blocks. The identified blocks are manually analysed and mapped to the features. Table 5.4 summarizes the number of AMEs for the corresponding features.

Table 5.4: Number of Atomic Model Elements of the features identified in the In-Flight Entertainment model variants.

Block	AMEs	Class	Attr	Ref
Block0 -Core	16,521	5,311	4,060	7,150
Block1 -Wi-Fi	73	24	15	34
Block2 -ExteriorVideo	30	10	6	14

Then, MoVa2PL automatically discovers the structural constraints among the different features. Thus, we discover that both **Wi-Fi** and **ExteriorVideo** requires the **Core** feature. With the gathered information, the CVL models are automatically constructed obtaining an MSPL for the IFE models. From this MSPL, we are able to generate a new variant that does not contain **Wi-Fi** nor **ExteriorVideo**.

5.4.4 Discussions about MoVa2PL

For evaluating MoVa2PL, based on the case studies outlined above, we concentrate on checking that, in each case, the extracted MSPL is able not only to 1) re-generate the previously existing variants using our systematic reuse approach, but also to 2) generate previously non-existing variants which are structurally valid. By realizing the experiments on different modeling meta-models, namely UML models and the Capella DSL, we have shown the flexibility and genericity of MoVa2PL. We obtained an exact match of the re-generated models and we generated possible non-existing variants (RQ1 and RQ2). We also checked manually the structural validity and we found that we successfully prevented invalid models (RQ1).

Currently, MoVa2PL presents a number of limitations and includes hypotheses which are threats to validity. These limitations correspond also to the limitations of BUT4Reuse itself as we presented in Section 4.5. Feature identification is a challenging task and can be complex for domain experts. Automatically identifying features using heuristics may lead to an output where a given identified feature is actually a set of different ones. This situation is likely when a set of features came always together in all the variants. In this case, the interdependent elements technique for block identification cannot distinguish among them. The coordination elements of possible feature interactions can also complicate the process.

Regarding feature constraints, MoVa2PL presents some limitation in ensuring the structural validity of models. Indeed, we are not considering constraints that could be defined using the Object Constraints Language (OCL) [OMG14, WK03, CP06]. There is also the issue of the semantic validity of derived model variants. A semantically valid model is a structurally valid model which also makes sense in the domain (i.e., a combination of features that does not violate any semantic rule of the domain). The extracted MSPL will allow the creation of new

models based on combinations of identified features. However, these new models might be semantically invalid. The challenging issue of assuring some notion of semantic validity has been addressed in other works such as Czarnecki *et al.* [CP06]. In Chapter 9, we propose a visualisation paradigm to help domain experts formalize these constraints using their domain knowledge.

5.5 Conclusion

The definition of the model adapter for BUT4Reuse enables MoVa2PL to chain extractive SPL adoption activities. We have presented MoVa2PL as a solution to MSPL adoption from existing model variants. Firstly, the feature identification process considers structural constraints discovery in order to extract a variability model that ensures the validity of the MSPL configuration space. Secondly, a product realization layer is extracted with the information of the AMEs related to each of the features. This realization layer will operate on a base model extracted by merging the variants in a base model. We assessed MoVa2PL in two case studies considering big-medium sized model variants with different meta-models.

Despite that MoVa2PL is extensible for the similarity calculation, it is interesting to continue researching on facilitating the definition of signature-based matching approaches [KDRPP09] by domain experts, or the evaluation of semantic techniques in other case studies. Also, as presented in Section 5.2, apart from descriptive models, models are used in a prescriptive way and therefore the artefacts associated to them (e.g., model transformations [CFSS16] or behavior semantics [MGC⁺16]) should be taken into account during extractive MSPL adoptions.

Part III

COLLECTING ARTEFACT VARIANTS
FOR STUDY AND BENCHMARKING

6

BENCHMARK FOR FEATURE LOCATION TECHNIQUES USING ECLIPSE VARIANTS

This chapter is based on the work that has been published in the following paper extended with the automatic generation of variants:

- Jabier Martinez, Tewfik Ziadi, Mike Papadakis, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Feature location benchmark for software families using Eclipse community releases. In *ICSR*, volume 9679 of *Lecture Notes in Computer Science*, pages 267–283. Springer, 2016

Contents

6.1	Introduction	86
6.2	The Eclipse family of integrated development environments	87
6.2.1	Tailored Eclipses for different development needs	87
6.2.2	Reasons to consider Eclipse for benchmarking	90
6.3	EFLBench: Eclipse Feature Location Benchmarking framework	91
6.3.1	Benchmark construction	91
6.3.2	Benchmark usage	92
6.4	Examples of EFLBench usage in Eclipse releases	93
6.5	Automatic and parametrizable generator of Eclipse variants	95
6.5.1	Strategies for the automatic selection of configurations	96
6.5.2	Results using automatic generation of variants	98
6.6	Conclusions	100

6.1 Introduction

As presented in Section 2.2.2, feature location is an essential activity of extractive processes towards systematic reuse. In Section 3.1.3, we discussed the diversity of proposed techniques and the increasing interest by the research community on this subject [RC13b, AV14]. Because of this, feature location benchmarks are required to push the state-of-the-art enabling intensive experimentation of the techniques. Concretely, there is a need to empirically evaluate and compare the strengths and weakness of the techniques in different scenarios.

Comparing and experimenting with feature location techniques is challenging

- *Most of the tools are strongly dependent on specific artefact types* that they were designed for (e.g., a given type of model or programming language).
- *Performance comparison requires common settings and environments.* There exist difficulties to reproduce the experimental settings to compare performance.
- *Most of the research prototypes are either unavailable or hard to configure.* There exists a lack of accessibility to the tools implementing each technique with its variants abstraction and feature location phases.

Given that common case study subjects and frameworks are in need to foster the research activity [SEH03], we identified two requirements for such frameworks in feature location:

- *A standard case study subject:* Subjects that are non-trivial and easy to use are needed. This includes: 1) A list of existing features, 2) for each feature, a group of elements implementing it and 3) a set of product variants accompanied by the information of the included features.
- *A benchmarking framework:* In addition to the standard subjects, a full implementation allowing a common, quick and intensive evaluation is needed. This includes: 1) An available implementation with a common abstraction for the product variants to be considered by the case studies, 2) easy and extensible mechanisms to integrate feature location techniques to support the experimentation, and 3) predefined evaluation metrics to draw comparable results.

Contributions of this chapter.

- We present the **Eclipse Feature Location Benchmark (EFLBench)** and examples of its usage. We propose a standard case study for feature location and a benchmark framework using Eclipse packages, their features and their associated plugins. We implemented EFLBench within BUT4Reuse which allows a quick integration of feature location techniques.
- We present the **automatic generation of Eclipse variants** as part of EFLBench capabilities to construct tailored benchmarks. This enables the evaluation of techniques in different scenarios to show their strengths and weaknesses.

The rest of the chapter is structured as follows: In Section 6.2 we present Eclipse as a case study subject and in Section 6.3 we present the EFLBench framework. Section 6.4 presents different feature location techniques and the results of EFLBench usage in the official Eclipse releases. Section 6.5 presents the strategies for automatic generation of Eclipse variants. Finally, Section 6.6 concludes and presents future work.

6.2 The Eclipse family of integrated development environments

This section extends the information presented in Section 4.4 about the domain of the Eclipse Integrated Development Environment (IDE) [Ecl16a]. Then we justify the creation of a benchmarking framework using this case study.

6.2.1 Tailored Eclipses for different development needs

The Eclipse community, with the support of the Eclipse Foundation, provides integrated development environments (IDEs) targeting different developer profiles. The IDEs cover the development needs of *Java*, *C/C++*, *JavaEE*, *Scout*, *Domain-Specific Languages*, *Modeling*, *Rich Client Platforms*, *Remote Applications Platforms*, *Testing*, *Reporting*, *Parallel Applications* or for *Mobile Applications*. Following Eclipse terminology, each of the customized Eclipse IDEs is called an Eclipse **package**.

As the Eclipse project evolves over time, new packages appear and some other ones disappear depending on the interest and needs of the community. For instance, in 2012, one package for *Automotive Software* developers appeared and, recently, in 2016, another package appeared for *Android mobile applications development*. The Eclipse Packaging Project (EPP) is the technical responsible for creating entry level downloads based on defined user profiles.

Continuing with Eclipse terminology, a *simultaneous release* (**release** hereafter) is a set of packages which are public under the supervision of the Eclipse Foundation. Every year, there is one main release, in June, which is followed by two service releases for maintenance purposes: SR1 and SR2 usually around September and February. For each release, the platform version changes and traditionally celestial bodies are used to name the releases, for example Luna for version 4.4 and Mars for version 4.5.

The packages present variation depending on the included and not-included **features**. For example, Eclipse package for Testers is the only one including the Jubula Functional Testing features. On the contrary, other features like the Java Development tools are shared by most of the packages. There are also common features for all the packages, like the Equinox features that implement the core functionality of the Eclipse architecture. The online documentation of each release provides high-level information on the features that each package provides ⁱ.

ⁱHigh-level comparison of Eclipse packages of the latest release:
<https://eclipse.org/downloads/compare.php>

It is important to mention that in this work we are not interested in the variation among the releases (e.g., version 4.4 and 4.5, or version 4.4 SR1 and 4.4 SR2), known as *variation in time*. We focus on the variation of the different packages of a given release, known as *variation in space*, which is expressed in terms of included and not-included features. Each package is different in order to support the needs of the targeted developer profile by including only the appropriate features.

Eclipse is feature-oriented and based on **plugins**. Each feature consists of a set of plugins that are the actual implementation of the feature. Table 6.1 shows an example of feature with four plugins as implementation elements that, if included in an Eclipse package, adds support for the Concurrent Versioning System (CVS). At technical level, the actual features of a package can be found within a folder called *features* containing meta-information regarding the included features and the list of plugins associated to each. A feature has an id, a name and a description as defined by the feature providers of the Eclipse community. A plugin has an id and a name defined by the plugin providers, but it does not have a description.

Table 6.1: Eclipse feature example. The Eclipse CVS Client feature and its associated plugins.

Feature	
<i>id:</i> org.eclipse.cvs	
<i>name:</i> Eclipse CVS Client	
<i>description:</i> Eclipse CVS Client (binary runtime and user documentation).	
Plugin id	Plugin name
org.eclipse.cvs	Eclipse CVS Client
org.eclipse.team.cvs.core	CVS Team Provider Core
org.eclipse.team.cvs.ssh2	CVS SSH2
org.eclipse.team.cvs.ui	CVS Team Provider UI

Table 6.2 presents data regarding the evolution of the Eclipse releases over the years. In particular, it presents the total number of packages, features and plugins per release. To illustrate the distribution of packages and features, Figure 6.1 depicts a matrix of the different Eclipse Kepler SR2 packages where a black box denotes the presence of a feature (horizontal axis) in a package (vertical axis). We observe that some features are present in all the packages while others are specific to only few packages. The 437 features are alphabetically ordered by their id. For instance, the feature *Eclipse CVS Client*, tagged in the figure, is present in all packages except in the *Automotive Software* package.

Table 6.2: Eclipse releases and their number of packages, features and plugins.

Year	Release	Packages	Features	Plugins
2008	Europa Winter	4	91	484
2009	Ganymede SR2	7	291	1,290
2010	Galileo SR2	10	341	1,658
2011	Helios SR2	12	320	1,508
2012	Indigo SR2	12	347	1,725
2013	Juno SR2	13	406	2,008
2014	Kepler SR2	12	437	2,043
2015	Luna SR2	13	533	2,377

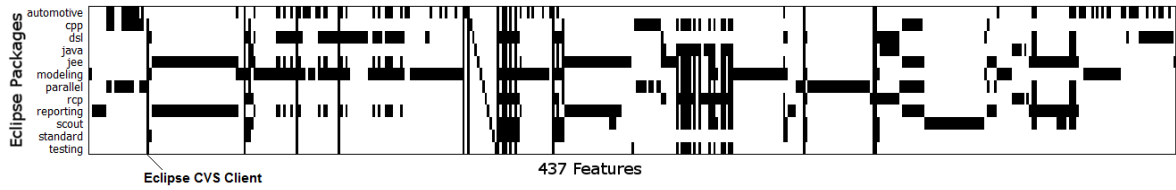


Figure 6.1: Eclipse Kepler SR2 packages and a mapping to their 437 features. For example, Eclipse CVS Client is present in all packages except in the automotive package.

Features have dependencies among them: *Includes* is the Eclipse terminology to define subfeatures, and *Requires* means that there is a functional dependency between the features. Figure 6.2 shows the dependencies between all the features of all packages in Eclipse Kepler SR2. We tagged some features and subfeatures of the Eclipse Modeling Framework to show cases of features that are strongly related. Functional dependencies are mainly motivated by the existence of dependencies between plugins of different features. In the Eclipse IDE family there is no *excludes* constraint between the features. Regarding plugin dependencies, they are explicitly declared in each plugin meta-data. Figure 6.3 shows a small excerpt of the dependency connections of the 2043 plugins of Eclipse Kepler SR2. Concretely, the excerpt shows the dependencies of the four CVS plugins presented in Table 6.1.

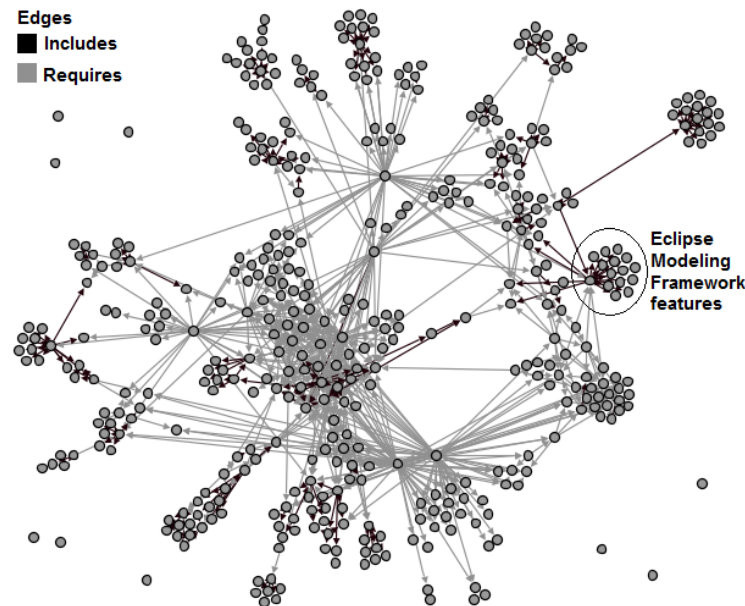


Figure 6.2: Feature dependencies in the Eclipse Kepler SR2 packages.

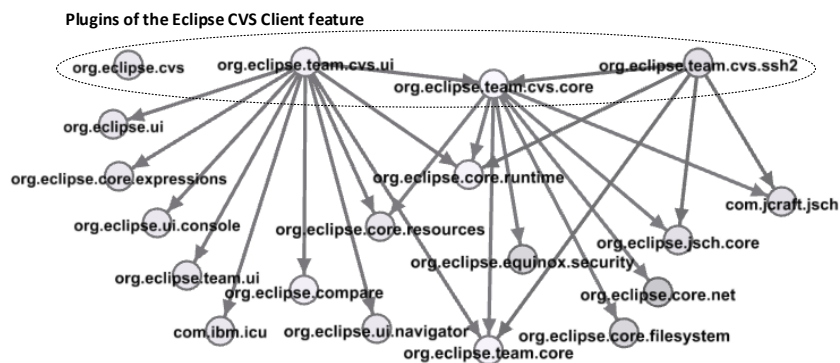


Figure 6.3: Plugin dependencies of the four plugins of the Eclipse CVS Client feature.

6.2.2 Reasons to consider Eclipse for benchmarking

We present characteristics of Eclipse packages that make the case study interesting for a feature location benchmark:

Ground truth available: The Eclipse case study fulfils the requirement, mentioned in Section 6.1, of providing the needed data to be used as ground truth. This ground truth can be extracted from features meta-information. Despite that the granularity of the implementation elements (plugins) is coarse if we compare it with source code AST nodes, the number of plugins is still reasonably high. In Eclipse Kepler SR2, the total amount of unique plugins is 2043 with an average of 609 plugins per Eclipse package and a standard deviation of 192.

Challenging: The relation between the number of available packages in the different Eclipse releases (around 12) and the number of different features (more than 500 in the latest release) is not balanced. This makes the Eclipse case study challenging for techniques based only in static comparison (e.g., interdependent elements or FCA) because they will probably identify few “big” blocks containing implementation elements belonging to a lot of features. The number of available product variants has been shown to be an important factor for feature location techniques [FLLE14].

Friendly for information retrieval and dependency analysis: Eclipse feature and plugin providers have created their own natural language vocabulary. The feature and plugin names (and the description in the case of the features) can be categorized as meaningful names [RC13b] enabling the use of several IR techniques. Also, the dependencies between features and dependencies between implementation elements have been used in feature location techniques. For example, in source code, program dependence analysis has been used by exploiting program dependence graphs [CR10]. Acher *et al.* also leveraged architecture and plugin dependencies [ACC⁺14]. As presented in previous section, Eclipse also has dependencies between features and dependencies between plugins enabling their exploitation during feature location.

Noisy: There are properties that can be considered as “noise” that are common in real scenarios. Some of them can be considered as non-conformities in feature specification [SFdOA13]. A case study without “noise” should be considered as an optimistic case study. In Eclipse Kepler SR2, 8 plugins do not have a name, and different plugins from the same feature are named exactly the same. There are also 177 plugins associated to more than one feature. Thereby the features’ plugin sets are not completely disjoint. These plugins are mostly related to libraries for common functionalities which were not included as required plugins but as a part of the feature itself. In addition, 40 plugins present in some of the variants are not declared in any feature. Also, in few cases, feature versions are different among packages of the same release.

Friendly for customizable benchmark generation: The fact that Eclipse releases contain few packages can be seen as a limitation for benchmarking in other desired scenarios with larger amount of variants. For example, it will be desired to show the relation between

the results of the technique and the number of considered variants. Apart from the official releases, software engineering practitioners have created their own Eclipse packages. Therefore, researchers can use their own packages or create variants with specific characteristics. In addition, the plugin-based architecture of Eclipse allows to implement automatic generators of Eclipse variants as we present later in Section 6.5.

Similar experiences exist: Analysing plugin-based or component-based software system families to leverage their variability has been shown in previous works [ACC⁺14, GRDL09, SSS16]. For instance, experiences in an industrial case study were reported by Grünbacher *et al.* where they performed manual feature location in Eclipse packages to extract an SPL involving more than 20 package customizations per year [GRDL09].

6.3 EFLBench: Eclipse Feature Location Benchmarking framework

EFLBench is aimed to be used with any set of Eclipse packages including packages with features that are not part of any official release. Figure 6.4 illustrates, at the top, the phase for constructing the benchmark and, at the bottom part, the phase for using it. The following subsections provide more details on the two phases.

6.3.1 Benchmark construction

The benchmark construction phase takes as input the Eclipse packages and automatically produce two outputs, 1) a Feature list with information about each feature name, description and the list of packages where it was present, and 2) a ground truth with the mapping between the features and the implementation elements which are the plugins.

We implemented an automatic extractor of features information. The implementation elements of a feature are those plugins that are directly associated to this feature. From the 437 features

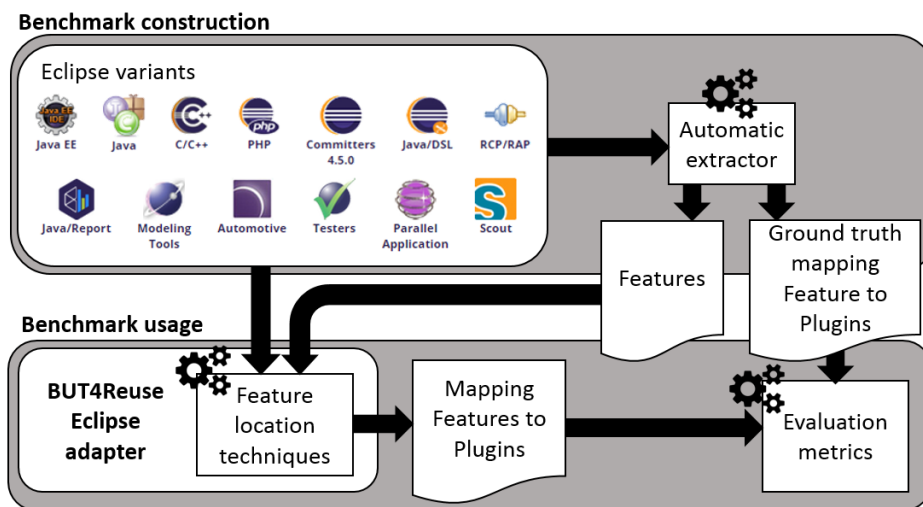


Figure 6.4: EFLBench: Eclipse package variants as benchmark for feature location.

of the Eclipse Kepler SR2, each one has an average of 5.23 plugins associated with, and a standard deviation of 9.67 plugins. There is one outlier with 119 plugins which is the feature *BIRT Framework* included in the Reporting package. From the 437 features, there are 19 features that do not contain any plugin, so they are considered *abstract* features which are created just for grouping other features. For example, the abstract feature *UML2 Extender SDK* (Software Development Kit) includes the features *UML2 End User Features*, *Source for UML2 End User Features*, *UML2 Documentation* and *UML2 Examples*.

Reproducibility is becoming quite easy by using benchmarks and common frameworks that launch and compare different techniques [SEH03]. This practice, allows a valid performance comparison with all the implemented and future techniques. We integrated EFLBench and its automatic extractor in BUT4Reuse.

6.3.2 Benchmark usage

Once the benchmark is constructed, at the bottom of Figure 6.4 we illustrate how it can be used through BUT4Reuse where feature location techniques can be integrated as presented in Section 4.3.4. The Eclipse adapter, detailed in Section 4.4.1, is responsible for the variant abstraction phase. This will be followed by the launch of the targeted feature location techniques which takes as input the feature list and the Eclipse packages (excluding the *features* folder). The feature location technique produces a mapping between features and plugins that can be evaluated against the ground truth obtained in the benchmark construction phase. Concretely, EFLBench calculates the *precision* and *recall* which are classical evaluation metrics in IR studies (e.g., [SSD14]).

We explain precision and recall, two metrics that complement each other, in the context of EFLBench. A feature location technique assigns a set of plugins to each feature. In this set, there can be some plugins that are actually correct according to the ground truth. Those are *true positives* (TP). TPs are also referred to as *hit*. On the set of plugins retrieved by the feature location technique for each feature, there can be other plugins which do not belong to the feature. Those are *false positives* (FP) which are also referred to as *false alarms*. Precision is the percentage of correctly retrieved plugins relative to the total of retrieved plugins by the feature location technique. A precision of 100% means that the ground truth of the plugins assigned to a feature and the retrieved set from the feature location technique are the same and no “extra” plugins were included. The formula of precision is shown in Equation 6.1.

$$precision = \frac{TP}{TP + FP} = \frac{plugins\ hit}{plugins\ hit + plugins\ false\ alarm} \quad (6.1)$$

According to the ground truth there can be some plugins that are not included in the retrieved set, meaning that they are *miss*. Those plugins are *false negatives* (FN). Recall is the percentage of correctly retrieved plugins from the set of the ground truth. A recall of 100%

means that all the plugins of the ground truth were assigned to the feature. The formula of recall is shown in Equation 6.2.

$$recall = \frac{TP}{TP + FN} = \frac{plugins\ hit}{plugins\ hit + plugins\ miss} \quad (6.2)$$

Precision and recall are calculated for each feature. In order to have a global result of the precision and recall we use the mean of all the features. Finally, BUT4Reuse reports the *time* spent for the feature location technique. With this information, the time performance of different techniques can be compared.

6.4 Examples of EFLBench usage in Eclipse releases

This section aims at presenting the possibilities of EFLBench by benchmarking four feature location techniques in official Eclipse releases. For the four techniques we use Formal Concept Analysis (FCA) as a first step for block identification. FCA is presented in Section 4.3.2. Concretely, the four feature location techniques are SFS, SFS+ST, SFS+TF, SFS+TFIDF which were detailed in Section 4.3.4.

In SFS+ST, SFS+TF, SFS+TFIDF, where we use IR and Natural Language Processing (NLP), we do not make use of the feature or plugin ids. In order to extract the meaningful words from both features (name and description) and elements (plugin names), we used two well established techniques in the IR field. We discuss them here with examples regarding the Eclipse case study:

- **Parts-of-speech tags remover:** These techniques analyse and tag words depending on their role in the text. The objective is to filter and keep only the potentially relevant words. For example, conjunctions (e.g., “and”), articles (e.g., “the”) or prepositions (e.g., “in”) are frequent and may not add relevant information. As an example, we consider the following feature name and description: “*Eclipse Scout Project. Eclipse Scout is a business application framework ~~that~~ supports desktop, web ~~and~~ mobile frontends. ~~This~~ feature contains ~~the~~ Scout core runtime components.*”. We apply Part-of-Speech Tagger techniques using OpenNLP [Apa10].
- **Stemming:** This technique reduces the words to their root. The objective is to unify words not to consider them as unrelated. For instance, “playing” will be considered as stemming from “play” and “tools” from “tool”. Instead of keeping the root, we keep the word with greater number of occurrences to replace the involved words. As example, in the Graphiti feature name and description we find “[...] *Graphiti supports the fast and easy creation of unified **graphical** tools, which can **graphically** display[...]*” so graphical and graphically is considered the same word as their shared stem is *graphic*. Regarding the implementation, we used the Snowball steamer [Por01].

Given that `tf-idf` is used in SFS+TFIDF, we illustrate it in the context of Eclipse features. For example “Core”, “Client” or “Documentation” are more frequent words across features but “CVS” or “BIRT”, being less frequent, are probably more relevant, informative or discriminating.

We used the benchmark created with each of the Eclipse releases presented in Table 6.2. The experiments were launched using BUT4Reuse at commit `ce3a002` (19 December 2015) which contains the presented feature location techniques. Detailed instructions for reproducibility are available ⁱⁱ. We used a laptop Dell Latitude E6330 with a processor Intel(R) Core(TM) i7-3540M CPU@3.00GHz with 8GB RAM and Windows 7 64-bit.

After using the benchmark, we obtained the results shown in Table 6.3. *Precision* and *Recall* are the mean of all the features as discussed at the end of Section 6.3.2. The results in terms of precision are not satisfactory in the presented feature location techniques. This suggests that the case study is challenging. Also, we noticed that there are no relevant differences in the results of these techniques among the different Eclipse releases. As discussed before, given the few amount of Eclipse packages under consideration, FCA is able to distinguish blocks which may actually correspond to a high number of features. For example, all the plugins corresponding specifically to the Eclipse Modeling package, will be grouped in one block while many features are involved.

Another example, in Eclipse Kepler SR2, FCA-based block identification identifies 60 blocks with an average of 34 plugins per block and a standard deviation of 54 plugins. In Eclipse Europa Winter, with only 4 packages, only 6 blocks are identified with an average of 80 plugins each and a standard deviation of 81. Given the low number of Eclipse packages, FCA identifies a low number of blocks. The number of blocks is specially low if we compare it with the actual number of features that we aim to locate (e.g., 60 blocks in Kepler SR2 against its 437 features). The higher the number of Eclipse packages, the most likely FCA will be able to distinguish different blocks.

Table 6.3: Precision and recall of the different feature location techniques.

Release	SFS		SFS+ST		SFS+TF		SFS+TFIDF	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
Europa Winter	6.51	99.33	11.11	85.71	12.43	58.69	13.07	53.72
Ganymede SR2	5.13	97.33	10.36	87.72	11.65	64.31	12.80	52.70
Galileo SR2	7.13	93.39	10.92	82.01	11.82	60.50	12.45	53.51
Helios SR2	9.70	91.63	16.04	80.98	25.97	63.70	29.46	58.39
Indigo SR2	9.58	92.80	15.72	82.63	19.79	59.72	22.86	57.57
Juno SR2	10.83	91.41	19.08	81.75	25.97	61.92	24.89	60.82
Kepler SR2	9.53	91.14	16.51	83.82	26.38	62.66	26.86	57.15
Luna SR2	7.72	89.82	13.87	82.72	22.72	56.67	23.73	51.31
<i>Mean</i>	<i>8.26</i>	<i>93.35</i>	<i>14.20</i>	<i>83.41</i>	<i>19.59</i>	<i>61.02</i>	<i>20.76</i>	<i>55.64</i>

ⁱⁱ<https://github.com/but4reuse/but4reuse/wiki/Benchmarks>

The first location technique (FCA+SFS) does not assume meaningful names given that no IR technique is used. The features are located in the elements of a whole block obtaining a high recall (few plugins missing). Eclipse feature names and descriptions are probably written by the same community of developers that create the plugins and decide their names. In the approaches using IR techniques, the authors expected a higher increment of precision without a loss of recall but the results suggest that certain divergence exists between the vocabulary used at feature level and at implementation level.

Regarding the time performance, Table 6.4 shows, in milliseconds, the time spent for the different releases. The *Adapt* column corresponds to the time to decompose the Eclipse packages into a set of plugin elements and get their information. This adaptation step heavily rely to access the file system and we obtain better time results after the second adaptation of the same Eclipse package. The FCA time corresponds to the time for block identification. We consider Adapt and FCA as the preparation time. Then, the following columns show the time of the different feature location techniques. We can observe that the time performance is not a limitation of these techniques as they take a maximum of around half a minute.

Table 6.4: Time performance in milliseconds for feature location.

Release	<i>Preparation</i>		<i>Concrete techniques</i>			
	<i>Adapt</i>	<i>FCA</i>	<i>SFS</i>	<i>SFS+ST</i>	<i>SFS+TF</i>	<i>SFS+TFIDF</i>
Europa Winter	2,397	75	6	2,581	2,587	4,363
Ganymede SR2	7,568	741	56	11,861	11,657	23,253
Galileo SR2	10,832	1,328	107	17,990	17,726	35,236
Helios SR2	11,844	1,258	86	5,654	5,673	12,742
Indigo SR2	12,942	1,684	100	8,782	8,397	16,753
Juno SR2	16,775	2,757	197	7,365	7,496	14,002
Kepler SR2	16,786	2,793	173	8,586	8,776	16,073
Luna SR2	17,841	3,908	233	15,238	15,363	33,518
<i>Mean</i>	<i>12,123</i>	<i>1,818</i>	<i>120</i>	<i>9,757</i>	<i>9,709</i>	<i>19,493</i>

It is out of the scope of the EFLBench contribution to propose feature location techniques that could obtain better results in the presented cases. The objective is to present the benchmark usage showing that quick feedback from feature location techniques can be obtained in the Eclipse releases case studies. In addition, we provide empirical results of four feature location techniques that can be used as baseline.

6.5 Automatic and parametrizable generator of Eclipse variants

As shown in Table 6.2, the number of official packages of an Eclipse release amounts to around 12 Eclipse variants. In order to provide a framework for intensive evaluation of feature location techniques, cases with larger number of Eclipse variants are desired. In addition, a parametrizable number of variants could serve to analyse the results of the same feature location technique under different circumstances. We extended the benchmark construction

phase of EFLBench with an automatic and parametrizable generator of Eclipse variants to construct benchmarks with tailored characteristics. The approach consists in automatically creating variants from a user-specified Eclipse package.

Figure 6.5 illustrates the benchmark construction phase using the automatic generation of Eclipse variants. First, as shown on the upper left side of the figure, we take as input an Eclipse package to extract its features and feature constraints. These features and constraints define a configuration space in the sense that, by deselecting features, we can still have valid Eclipse configurations (i.e., all the feature constraints are satisfied). Then, we leverage this configuration space to select a set of configurations. The automatic selection of configurations is parametrized by a given strategy, thus, this step is extensible to different implementations. Shortly below, we present three different strategies that we have implemented. Finally, once the set of configurations are selected, we implemented an automatic method to construct the variants through the input Eclipse and the feature configurations. The constructed variants are created for preparing the benchmark construction but, if desired, given that constraints are respected, they can be executed in the same way as the packages in Eclipse releases.

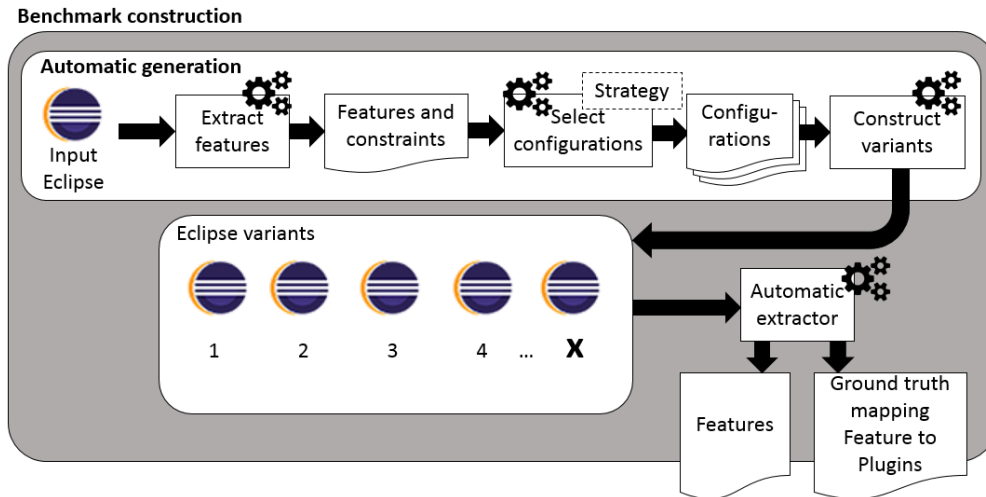


Figure 6.5: Automatic and parametrizable generation of Eclipse package variants to construct a feature location benchmark. The use of different strategies in the step to select configurations enables to construct benchmarks exhibiting different characteristics.

6.5.1 Strategies for the automatic selection of configurations

We implemented three strategies to select configurations from a set of features and constraints with the final objective to construct benchmarks presenting different characteristics. Apart from the input Eclipse, the three take as input a user-specified number of variants that want to be generated. We present the three strategies and then discuss their properties:

- *Random selection strategy:* In this strategy, we randomly select configurations from the configuration space. The selection of random valid configurations, taking as input features and their constraints, is implemented through a functionality offered by the PLEDGE library (Product Line Editor and tests Generation tool) [HPP⁺13b] which internally relies on a SAT solver [BP10].

- *Random selection strategy trying to maximize dissimilarity:* Given the specified number of variants, this strategy aims to obtain a set of configurations that maximize their global dissimilarity (i.e., different among them). For this we use a similarity heuristic between configurations which is supported by PLEDGE relying on the Jaccard distance [HPP⁺14]. First, PLEDGE selects random configurations and then it applies a search-based approach guided by a fitness function that tries to identify the most dissimilar configurations. This strategy demands to select the time allocated to the search-based algorithm. Once the allowed time is over, the set of configurations are obtained.
- *Percentage-based random selection strategy:* This strategy consists of two steps. First, we ignore the constraints and we go through the feature list deciding if we select or not each feature. This is automated by a user-specified percentage defining the chances of the features of being selected. Second, once some features are randomly selected, we need to guarantee that the feature constraints are satisfied. We may have included a feature that requires another one that was not included. Therefore, we *repair* the configuration including the missing features until obtaining a valid configuration.

The first and second strategy can be used to evaluate how a feature location technique behaves with dissimilar variants with high t-wise coverage. Empirical studies of Henard *et al.* showed that dissimilar configurations exhibit interesting properties in terms of t-wise coverage [HPP⁺14]. They also showed that the strategy of selecting random configurations from the configuration space, without the search-based step, already obtained a median of more than 90% of pairwise coverage in 120 FMs of moderate size (i.e., less than one thousand features). The third strategy, compared to the first two, allows to have more control over the total number of selected features per configuration.

Using as input the Modeling package of Eclipse Kepler SR2, Figure 6.6 shows, in the vertical axis, the number of features in 1000 automatically selected configurations using the presented strategies. The total number of features of the input Eclipse package is 173 corresponding

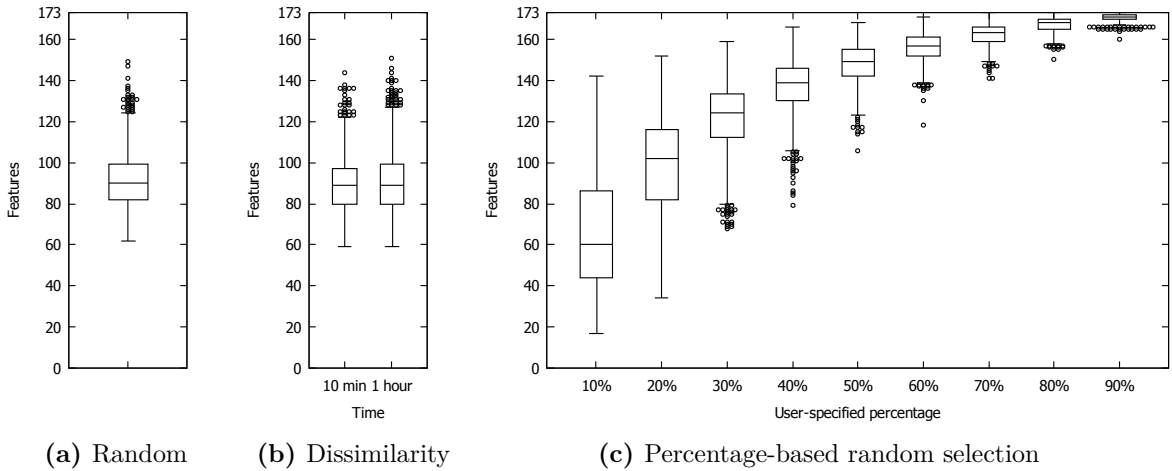


Figure 6.6: Different settings of the three strategies for selecting configurations taking as input the features and constraints extracted from the Modeling package of the Eclipse Kepler SR2. Each boxplot shows the number of features in the selection of 1000 configurations.

to the maximum value. Considering the feature constraints, the configuration space exceeds the million configurations. In the case of the random and dissimilarity strategies, as shown in Figures 6.6a and 6.6b, we can observe that only some outlier configurations reach a large number of selected features. Given that the dissimilarity strategy depends on the number of desired variants to generate, we repeated the process with different number of configurations (not only 1000) obtaining analogous results. We also observed that the time allowed for the search-based algorithm did not affect the number of selected features, at least from 10 minutes to 1 hour as shown in Figure 6.6b. On the contrary, in Figure 6.6c, we can observe how the user-specified percentage has an impact in the median of selected features. Larger percentages allow to obtain configurations with larger number of selected features and, therefore, there will be less chances to obtain dissimilar variants using this strategy compared to the ones using random selection.

6.5.2 Results using automatic generation of variants

We show examples of using the EFLBench strategies for automatic generation of Eclipse variants. We focus on discussing the results of evaluating the FCA+SFS feature location technique. As input for the random generation strategies, we use the Modeling package of Eclipse Kepler SR2 which is the same used to illustrate the strategies for selecting configurations in Figure 6.6.

Using percentage-based random selection of features, we aim to empirically analyse if the number of available variants has an impact on the FCA+SFS technique. First, we generated 100 variants using 40% as percentage for feature selection. By setting this percentage, the first 10 variants cover the 173 features which is the total number of features of the input Eclipse. This allows the construction of different benchmarking settings adding 10 variants each time while keeping the total number of possible features constant.

Table 6.5 shows the precision and recall obtained for FCA+SFS when considering different number of variants. We can observe how precision improves with the number of variants.

Table 6.5: Precision, recall and time measures in milliseconds of the FCA+SFS feature location technique in sets of randomly generated Eclipse packages using the percentage-based random strategy.

Percentage-based random using 40%	FCA+SFS		Time	
	Precision	Recall	FCA	SFS
10 variants	33.40	96.55	122	84
20 variants	47.91	96.02	415	320
30 variants	55.62	95.41	502	630
40 variants	58.60	95.41	1,268	905
50 variants	61.01	93.10	2,168	1,105
60 variants	62.57	90.73	2,455	1,382
70 variants	64.78	90.63	2,636	1,717
80 variants	65.40	90.02	4,137	4,049
90 variants	66.02	89.57	6,957	7,774
100 variants	66.02	89.57	7,515	7,251

From 10 to 20 variants, we have a precision improvement of around 15%. Beyond 30 variants, it seems that the included variants, with their feature combinations, are not adding more information that can be exploited by the FCA+SFS technique. As an extreme case, we can observe how we obtain the same precision with 90 and 100 variants. Regarding recall, independently of the number of variants we obtain very high levels of recall. It slightly decrease 7% from 10 to 100 variants, while precision increase, mainly because of the “noise” introduced by non-conformities in feature specification discussed in Section 6.2.2. Table 6.5 also presents time measures of one execution showing that the FCA+SFS technique scales correctly for 100 variants in this benchmark. Concretely, it took only around 15 seconds in total for FCA and SFS. If we include, as part of the feature location process, the time for adapting the variants using the Eclipse adapter (the *Adapt* time mentioned in Section 6.4), in the case of 100 variants it took 35 minutes which is still acceptable.

We used the same Modeling package as input to generate 100 variants with the random selection strategy. As in the previous experiment, we keep the number of features constant given that 10 variants already cover the 173 features. Then, we calculate the results by incrementally adding another 10 variants. Table 6.6 shows the results where we can observe that, with only 10 variants, we have 72.83% of precision. The result with 10 variants generated with this random selection strategy is better compared to using 100 variants generated through the percentage-based random selection which was 66.02% as shown in Table 6.5. This result empirically suggests that the FCA+SFS feature location technique performs better when the variants are more dissimilar. Then, starting with 20 variants we reach 90% precision and then from 40 to 80 variants it stays constant in 93.13%. It is worth to mention that the dissimilarity strategy obtained similar results as presented in Table 6.6. In several runs, for 10 variants we obtain around 70% of precision while for 20 variants we already reach 90%.

Table 6.6: Precision, recall and time measures in milliseconds of the FCA+SFS feature location technique in sets of randomly generated Eclipse packages using the random strategy.

Random	FCA+SFS		Time	
	Precision	Recall	FCA	SFS
10 variants	72.83	86.33	328	190
20 variants	90.49	84.97	400	260
30 variants	91.81	84.97	451	394
40 variants	93.13	84.97	802	603
50 variants	93.13	84.97	1,122	905
60 variants	93.13	84.97	1,485	866
70 variants	93.13	84.97	1,878	2,961
80 variants	93.13	84.97	3,692	1,637
90 variants	93.80	84.97	4,539	1,567
100 variants	93.80	84.97	7,967	2,177

The presented examples are intended to show the capabilities of EFLBench in creating scenarios to compare the results of feature location techniques. Concretely, we have shown how to analyse the result 1) with different number of variants and 2) with the same number of variants but with different degrees of similarity. In the case of FCA+SFS, we provided

empirical evidences that having more available variants do not necessarily means better results in precision. However, dissimilar variants is an important factor for obtaining higher levels of precision.

6.6 Conclusions

We have presented EFLBench, a framework and a benchmark for supporting research on feature location in artefact variants. Existing and future techniques dealing with this activity in extractive SPL adoption can find a challenging playground which is directly reproducible. The benchmark can be constructed from any set of Eclipse packages from which the ground truth is extracted. We have shown examples of its usage with the Eclipse packages of the official releases for analysing four different feature location techniques. We also provide automatic generation of Eclipse variants using three strategies to support the creation of different benchmarking scenarios. We discussed the evaluation of one of the feature location techniques using randomly generated sets of Eclipse packages.

As further work we aim to generalize the usage of feature location benchmarks inside BUT4Reuse providing extensibility for other case studies. We also plan to use the benchmark in order to evaluate existing and innovative feature location techniques while also encouraging the research community on using it as part of their evaluation. Given the high proliferation of techniques, meta-techniques for feature location can be proposed such as voting systems where the results of several techniques could provide better results than using each of them independently. The harmonization provided by BUT4Reuse is an enabler to implement these ensemble techniques and EFLBench could be used for experimentation.

Another interesting open research question is related to the impact in extractive SPL adoption of the results obtained with feature location techniques. We need more empirical analysis of what is the actual meaning of precision and recall by measuring the time and effort required by domain experts to fully locate the features after applying these techniques (i.e., manually removing false positives and adding false negatives).

FEATURE IDENTIFICATION IN MINED FAMILIES OF ANDROID APPLICATIONS

This chapter is based on the work that has been published in the following paper:

- Li Li, Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Mining families of Android applications for extractive SPL adoption. In *Proceedings of the 20th International Conference on Software Product Lines, SPLC 2016, Beijing, China, September 19-23, 2016*

Contents

7.1	Introduction	102
7.2	Families of Android applications	103
7.2.1	An example of feature identification in an Android app family	103
7.3	Identifying families of Android apps in app markets	105
7.3.1	Implementation and preliminary results	106
7.4	Feature identification in selected families using BUT4Reuse	108
7.4.1	BUT4Reuse adapter for Android apps	108
7.4.2	Categorization of Android families after applying feature identification	109
7.5	Conclusions	112

7.1 Introduction

As presented in Section 2.2.2, feature identification is an important activity in extractive SPL adoption. Concretely, it is relevant when there is not enough knowledge about the features included in a set of artefact variants. Case studies are essential in order to empirically evaluate the proposed techniques for this activity, however, there is a:

Lack of realistic case studies to intensively experiment with feature identification techniques

- *We should avoid artefact variants created “in the lab”*: Artefact variants directly derived from an SPL or from feature-based generators are optimistic scenarios for experimenting with feature identification techniques. This automatic derivation does not introduce “noise” that can be originated by manually producing the variants with ad-hoc reuse. For example, maintenance changes such as bug fixes, which are not applied to all family members, have a significant impact on extractive SPL adoption activities [FLLE14].
- *Mining software repositories is challenging*: Software repositories contain a wealth of artefacts and information that can be mined to study development processes and experimentally assess research approaches. The challenge is thus to create a method to identify families of artefact variants that can be used for experimenting with feature identification techniques.

Contributions of this chapter.

- **Android Application families identification (AppVariants)**: We propose an approach to identify families of Android applications (hereafter *apps*) in large app repositories with the objective to provide realistic case studies for research on the feature identification activity of extractive SPL adoption.
- **BUT4Reuse adapter for Android apps**: We describe the design and implementation of an adapter to enable feature identification experimentation within the BUT4Reuse framework.
- **Preliminary categorization of families after applying feature identification in selected app families**: We perform feature identification to discuss the characteristics of some identified app families.

The remainder of the chapter is structured as follows: Section 7.2 defines what we mean by Android app families and includes an example on mining one family of app variants. Section 7.3 presents our solution for the family variants identification process and Section 7.4 presents and summarizes the results of feature identification in selected families. Finally, Section 7.5 concludes this chapter outlining future research directions.

7.2 Families of Android applications

The myriads of smart phones around the globe have given rise to a vast proliferation of mobile apps. Each app targets specific user tasks and there is an increasing number of targeted user profiles. In this context, Android is a leading technology for their development and on-line markets are the main means for their distribution.

Games, weather, social networking services (SNS), navigation, music or news are among the most popular app categories. In our context, it is not strictly necessary that apps within a family belong to the same app category. We focus on interesting cases for experimenting with feature identification during extractive SPL adoption, therefore, we consider a family as several app variants created by reusing assets to fit different customer needs.

We propose discovering families from on-line Android app markets. Regarding other software repositories, GitHub has offered many opportunities to the software engineering research community with its millions of projects which include Android apps. Unfortunately, there are perils in using GitHub data [KGB⁺14]. Among these, it is noteworthy that a large proportion are toy projects or used for sharing code between a limited number of people. Findings on such software data are thus often not generalisable to software development.

7.2.1 An example of feature identification in an Android app family

8684 is a company specialized in travel and transportation apps which reached more than five million users. The 8684 family of variants that we manually identified motivated our interest in trying to automatically identify other families in app markets. In the 8684 portfolio we have found several apps including **CityBus**, **LongWayBus**, **Metro**, **Train** or **TrainTicket**. Given the specialized nature of this company, their apps have many aspects in common such as time schedule or location management. They also have apps from other categories, for example one dedicated to **FastFood** restaurants which also manages locations. Figure 7.1 shows screenshots of the six mentioned mobile apps. We aim to explore the reuse that has been conducted while implementing the different apps and check if some of this reuse corresponds to distinguishable features. Our hypothesis is that, apart from the different graphical elements, there may be shared implementations of the business logic.

We perform a preliminary analysis of the six app variants using the File structure adapter within BUT4Reuse which decomposes the app in File elements as presented in Section 4.3.1. Then, we selected FCA as block identification technique (details in Section 4.3.2, it is based on separating the intersections of elements across the variants) to identify features. In the Java packages that we manually identified as related to the company source code (`cn.tianqu` and `cn.chinabus`), 270 Java files are shared by at least two variants, meaning that they are identical files. On the other hand, 470 Java files in average are specific to each variant. We only consider these two packages because the others are related to general libraries used in Android development.



Figure 7.1: Screenshots of six app variants of the 8684 family.

Figure 7.2 illustrates the blocks of File elements automatically identified among the variants in these two Java packages using the *pruned concept hierarchy* visualisation [Pet01] supported by BUT4Reuse. The six variants are at the bottom of the figure, and we can visualise, by following the arrows, the blocks integrating each variant. For example, *LongWayBus*, which is tagged in Figure 7.2, contains Blocks 10, 5, 6, 2 and 0. The blocks appearing in the *concept* of the variants (e.g., Block 10 for *LongWayBus*), are the ones that are specific to each variant. Block 0, at the top, are source code files shared by the six apps except *TrainTicket* in the bottom right corner. *TrainTicket* is only sharing files with the *Train* app.

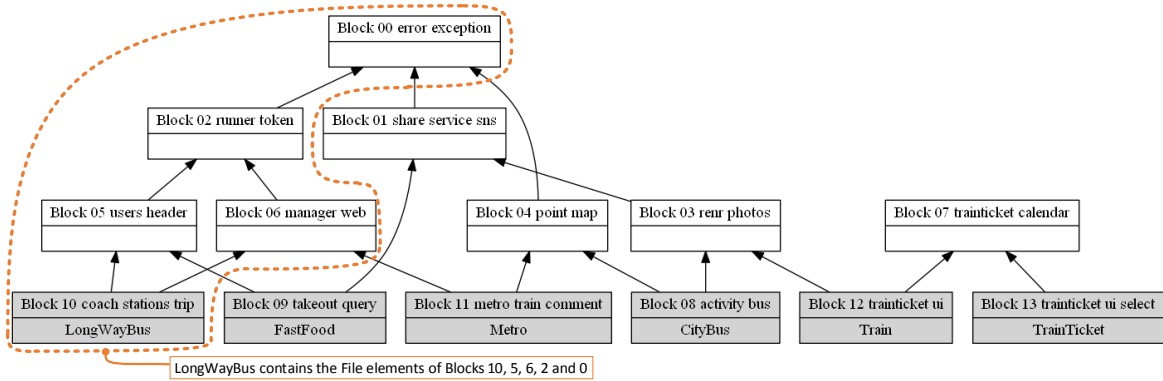


Figure 7.2: Pruned concept hierarchy visualisation showing the variants and blocks in the packages cn.tianqu and cn.chinabus of the 8684 family. The Blocks related to the *LongWayBus* are tagged to illustrate how to interpret this visualisation supported by BUT4Reuse.

We ignore if the reuse was performed using copy-paste-modify or if more advanced reuse techniques were being applied. However, this analysis of artefact variants concludes that reuse is being performed. We analysed the size of the Java files in terms of lines of code (LOC). The variant-specific **Blocks 8 to 11** are 20 KLOC in average, and **Blocks 12 and 13** are 3 KLOC and 5 KLOC respectively. **Block 0**, shared by all apps except one, consists of 10 files with 333 LOC. The other shared **Blocks 1 to 7** have an average of around 2 KLOC.

In Figure 7.2, we included for each block, relevant words present in the file names. Concretely, for obtaining these words, we used the VariClouds approach for block naming during feature identification which will be explained in Chapter 8. **Block 0** seems to be related to error handling while the other blocks can be potentially related to different features shared among the apps. From a research perspective, one objective is to apply and validate techniques for feature identification towards the extraction of an SPL that will be able to, at least, derive the same six apps by selecting their features. Nevertheless, in this chapter, our objective is to automatically identify a large number of other relevant families within app markets to support experimentation.

7.3 Identifying families of Android apps in app markets

Thanks to manual observation, we found three essential and simple characteristics which could be leveraged to identify app families in app markets, including 1) the unique package name of apps, 2) the certificate signed by app developers and 3) code similarity among apps. Figure 7.3 illustrates the working process of our approach, which takes as input a set of apps and output the families of apps. We provide details about these three steps.

1. **Package-based categorization:** In Android, each app is uniquely specified through a full Java-language-style package name. This package can be recognized because it is declared in the app meta-data. As officially recommended by Googleⁱ, to avoid conflicts with other apps, developers should use internet domain ownership as the basis for their package names (in reverse). As an example, apps published by Adobe should start with *com.adobe*. Thus, if two apps start with a same company domain in their package names, these two apps are more likely developed by the same provider.

ⁱDeveloper guide: <http://developer.android.com/guide/topics/manifest/manifest-element.html>

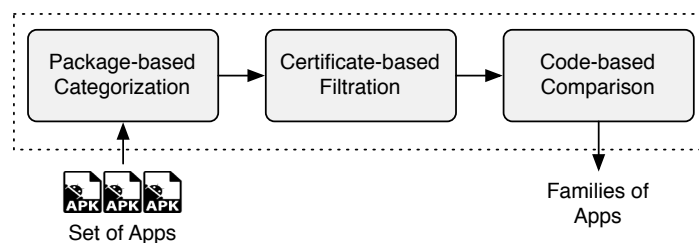


Figure 7.3: Steps of AppVariants app families discovery approach.

2. **Certificate-based filtering:** Unfortunately, the package naming style is not respected in every case. Concretely, it is a common practice for malicious apps to use the same package names as other legitimate apps [LLB⁺16]. This is possible with a technique called *repackaging* where apps are disassembled and assembled again. Therefore, the validity of the package name characteristic is threatened. We complement the first characteristic with another one related to the certificate of apps. In order to make an app publicly available in a market, developers have to sign their apps through their unique certificate. Thus, if two apps are signed by a same certificate, we have reason to believe that these two apps are developed by the same provider.
3. **Code-based comparison:** In the last step, we perform pairwise comparison on apps that are located in the same family thanks to the aforementioned two steps. Based on the comparison results, we attempt to filter out such apps that are not sharing code with others and consequently can be considered as outliers. Given a threshold t , an app family F , and an app $a_i \in F$, for every $a_j \in F$, where $j \neq i$, if the similarity distance $a_{ij} < t$, we consider that a_i is an outlier of family F , and thus we drop it from F in this step. The source code similarity is computed with the formula $similarity = identicalMethods/totalMethods$, which has already been used in other studies [LBKLT16]. Because this step is more computationally expensive than the previous two, we only use it to exclude outlier apps from the app families.

Summarizing the mentioned steps, we cluster apps into the same family if they belong to the portfolio of the same app provider (step 1 and 2) and as long as they are partially similar in terms of source code similarity (step 3).

7.3.1 Implementation and preliminary results

We implemented AppVariants which is dedicated to validate the pertinence of the aforementioned three characteristics. It is important to clarify that we are interested in variation in space (variants) and not in variation over time (versions) [ABKS13]. However, we use the information of the versions to decide which versions of the variants should be simultaneously used to perform feature identification.

Prototype implementation: AppVariants adopts a tree model to store the meta information of the identified apps. Figure 7.4 shows an illustrative example for `com.baidu` apps on how their meta-data are stored. Each non-leaf node of the tree is represented by a package segment (e.g., `baidu`) while leaf nodes are represented through the remaining package segments (e.g., `BaiduMap`). Furthermore, each leaf node is accompanied by a ranked list of meta-data of apps, including their certificates and assemble times. As shown in Figure 7.4, the vertical axis is a time line referring to different versions of the app indicated by the leaf node. Given a time point, the tree model also provides a way to identify family variants. For example, as shown in the dashed rectangle, we are able to collect a set of variants for `com.baidu` given the latest time point. In our current implementation, for the selection of the version of the apps within a family, we use the latest available versions of each of them.

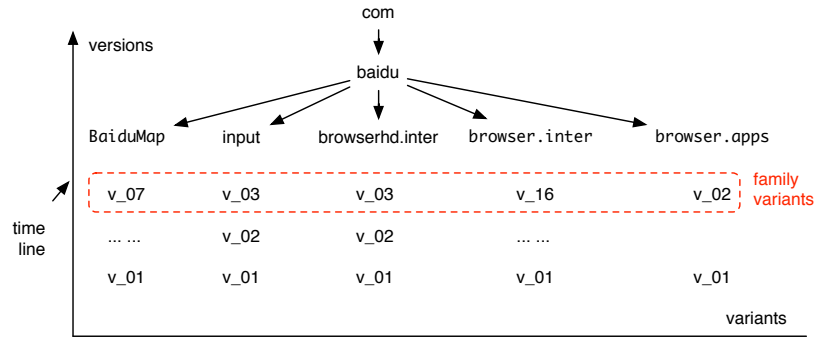


Figure 7.4: A simplified example showing how meta-data of app variants of `com.baidu` are stored.

Experimental setup: For the purpose of providing real family variants for feature identification research, we need to apply our approach on a market scale. Thus, for this mining process, we use a part of the AndroZoo repository which is a collection of apps crawled from several markets to support research in Android [ABKLT16]. Concretely, we select around 1.5 million apps from AndroZoo belonging exclusively to the Google Play market. This data set has already been used in other analyses [LBB⁺15b, LBB⁺15a].

Preliminary mining results: Among the 1.5 million Android apps, based on package-based categorization and certificate-based filtration (steps 1 and 2), we are able to collect 75,963 families of apps. The amount of variants in each family ranges from 2 to 12,702 apps. Figure 7.5 shows the distribution of the number of variants in our collected families before the third step using a boxplot. The median number of variants is three, meaning that half of the collected families have at least three variants. Given that we do not show outliers in this boxplot, only 88% of the families in the mined data set are shown. The remaining 12% are families with more than 10 variants each. Furthermore, 760 families, representing the 1% of our mined dataset, have over 100 variants. Table 7.1 shows the top 10 families regarding the number of apps in the app provider portfolio. As an example, `com.andromo`, the top ranked package prefix, has 12,702 variants. In the next section we will explain the reason behind this large amount of apps.

Thanks to our manual observation, we have found that, based only on the first two steps of our approach, we are already able to collect a big data set of families. However, some families do not share any common source code from one another. For example, the family `com.ethereal` contains four variants, where the similarity of all of them is below 5%. To this

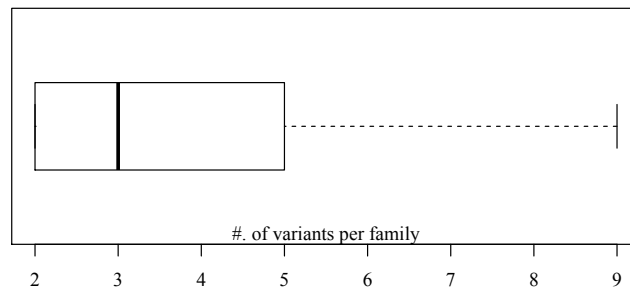
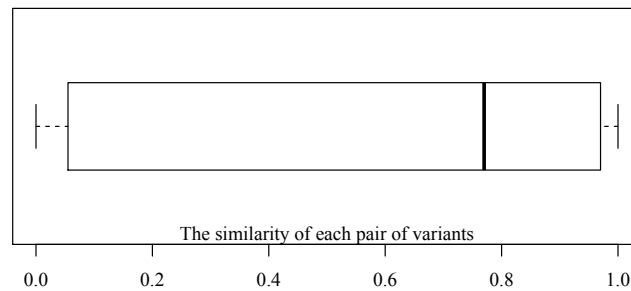


Figure 7.5: Boxplot on the number of variants per each family.

Table 7.1: Top 10 families in number of variants.

Package Prefix	Variants
com.andromo	12,702
com.conduit	11,766
sk.jfox	9,055
com.reverbnation	3,932
air.com	2,732
com.jb	2,203
com.appsbar	2,143
com.skyd	1,922
com.appmakr	1,787
com.gau	1,764

end, we leverage code-based comparison (step 3) in order to mitigate those potential false positives. As our pairwise code comparison between apps is not scalable to families with large number of variants, to evaluate the soundness of this step we randomly selected 100 families of less than 10 variants. Figure 7.6 illustrates the distribution of similarities between pairs of variants within the families. The median value is around 77%, showing that variants of most families are actually sharing a lot of common code. This fact makes them good samples for exploring and assessing feature identification techniques.

**Figure 7.6:** Distribution of the similarity of the variant pairs.

7.4 Feature identification in selected families using BUT4Reuse

In order to perform more in-depth feature identification analyses than the one shown in Section 7.2.1 regarding the 8684 family, we implemented a BUT4Reuse adapter for Android apps. This section presents the design of the adapter and its usage in selected families obtained through AppVariants.

7.4.1 BUT4Reuse adapter for Android apps

Android apps are distributed in the form of *apk* files (Android application package) which contains all the resources for the execution of the app. In Section 4.2.2, we described the

tasks needed to design an adapter. We detail the design decisions of the first task, element identification, as it is specially relevant for the feature identification activity. We decided to simultaneously decompose the app artefact variant at different levels: files, source code and meta-data defined in the Android manifest file. Therefore, to implement the Android adapter, we complement the available Java source code adapter (AST elements) and Files adapter (File elements) with Permission elements. Permission elements help in the comparison of the different privileges needed by the app for its execution. For example, accessing the information of the phone contact list, or using the camera, can be required by one variant but not in others.

The apk file need to be pre-processed before the Android adapter can decompose the app. As part of the BUT4Reuse Android adapter, we automated a chain to unpack these files and decompile them. Figure 7.7 presents this chain and the used techniques. First, the apk file is uncompressed to obtain all the resources. Then, there are two files requiring further processing. The file *classes.dex*, which is a Dalvik executable used in Android, is transformed to Java byte-code with the *dex2jar* tool ⁱⁱ, and then decompiled with the Java decompiler called *jd-cmd* ⁱⁱⁱ to obtain actual Java classes that can be parsed by the adapter. Also, *AndroidManifest.xml* is a binary file that can not be read without unpackaging it with a dedicated method provided by the Android asset packaging tool. From this manifest file, the adapter creates elements related to Android permissions.

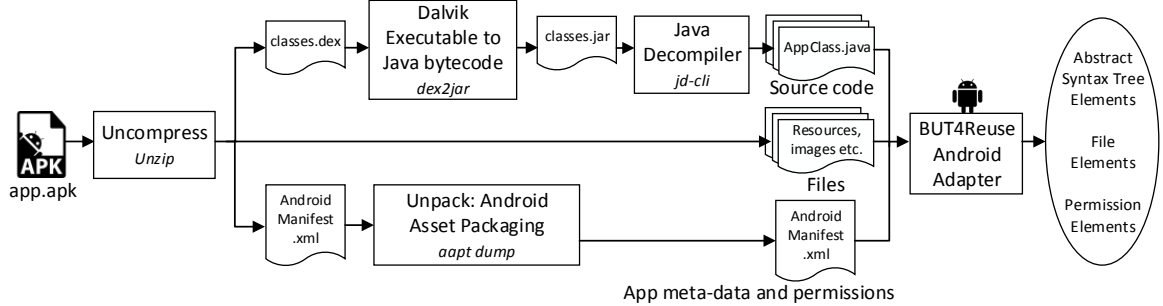


Figure 7.7: Pre-processing of the apk file for its decomposition in elements by the Android adapter.

7.4.2 Categorization of Android families after applying feature identification

In this section we present the categorization of four types of mined families and we present a representative example of each of them.

- **Feature-based app generators and distributors:** In Table 7.1 we presented app provider portfolios with elevated number of apps. We analysed variants from the `com.andromo` family to investigate the reason behind this fact. Blocks are intensively shared between variants, and they all share a common block which is mainly related to screen layout management. Preliminary feature identification within the app variants confirmed that the Andromo framework is feature-oriented providing initial components for user customization.

ⁱⁱ Dalvik executable to Java byte-code, dex2jar tool: <https://github.com/pxb1988/dex2jar>

ⁱⁱⁱ Java Decompiler, command line version, jd-cmd tool: <https://github.com/kwart/jd-cmd>

By looking at their website, Andromo is a company that offers a framework aiming at hiding technical details of app development. Figure 7.8 shows examples of apps created with Andromo. They also provide services for easing app distribution. That is the reason why they all have the same unique certificate `com.andromo` even if they are from different developers within their community. Concretely, an example of the main package of an app is `com.andromo.dev282094.app267841` where `dev` is the id of the user and `app` is the id of the app. In their website^{iv} we can read “*You can add [...] features to your app [...]. Interactive maps, photo galleries, blog/news feeds, embedded websites, mobile websites, HTML5/Javascript code, YouTube videos, Twitter feeds, Facebook pages, music tracks, soundboards [...]*”.

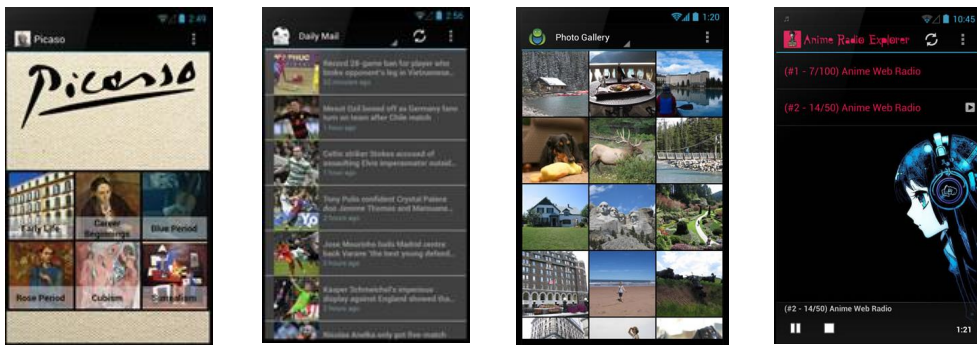


Figure 7.8: Screenshots of applications generated with Andromo. The first app is a thematic app with several features, the second app uses the RSS feed feature, the third uses the photo gallery feature and the fourth uses the radio feature.

Despite that user customizations complicate feature identification in our mined app variants, in-depth analysis of frameworks like this one could aim to reverse engineer their variability and architecture. Given that the business value of these frameworks is their architecture, protection from extractive approaches, which has been coined as *variability-aware security* [ABC⁺15], is an important research direction that can highly benefit from this category of mined families.

- **Content-driven variability:** In this category of family we have variability at the displayed content level while the app architecture remains the same. For example, we analysed the `tudou` family with six variants. Tudou is a Chinese on-line video services company that broadcast users content and television series. We found three which were very similar, almost complete app clones. A manual examination showed that they reused the same source code to distribute apps targeting different television series. Figure 7.9 shows two of these apps where only the content is changing while the app structure and functionality remains constant.
- **Device-driven variability:** The variability in this category exists because of the targeted devices. We analysed the six variants from the `baidu` family, a Chinese web services provider. In two of them, `browser.inter` and `browserhd.inter`, we found that much code has been reused. They are both internet browser apps where one is

^{iv}Features provided by Andromo for app creation: <http://andromo.com/features>



Figure 7.9: Screenshots of two different apps with content-driven variability.

specialized for mobile phones and the other for tablets. Figure 7.10 shows a screenshot of these two apps. It is not unusual to provide the same app in terms of functionalities, but specialized for different devices.

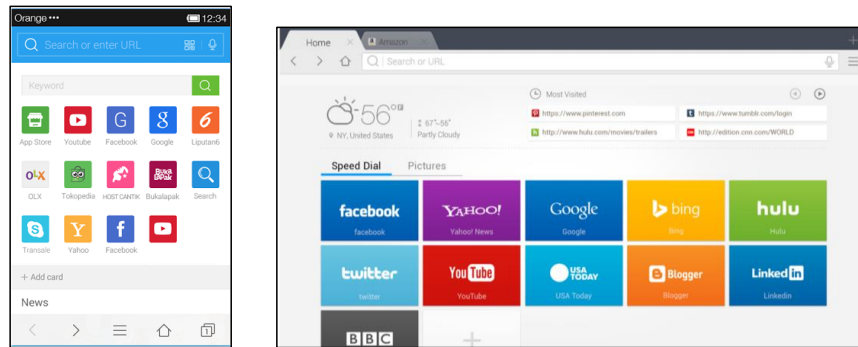


Figure 7.10: Screenshots of two apps with device-driven variability.

- **Libraries reuse (AppVariants undesirable cases):** We have found mined app families where the reuse between variants only concerns the use of the same libraries. As an example, we analysed a mined family with four apps: **FrancsEuros**, **StrongBox**, **MyShopping** and **BattleSpace**. Figure 7.11 shows screenshots of these apps which are a currency converter, a quiz game, a shopping list manager and an arcade game respectively. The identified reuse concerns the code of the used library `fr.pcsoft` while slight reuse between the apps was identified in the main package. A manual examination showed that reuse in the main package belongs to technicalities of the used development environment (i.e., WinDev provided by pcsoft) but not to actual relevant code.

In the **baidu** family presented before, Baidu also provides an app software development kit which they extensively use in the six **baidu** variants. Apart from the mentioned internet browser apps, in two apps, **Baidu Maps** and **input**, we identified that they share a voice recognition feature. However, the rest of the reused elements are only related to their app development kit.

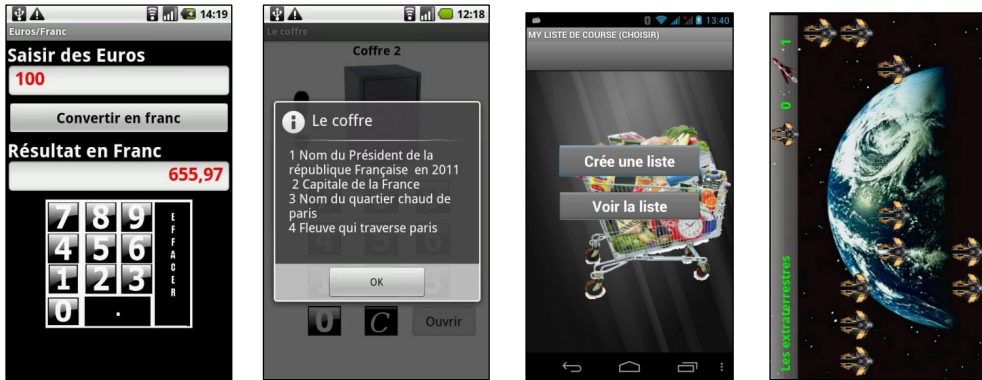


Figure 7.11: Screenshots of a mined app family that is only reusing libraries.

As presented in Section 7.3, the third step of the mining approach related to source code similarity does not restrict the similarity analysis to the main package of the apps. The analysis is performed in the whole source code including libraries. In general, around 41% of an app source code is related to libraries [LBKLT16]. In the field of apps clone detection, as well as in our approach, distinguishing actual source code from libraries is a known limitation [LBKLT16]. In fact, there is no explicit meta-information about the code that corresponds to libraries in Android apps. The main technique to avoid these false positives is to use white-lists with known and frequent libraries [LBKLT16]. However, this does not guarantee that all libraries will be filtered during our third step regarding the similarity calculation.

We do not intend this categorization to be exhaustive but we present representative examples of our analysis of the identified families. We have shown that the results of AppVariants are promising to enable intensive experimentation of feature identification techniques.

7.5 Conclusions

Android app development is a flourishing industry whose products are publicly available in app markets. We propose a method, called AppVariants, to identify app families with the objective to provide realistic case studies for research on the feature identification activity of extractive SPL adoption. The identified app families serve to study reuse practices and provide assessment about app families suitability for SPL adoption. We presented our implementation of the mining process and the analysis of selected families.

As further work, we can envision an automatic approach to explore the identified app families to determine the semantic cohesion of the family members. The inclusion of natural language analysis of app descriptions and user comments in the market website could help in improving this mining. The mining process of AppVariants could also be extended with approaches that could escape cases of libraries reuse or code obfuscation. Also, concerning feature identification within app families, we performed preliminary analysis of several families but more in-depth experimentation on feature identification and extractive SPL adoption could be performed.

Part IV

ASSISTANCE AND VISUALISATION
IN ARTEFACT VARIANTS ANALYSIS

8

NAMING DURING FEATURE IDENTIFICATION WITH WORD CLOUDS

This chapter is based on the work that has been published in the following paper:

- Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Name suggestions during feature identification: The VariClouds approach. In *Proceedings of the 20th International Conference on Software Product Lines, SPLC 2016, Beijing, China, September 19-23, 2016*

Contents

8.1	Introduction	116
8.2	VariClouds approach	117
8.2.1	Word clouds and weighting factors	117
8.2.2	Retrieving the words through the adapters	117
8.3	Using VariClouds: Phases for the domain experts	119
8.3.1	Phase 1: Preparation of the word clouds	120
8.3.2	Phase 2: Block naming	123
8.4	Case studies and evaluation	125
8.4.1	Requirements of selected case studies	125
8.4.2	Quality of the word clouds	127
8.4.3	The benefits of summarization	129
8.5	Threats to validity	130
8.6	Conclusion	131

8.1 Introduction

As discussed in Section 2.2.2, in extractive SPL adoption, identifying the features across the artefact variants is a needed activity when there is no complete information about the features within a domain. The lack of complete documentation, the amount of fields of expertise or just the complexity of artefacts created to satisfy different customer needs, can lead to the lost of the global view of the existing features. During the feature identification activity, there is the naming sub-activity where we need to unequivocally provide names to the features. However, there is a:

Lack of assistance during feature naming for domain experts.

- *Domain experts need interactive visualisations to reason about possible names.* We detailed in Section 3.1.2 several reported experiences in naming during feature identification showing the general lack of support for domain experts. In practice, automated comparison approaches will identify distinguishable blocks shared by the artefact variants, and would thus still require domain experts to manually map them with actual features. To that end, domain experts must look at the elements of the blocks, understand their semantics, and guess the functionality that each block provides when present in a variant.

Contributions of this chapter.

- **VariClouds:** A visualisation that provides support for feature naming during feature identification. This paradigm facilitates the first encounter between the domain experts and the variants so as to help in understanding the semantics hidden in the variants as well as in the variability among them. Concretely, in order to suggest names, we leverage word clouds, a widely used visualisation paradigm for textual data [Smi07]. Blocks can be renamed by interacting with the suggested words of the word clouds.
- **An automated approach** for heuristically assigning the names of the blocks based on the same weighting factors used for displaying the word clouds. Domain experts can use this option for one of the blocks, or to all of them, as a previous step for manual validation or refinement.
- **An integrated visualisation** in the process promoted by the BUT4Reuse framework presented in Chapter 4. BUT4Reuse adapters are extended to provide meaningful words from the elements obtained when decomposing artefacts.

The remainder of this chapter is structured as follows: Section 8.2 presents VariClouds and Section 8.3 presents its phases for domain experts. We present and discuss the evaluation and the characteristics of the selected case studies in Section 8.4. Section 8.5 discusses the limitations and threats to validity we identified in our approach. Section 8.6 concludes the chapter and presents further work.

8.2 VariClouds approach

The VariClouds approach leverages word clouds to visualise variants, or parts of variants, providing a means for exploratory analysis in the context of the feature identification activity during extractive SPL adoption. More specifically, it is used to solve the problem of naming. We first provide background information on word clouds. Then, we detail VariClouds by explaining how words are retrieved from product variants. Later, we explain the phases from the perspective of a domain expert.

8.2.1 Word clouds and weighting factors

Word clouds gained momentum in the Web as aggregators of activity, as a means to measure popularity, and as a mechanism for social tagging/indexing [Smi07]. Word clouds have been also used for text summarization and analysis in several domains such as patent or opinion analysis [KBGE11, WWL⁺10, SGLS07]. The principle underlying word clouds is the weighting factor of the words appearing in a document. This weighting factor is used to change the relevance of the word in the visualisation, typically by assigning larger font sizes to the more weighted words (e.g., more frequent words).

Term frequency (**tf**) is a metric consisting in giving more relevance to the terms appearing with more frequency in a document d . When dealing with a set D of documents d_1, \dots, d_n , *term frequency-inverse document frequency* (**tf-idf**) is another metric used in IR [SWY75]. For a document d , **tf-idf** penalizes common terms that appear across most of the documents in D and emphasizes those terms that are more specific to d . There are different formulas to calculate them. In this work, we used the formulas presented in Equation 8.1, where we use *raw term frequency* (**tf**) which is calculated counting the occurrences of a given term in a document, *inverse document frequency* (**idf**) which measures how much rare or common a term is across all the documents using a logarithmic scale and, finally, **tf-idf** uses **tf** multiplied by **idf** to penalize or encourage a term depending on its occurrence across D .

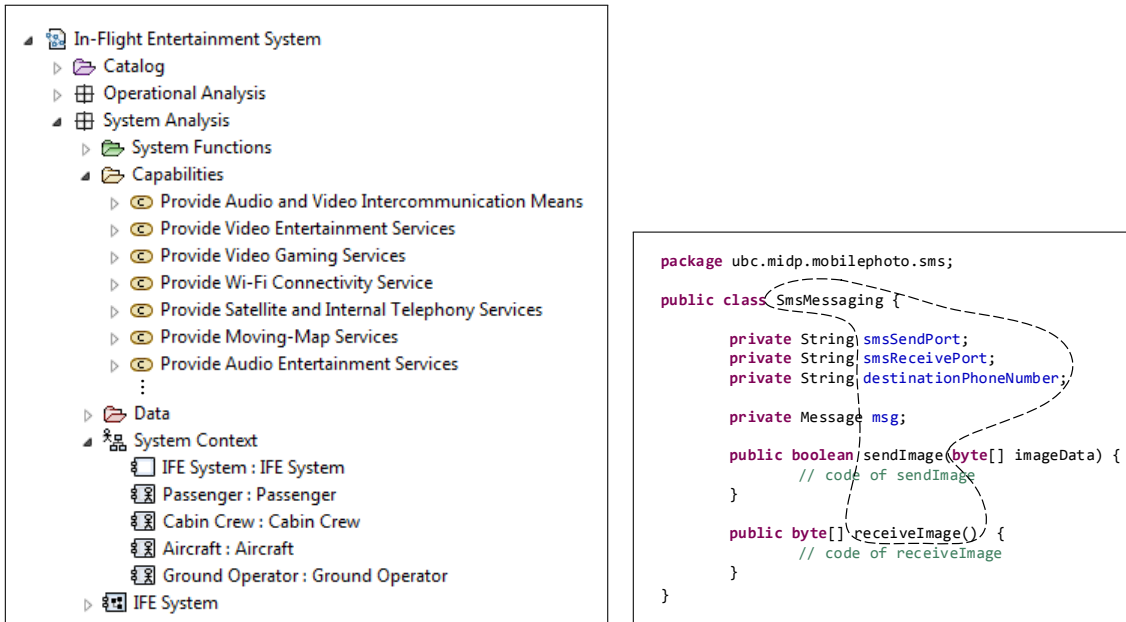
$$\begin{aligned}
 tf(term_i, d) &= f_{term_i, d} \\
 idf(term_i, D) &= \log \left(\frac{|D|}{|\{d \in D : term_i \in d\}|} \right) \\
 tf-idf(term_i, d, D) &= tf(term_i, d, terms) \times idf(term_i, D)
 \end{aligned} \tag{8.1}$$

8.2.2 Retrieving the words through the adapters

As presented in Chapter 4, each adapter is associated to an artefact type and is responsible for decomposing a given artefact into a set of elements. The first task for designing an adapter is the identification of the elements that want to be considered. To support VariClouds, each

adapter must be enriched to yield the words exposed for each element. A word is therefore a term that can be inferred from an element (e.g., the label of a model element or the name of a Java method). We overview below how the adapters for artefact types relevant to our case studies were implemented:

- **Models adapter:** Models can be decomposed in atomic model elements as we presented in Section 5.3.1. Specifically, we consider the *Class*, *Attribute* and *Reference* which are the mainly used concepts to define domain-specific languages (DSLs). We get the words through technicalities of the Eclipse Modeling Framework (EMF) [Ecl16a]. To get the text of a class instance we use its *EMF Item Provider*. Item providers are registered for each class type and they have a label provider that return its associated text as defined by the DSL implementation. Figure 8.1a shows an excerpt of a model with the text retrieved from the label provider of each class. Concretely, it is a screenshot of the EMF reflective editor which uses these label providers. For the Attributes, we get the value of the attribute in string format. We decided to ignore the text from References and we did not evaluate the impact of adding the text of all the referenced classes. We further post process the text of Class and Attribute by tokenizing the text to obtain separated words.
- **Eclipse adapter:** In Section 4.4.1 we presented the Eclipse adapter for decomposing an Eclipse package in its set of plugin elements. Also, some plugin examples were shown in Section 6.2 (e.g., *Eclipse CVS Client* or *CVS Team Provider*). For each plugin, we take its name as defined by the plugin providers in the plugin metadata. We also tokenize the name to obtain the set of words.



(a) Excerpt of the In-Flight Entertainment System model [Pol15] to illustrate the meaningful words retrieved using the models adapter.

(b) Excerpt of a Java class from the MobileMedia case study [FCS⁺08] to illustrate the meaningful words retrieved using the source code adapter.

Figure 8.1: Examples of words retrieved from two excerpts of artefact variants.

- **Source code adapter:** In source code, classes and methods usually have meaningful names provided by developers which can be exploited to retrieve relevant words regarding the implemented functionality. Automatic summarization [Jon07] is a research field extensively studied in source code artefacts to generate documentation or helping developers in program comprehension (e.g., [SPV11]). In our context, the BUT4Reuse source code adapter decomposes the artefact in Abstract Syntax Tree (AST) elements. To retrieve words, we use class names, method names and declared field names contained in these elements. We ignored the text in the body of the methods or in source code comments. Figure 8.1b shows, inside the dashed zone, the words retrieved from the source code adapter in an excerpt of a Java artefact.

The adapters mechanism, and its flexibility to implement the way that words are extracted, enable the genericity of VariClouds to support different artefact types.

8.3 Using VariClouds: Phases for the domain experts

Figure 8.2 shows the process of VariClouds from the perspective of the domain experts which consists of two phases: 1) Preparation and 2) Block naming. At the top of the figure, we show the BUT4Reuse process which is needed to use VariClouds. Concretely, the artefact variants are decomposed in elements through an adapter, and a block identification technique is used to obtain the blocks. In the use of VariClouds, we assume that an effective block identification technique is available and yields the necessary blocks for use as part of the feature identification process. Thus, it is important to clarify that block identification techniques themselves are out of the scope of VariClouds. We presented block identification techniques in Section 4.3.2.

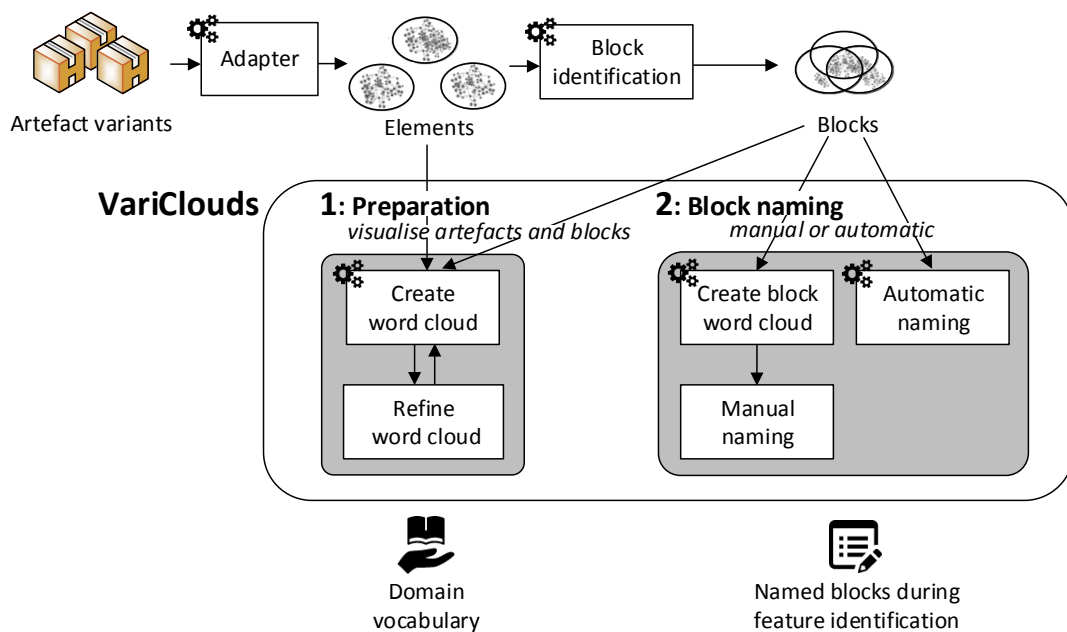


Figure 8.2: Overview of the VariClouds process with the preparation and block naming phases.

The two phases are explained in detail in the next two subsections. In short, in the preparation phase, during the creation of a word cloud, several word filters can be applied and tuned. Therefore, this phase aims to refine the word cloud creation by visualising the summarization of all variants, specific variants or identified blocks. In the second phase, block naming is performed through visualising the **tf-idf** word cloud of each block. Alternatively, the domain expert can use an automatic algorithm to name the blocks.

Each phase is explained using the running example of six variants of Vending Machine statechart models. Figure 8.3 shows the statechart diagrams of variants number 1 and 4 which cover the features available in the family: Three different beverages (coffee, tea and soda), two payment systems (cash and credit card) and ring tone alert. The objective of this running example is to show how to use VariClouds for naming during the process of identifying these features.

8.3.1 Phase 1: Preparation of the word clouds

The word clouds can be created using the words from any set of elements. Figure 8.4 shows three examples of word clouds where Figure 8.4a presents frequent words considering all the vending machine variants (**tf** weighting factor). Therefore, it is a summarization of the whole

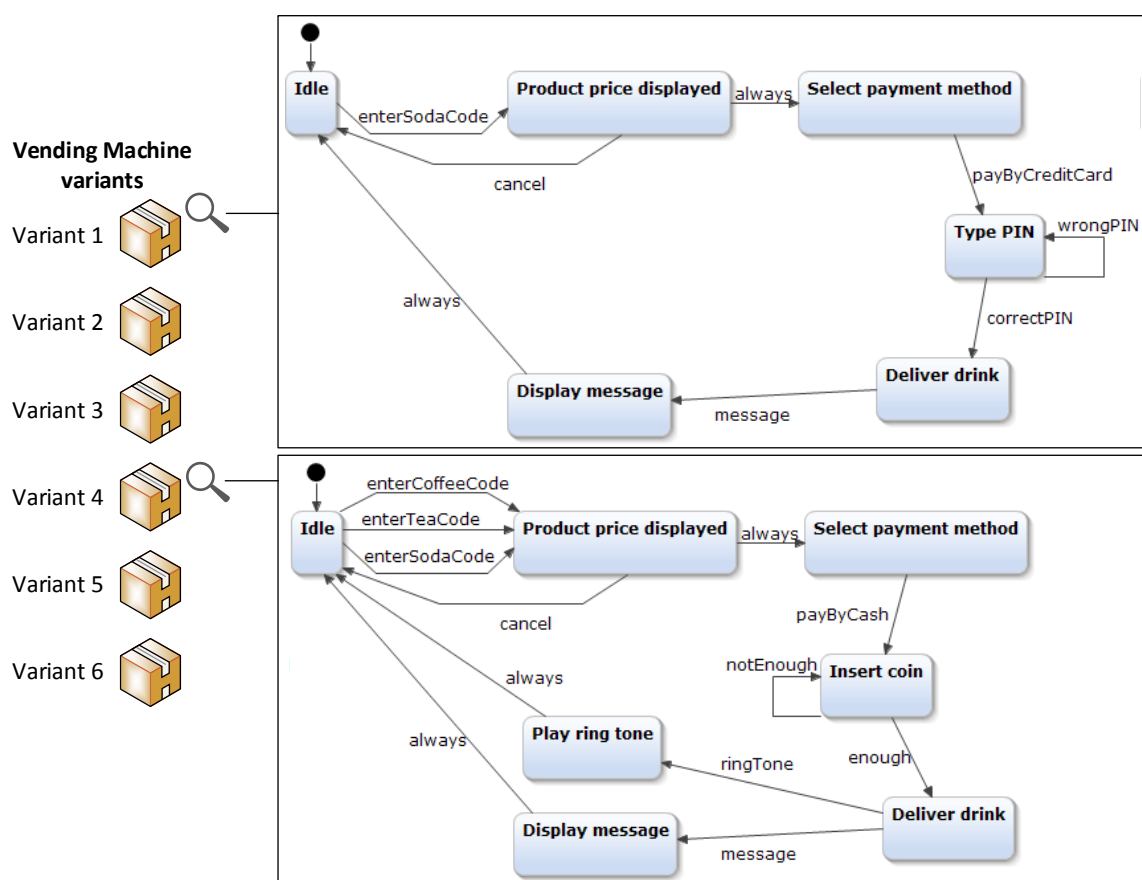


Figure 8.3: Vending machine variants and statechart diagrams of variants 1 and 4.

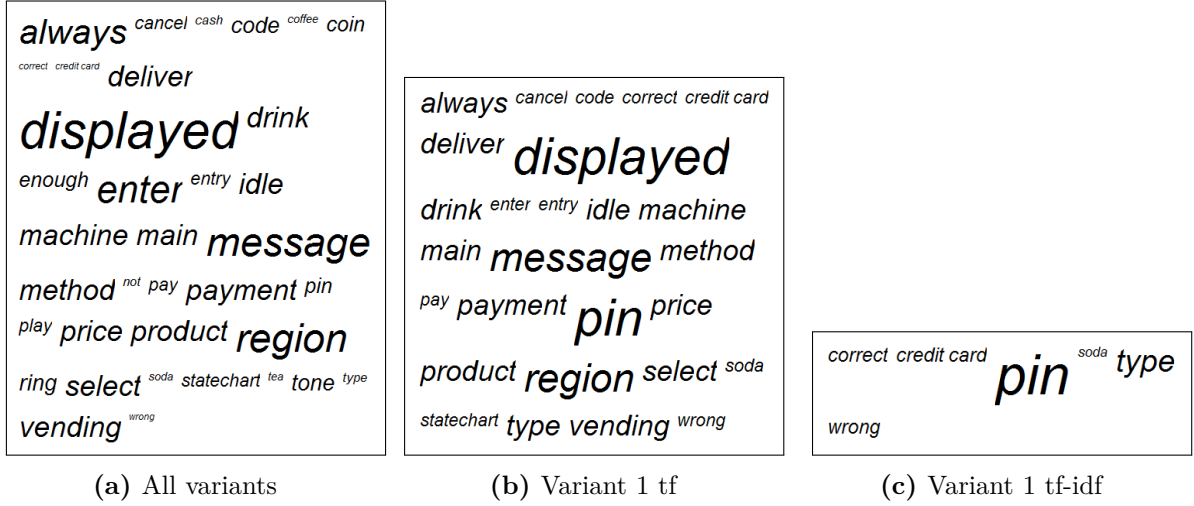


Figure 8.4: Word cloud visualisations of vending machine variants.

family. Figure 8.4b displays the words which are frequent (**tf**) in **Variant 1**. Figure 8.4c shows another word cloud of **Variant 1** but using the **tf-idf** weighting factor. That means that, being six the number of variants, the words appearing in Figure 8.4c are the words that make **Variant 1** special regarding to the other five variants (i.e., management of the pin of a credit card and providing soda).

More meaningful word clouds can be obtained by pre-processing the words of the elements provided by the adapter. The domain experts can explore word clouds, as the ones in Figure 8.4, in order to include word filters and to refine filter settings. Therefore, the preparation phase is an iterative process where the domain experts create and refine the word clouds until they consider that the domain vocabulary is correctly represented in them.

Figure 8.5 shows the automatic *create word cloud* step of VariClouds shown in Figure 8.2, and illustrates a possible chaining of filters. Regarding the implementation of the word cloud creation, we have implemented a set of well-established filters in the information retrieval community, and for displaying the word cloud, we used OpenCloudⁱ. We present some details about the implemented filters whose activation are optional to the domain experts:

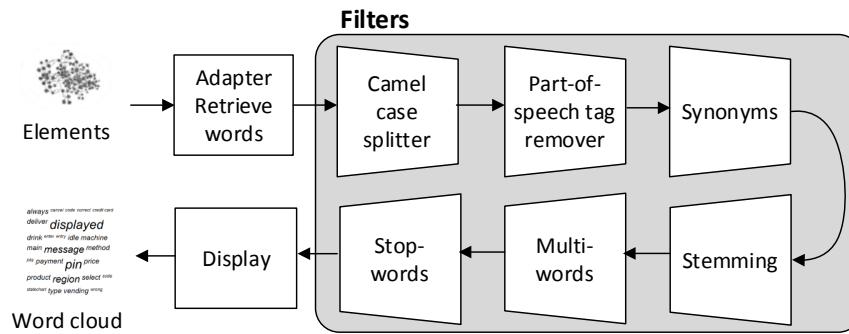


Figure 8.5: Creation of a word cloud with VariClouds to summarize a set of elements. The words retrieved from the elements are filtered and displayed as a word cloud.

ⁱOpenCloud library: <http://sourceforge.net/projects/opencloud>

- **CamelCase splitter:** CamelCase is a de-facto naming convention for source code that assemble words to avoid white spaces. For instance, the method `getPersonAddress` is split in the words “get” “person” and “address”.
- **Parts-of-speech tags remover:** These techniques analyse and tag words depending on their role in the text. For example, we can decide to remove conjunctions (e.g., “and”) and articles (e.g., “the”) as they may not add relevant information and they are very frequent. We apply Part-of-Speech Tagger techniques as implemented in OpenNLP [Apa10]. In Figure 8.6, this filter removed the word “by” because it was tagged as a preposition.
- **Synonyms:** This filter calculates possible synonyms for each pair of words by using a similarity threshold. We used an implementation of WordNet [fJ15] and the WUP similarity metric [WP94]. In Figure 8.6, “insert” and “enter” were automatically detected as synonyms.
- **Stemming:** This filter considers as equal all words that have the same root. For example, “playing” stems from “play” and it is considered equal to “play” if this word appears. Instead of keeping the root, we keep the word with greater number of occurrences to replace the involved words. For the implementation we used the Snowball steamer [Por01]. In Figure 8.6, “display” and “displayed” had the same stem.
- **Multiwords:** Multiwords are a set of words that should not be separated [SBB⁺02]. Multiwords will be treated like an unique word. In Figure 8.6, “credit card” was included by the user as a multiword.

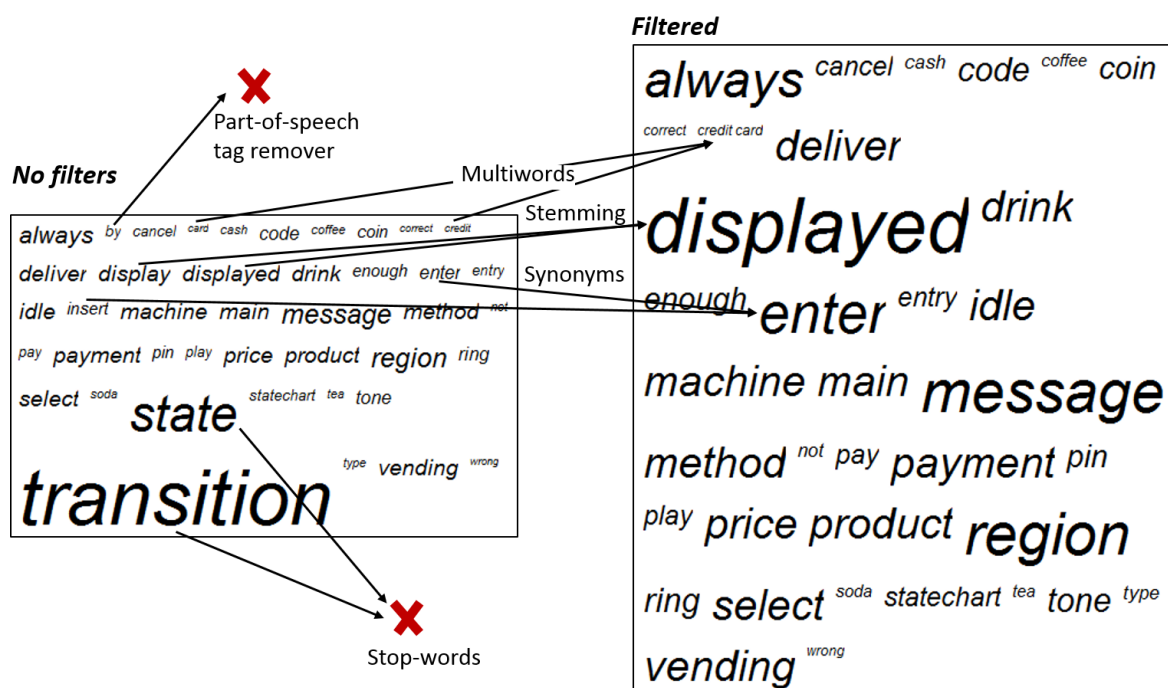


Figure 8.6: Word clouds of all vending machine statechart variants without any filter and word cloud after filters refinement.

- **Stop words:** Stop words are words to be ignored because they do not add information. The users can define their own stop words. In Figure 8.6, “state” and “transition” were included as stop words as they are just language concepts in statecharts which are very frequent and that do not add relevant information.

The users that select and configure some of these filters must be aware that these techniques can have positive improvements in their task but it also can represent a risk of losing information. We present examples in Section 8.4.2.

8.3.2 Phase 2: Block naming

Blocks are obtained using the block identification technique. Then, the domain experts use interactive word cloud visualisations constructed with the elements of each block in order to name each block. The hypothesis of the VariClouds approach for block naming is that relevant words are those that make each block special regarding the rest of the blocks. For this reason, **tf-idf** weighting factor is used.

Figure 8.7 shows the **tf-idf** word clouds of the blocks identified while automatically analysing and comparing the vending machine statechart variants of the running example. As presented in Section 8.2.1, **tf-idf** penalizes the words appearing frequently in most of the blocks giving more weight to the words that make this block special. For example, “code” and “enter” appear in Blocks 2, 3 and 4 and that is the reason why these words lose weight compared to the different beverage names “soda”, “coffee” and “tea” which are, in this case, the actual feature names. In Block 6 we have “pin” as the largest word while “credit card”, the actual feature name, is also present but smaller. The reason is that both “pin” and “credit card” are special for this block but “pin” is more frequent. The domain experts can interact on the suggested names to set the block names.

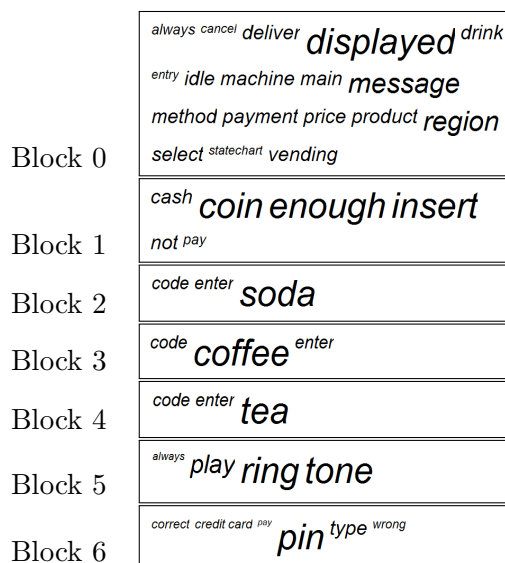


Figure 8.7: Word cloud visualisations of the identified blocks in the vending machine statechart variants.

VariClouds approach proposes an automatic algorithm to suggest the names of each block. Algorithm 1 shows the pseudo-code of this automatic renaming which is not complex. The parameter k is the initial number of words to be used when renaming. Using $k = 1$ each block name will initially have only the word with highest **tf-idf** score. As we will present later, given the empirical results of our case studies, by default we suggest to use at least $k = 2$. For each block, we assign the concatenation of the first k words with the highest **tf-idf** scores (lines 2 to 12). When renaming all blocks, it is possible that two or more blocks have the same name. Therefore, we avoid name conflicts by iteratively appending the word with the next highest score until there are no conflicts (lines 13 to 33). If there are no more remaining words in the ranking and there are still conflicts, we append different numbers at the end of the name.

Algorithm 1 Automatic renaming of blocks.

```

input: blocks, k                                ▷ k is the number of initial words, 2 by default
output: names of the blocks
1: conflictSet  $\leftarrow \emptyset$ 
2: for  $block_i \in \{blocks\}$  do                                ▷ concatenate the first  $k$  words with the highest tf-idf scores
3:   wordsRanking $_{block_i} \leftarrow \text{sortWordsTFIDF}(block_i, blocks)$ 
4:   name  $\leftarrow$  empty string
5:   for  $j \in [1..k]$  do
6:     name.append(wordsRanking $_{block_i}(j)$ )
7:   end for
8:   if name  $\in$  names then                                ▷ prepare the list of name conflicts
9:     conflictSet.includeInConflictFor(name)
10:  end if
11:  names $_{block_i} \leftarrow$  name
12: end for
13: while conflictSet  $\neq \emptyset$  do                                ▷ resolve all name conflicts
14:    $k \leftarrow k + 1$ 
15:   newConflictSet  $\leftarrow \emptyset$ 
16:   for conflict  $\in$  conflictSet do                                ▷ try the concatenation of the next word for each conflict
17:     auxNumber  $\leftarrow 1$                                 ▷ number to add in case that there are no more words
18:     for  $block_i \in$  conflict do
19:       name  $\leftarrow$  names $_{block_i}$ 
20:       if  $k < \text{wordsRanking}_{block_i}.\text{size}$  then
21:         name.append(wordsRanking $_{block_i}(k)$ )
22:       else
23:         name.append(auxNumber)
24:         auxNumber  $\leftarrow$  auxNumber + 1
25:       end if
26:       if name  $\in$  names then                                ▷ check if the conflict persists
27:         newConflictSet.includeInConflictFor(name)
28:       end if
29:       names $_{block_i} \leftarrow$  name
30:     end for
31:   end for
32:   conflictSet  $\leftarrow$  newConflictSet
33: end while
34: return names

```

8.4 Case studies and evaluation

We assess the soundness of VariClouds via experiments for answering the following research questions:

- RQ-1: To what extent can the blocks of implementation elements (e.g., source code elements) automatically provide insights that correspond to expert judgement about the semantics of these blocks?
- RQ-2: Is the word cloud visualisation paradigm effective for naming during feature identification?

The following three subsections present the case studies and the answers to the two research questions.

8.4.1 Requirements of selected case studies

We consider case studies that satisfy three conditions:

1. Previously published/used by the SPL research community.
2. The artefacts of the case studies (i.e., variants) are publicly available.
3. The name of the features are reported in their corresponding referenced works.

The first two conditions aim to guarantee that our approach can be easily reproduced by other researchers. The third condition provides a ground truth to evaluate the naming process given that the result of the manual naming was reported in their respective publications.

We further take care of selecting case studies that differ with regards to their artefact types (source code, modeling languages and plugin-based software). Table 8.1 characterizes the case studies (CS). First, source code-based case studies are software systems written in Java. We show the number of source code compilation units (i.e., Classes), the number of methods, and the number of source lines of code (LOC)ⁱⁱ. For the different case studies regarding modeling languages, we show the number of Classes that each model variant has. The term Classes should not to be confused strictly with UML Classes as we explained in Section 5.3.1. In fact, only CS 7 is a case study with UML model variants. Attributes (Attrs) and References (Refs) are the number of non-null properties of the classes. Finally, we show the characteristics of variants of Eclipse, an IDE known for its plugin-based architecture. Concretely, we focus on variants simultaneously released in Kepler which were already used in Section 4.4.1.

We leverage the source code and model case studies (CS 1 to CS 7) to evaluate RQ-1 providing quantitatively results using the ground truth. The Eclipse case study (CS 8), on the other hand, is used for RQ-2 providing time measurements and qualitative results with the involvement of domain experts.

ⁱⁱLines which contain characters other than white space and comments using Google CodePro

Table 8.1: Characteristics of the case studies for the VariClouds approach separated by artefact types (source code, models and component-based systems). For each case study, it is included the reference works in the literature and an enumeration of the variants.

Source code case studies				Model case studies			
	Classes	Methods	LOC		Classes	Attrs	Refs
CS 1: ArgoUML [CVF11a, ASH ⁺ 13c, ZFdSZ12]				CS 5: In-Flight Entertainment Systems [MZB ⁺ 15a], Section 5.4.3: <i>Capella models</i> [Pol15]			
Original	1,773	14,904	148,715	Original	5,345	4,081	7,198
No cognitive support	1,553	13,567	132,397	Low Cost 1	5,321	4,066	7,164
No activity diagram	1,756	14,710	146,428	Low Cost 2	5,335	4,075	7,184
No state diagram	1,738	14,508	144,799				
No collab. diagram	1,754	14,783	147,137	CS 6: Vending Machine Statecharts [IBK11, MZKT14], Section 8.3: <i>SCT models</i> [Ite14]			
No sequence diagram	1,720	14,483	143,337	Variant 1	18	17	15
No use case diagram	1,736	14,684	146,002	Variant 2	22	21	19
No deploy. diagram	1,740	14,667	145,569	Variant 3	18	17	15
No logging	1,773	14,901	146,206	Variant 4	23	22	20
All disabled	1,357	11,934	110,933	Variant 5	18	17	15
				Variant 6	21	20	18
CS 2: Notepad [TKB ⁺ 14, ZHP ⁺ 14]				CS 7: Banking Systems [MZB ⁺ 15a, MZKT14], Section 5.2: <i>UML models</i>			
Notepad 1	7	44	600	Bank 1	62	60	28
Notepad 2	7	47	652	Bank 2	63	62	29
Notepad 3	7	46	643	Bank 3	47	47	19
Notepad 4	7	49	695				
Notepad 5	9	50	689				
Notepad 6	9	53	740				
Notepad 7	9	52	730				
Notepad 8	9	55	782				
CS 3: Draw application [FLE14, LLE16]				Component-based case study			
						Plugins	
P1	3	25	134	Eclipse Kepler packages [MZB ⁺ 15b], Section 4.4.1			
P2	3	28	174		Standard		450
P3	4	36	219		Java		416
P4	3	26	170		Java EE		831
P5	3	26	147		C/C++		387
P6	3	27	183		Scout		734
P7	3	29	210		Java and DSL		715
P8	3	29	187		Modeling Tools		888
P9	3	30	223		RCP and RAP		619
P10	4	37	257		Testing		317
P11	4	37	233		Java and Reporting		892
P12	4	38	271		Parallel Applications		546
					Automotive Software		518
CS 4: Mobile media [YM05, FCS ⁺ 08]							
MobileMedia R1	15	92	831				
MobileMedia R2	24	124	1,159				
MobileMedia R3	25	140	1,314				
MobileMedia R4	25	143	1,363				
MobileMedia R5	30	160	1,555				
MobileMedia R6	37	200	2,051				
MobileMedia R7	46	239	2,523				
MobileMedia R8	50	271	3,016				

8.4.2 Quality of the word clouds

We use the Mean Reciprocal Rank (MRR) [Voo99] for the evaluation. MRR is a metric used in IR to measure the quality of rankings. Concretely, it captures how early the relevant result appears in the ranking. It is also considered that MRR measures users' effort as it is related to their search length. This is the case of using word clouds where the user look at the largest name, then to the second largest, then to the third and so on. MRR is used in the cases that there is only one relevant solution as it is the case of known-item search (the feature name of the ground truth). The reciprocal rank (RR) of a given feature name in its associated block is calculated as $1/rank_i$ where $rank_i$ is the position in the ranking where the feature name appears. Let F be the features that we want to search, the formula of MRR is presented in Equation 8.2. For example, a MRR of 1 means that all feature names were the largest in the word cloud of its associated block.

$$MRR = \frac{1}{|F|} \sum_{i=1}^{|F|} \frac{1}{rank_i} \quad (8.2)$$

In the worst case scenario the name is not found in any rank position and therefore the denominator of RR is 0. In this special case we consider $1/rank_i = 0$ as if $rank_i$ was tending to infinity. However, given the importance of this fact we will discuss these cases separately. We have observed that the core components common to all variants use to be encompassed in a feature whose names are *Core* or *Base*. These names do not use to be part of the emerging vocabulary and for this reason, we refer to them as MRR2, the MRR metric where the *Core* or *Base* feature is excluded from the set F .

Table 8.2 shows the results in terms of MRR2, MRR and the rank of each of the features. If a name is not found we denote it with the empty set symbol \emptyset . *MRank* shows the average of the rank of the features without considering the core feature and the features that were not found. For the calculation of the *rank* of some features, we were slightly flexible in manually deciding if two words were the same. We present an exhaustive list of these cases

Table 8.2: Evaluation of the quality of the word clouds.

	MRR2	MRR	MRank	Rank of each feature, \emptyset for not found
<i>CS 1</i>	0.71	0.63	1.57	<i>Core</i> (\emptyset), Logging (1), Activity diagram (1), State diagram (1), Collaboration diagram (1), Sequence diagram (4), Use case diagram (1), Deployment diagram (2), Cognitive support (\emptyset)
<i>CS 2</i>	0.83	0.62	1.33	<i>Base</i> (\emptyset), Cut-Copy-Paste (1), Find (1), Undo-Redo (2)
<i>CS 3</i>	1.00	0.80	1.00	<i>Base</i> (\emptyset), Line(1), Rect (1), Color (1), Wipe (1)
<i>CS 4</i>	0.63	0.57	3.33	<i>Core</i> (\emptyset), ExceptionHandling (1), LabelMedia (2), Sorting (6), Favourites (1), Photo (2), Music (2), Video (1), Sms (1), CopyMedia (14)
<i>CS 5</i>	1.00	0.66	1.00	<i>Core</i> (\emptyset), Wi-Fi (1), ExteriorVideo (1)
<i>CS 6</i>	0.76	0.69	1.83	<i>Main</i> (4), Soda (1), Coffee (1), Tea (1), Cash payment (4), Credit card payment (3), Ring tone alert (1)
<i>CS 7</i>	0.62	0.50	1.33	<i>BankCore</i> (\emptyset), CurrencyConverter (2), WithdrawWithLimit (1), Consortium (1), WithdrawWithoutLimit (\emptyset)
<i>Mean:</i>	<i>0.79</i>	<i>0.63</i>	<i>1.62</i>	

that can be debatable. We considered *log* similar to *logging*, *cash* similar to *cash payment*, *credit card* similar to *credit card payment*, *ring tone* similar to *ring tone alert*, *exterior-view* similar to *ExteriorVideo*, *Exception* similar to *ExceptionHandling*, *Label* similar to *LabelMedia*, *Favourites* similar to *Favorite* and for the names of the UML diagrams we did not consider the word *diagram* (e.g., *Activity diagram* similar to *Activity*).

The mean of MRank is 1.62 which indicates that, ignoring the core features and the not found features, the feature names appear in the first two positions of the ranking. The mean of MRR2 in this set of case studies is 0.79. This result is promising and, therefore, guarantee the soundness of the VariClouds which will show the largest words for the most relevant terms thus reducing users' search effort. Despite the promising average results, there are some unsuccessful results that cannot be neglected. We discuss the two main reasons for unsuccessful RR results with a special focus on the two not-found feature names ($rank = \emptyset$).

- **Mismatch between domain names and implementation details:** In the case of the *Cognitive support* feature in ArgoUML, there is a complete mismatch between the feature name and the vocabulary emerging from the implementation. This is the worst case for name suggestion during feature identification using VariClouds. The largest words of its associated blocks are *cr*, *criticized* and *design*. Specifically, *criticized* corresponds to the Critics subsystem of the ArgoUML architecture [CVF11a] which implements the *Cognitive support*. In the ArgoUML SPL websiteⁱⁱⁱ this feature is called *design critics*. The reference work [CVF11a] took the feature name from the publication that explained this functionality [RR00].
- **Filters undesired effect:** In the case of *WithdrawWithoutLimit* of the Banking systems, *with* and *without*, despite being prepositions, were important words which were discarded by the parts-of-speech tag remover. By deactivating this filter, MRR2 can be 0.875 if we consider *without* similar to *WithdrawWithoutLimit* (*withdraw* and *limit* are the rank items following the first one that is *without*). In the preparation phase, the camel case splitter was activated for all the source code based case studies, as well as for the UML model variants of the Banking systems given that UML model vocabulary was close to source code language. However, in the case of the Notepad case study, camel case is not the style followed by the developers. There are methods called *fnD* or field declarations called *findNext* therefore the largest words for the Find feature are *find*, *fin* and *d*. This fact complicates the readability of the word cloud but luckily it does not affect the RR in this case.

ⁱⁱⁱArgoUML SPL website: <http://argouml-spl.stage.tigris.org/>

8.4.3 The benefits of summarization

Textual representations were used in previous works to characterize, separately, each implementation element. In small artefacts, or for illustrative purposes, this can be useful but it does not scale for human comprehension. In blocks with thousands of elements, a summarization approach might avoid the domain experts to spend time, during naming, looking at the technicalities of the artefacts' implementation, or trying to reason using the textual representation of individual elements.

A small excerpt of the textual representation of model elements was presented in Figure 5.2. Other examples can be found in the literature of extractive SPL adoption. For instance, AL-msie'deen *et al.* [ASH⁺13b] refer to a Java class named `ImagePath` in the package `Drawing.Shapes.Image` using the textual representation `Class(ImagePath_Drawing.Shapes.Image)`. The same author [ASH⁺13c], referring to a Java method named `DatabaseState` in the `Database` class, used `Method(DatabaseState())_Database`. Ziadi *et al.* use a textual notation where each element is represented as a construction primitive [ZHP⁺14]. For example, `CreateTerminal(deposit, method, Account)` refers to add, in the AST, a method named `deposit` in a class named `Account`.

This subsection presents our case study to analyse the benefits of summarization with VariClouds instead of using plain textual representation of individual elements. Before designing and implementing VariClouds, we considered the scenario of Eclipse variants presented in Section 4.4. In those experiments, we requested the expertise of three domain experts with more than ten years of experience on Eclipse development, who analysed, independently from each other, the Elements' textual representations of the 61 Blocks that were identified. This manual task took an average of 51 minutes which provides us a baseline.

For the evaluation of RQ-2 we consider the Eclipse case study again. We evaluated VariClouds with another three domain experts on Eclipse which were not the same persons as in the previous experiment. However, they have similar professional expertise in Eclipse development as the previous ones. We asked them, independently, to perform the feature identification tasks using the word cloud visualisation as support for the element textual representations of each block. In addition, they were asked to think aloud and report their mental process to select the names. During the experiment all the domain experts stated the following process for block naming:

1. Read very quickly the textual descriptions of the elements (beginning, middle and end) in order to have an initial clue about the logic and identify a word in their mind.
2. Read the word cloud largest names and contrast them with the one in their mind.
3. Select one from the word cloud or use the guessed one.
4. Optionally refine the selected word with an extra word.

The average time was reduced from 51 minutes to 28 minutes. Even if 23 minutes may not seem a very relevant reduction, it constitutes a 45% decrease in this case study. Qualitatively, all of them stated that the word clouds were useful for assigning the names. This was emphasized when they were not completely sure about the logic of the block. They stated

that the presence of the word clouds served as reinforcement or confirmation for the final naming decision. According to the time reduction and their mental process, we can say that word clouds reduce domain experts' comprehension time and help them to be more confident with the naming decisions while accelerating the process.

8.5 Threats to validity

VariClouds is a visualisation paradigm that makes use of well established IR techniques. We provide empirical results about its soundness. We found that the results on the case studies are promising but we cannot assure that the findings can be generalized. However, as discussed in Section 3.1.2, in the past, the efforts were mainly focused on the technicalities of innovative and promising block identification techniques rather than on support for final users. VariClouds fills the gap of a domain experts' process that had a very rudimentary support.

Other threats to validity regarding the generalization of the findings is that the feature names of the ground truth obtained from the SPL literature are conditioned to human factors. Some domain experts decided to put the names we use as ground truth. Also, human factors determine the match between this ground truth and the words used by the developers of the artefacts' implementation. All the case studies consider variants that belongs to the same developers or providers. In the case of completely independently developed artefact variants, the emerging vocabulary from the variants may use a completely different terminology in a way that the synonym filter or other more advanced filters may not be able to correlate.

VariClouds claims for generality in supporting different artefact types, however it assumes the existence of an adapter as presented in Section 8.2.2. In addition, for some reason or for implementation details, some artefact types can have the limitation that their elements may lack meaningful names (e.g., compiled or obfuscated artefacts). In the same way, VariClouds assumes the existence of a block identification technique. As presented in Section 4.3.2, BUT4Reuse provides a set of them, however, it is accepted that there are many factors affecting the quality of the results of these techniques such as the number of variants, their diversity, the match and diff policies, the number of features or the presence of feature interactions among others [RC13a, ZHP⁺14, FLLE14]. We agree that the results of VariClouds strongly depend on the block identification technique, however, the research conducted to propose VariClouds is complementary as it is focused in the interaction and visualisation paradigm for the domain experts. Therefore the two research directions can evolve in parallel.

Finally, as threat to validity regarding the Eclipse case study, even if we consider that the new domain experts have a very similar background to the previous domain experts, we cannot assure that the difference is because they have a different set of skills.

8.6 Conclusion

We contributed VariClouds as an approach that extensively uses word cloud visualisations in order to provide insights of the emerging vocabulary and variability from a set of variants. Specifically it is designed for helping domain experts in feature identification and naming. We evaluated it in several case studies dealing with different artefact types to show its soundness and genericness.

As further work we aim to evaluate the use of different weights for different Element types. For example, in source code, we can have the hypothesis that the words that belong to a class name have more relevance than the words belonging to a method name. From a visualisation perspective, rather than alphabetically ordering the words, we aim to evaluate the use of more structured clouds, such as tree clouds or force-directed graph drawing. The adapter provides information about element dependencies that can help in creating these structured clouds. We also aim to design other advanced word cloud filters where domain ontologies could be leveraged.

FEATURE CONSTRAINTS ANALYSIS WITH FEATURE RELATIONS GRAPHS

This chapter is based on the work that has been published in the following paper:

- Jabier Martinez, Tewfik Ziadi, Raúl Mazo, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Feature relations graphs: A visualisation paradigm for feature constraints in software product lines. In *Second IEEE Working Conference on Software Visualization, VISSOFT 2014, Victoria, BC, Canada, September 29-30, 2014*, pages 50–59, 2014

Contents

9.1	Introduction	134
9.2	Manual feature constraints discovery	135
9.3	Mining configurations to create feature relations graphs	136
9.3.1	Data abstraction and formulas	136
9.3.2	Graphical aspects	138
9.3.3	Interaction possibilities	142
9.3.4	Rejected alternatives	143
9.4	Electric Parking Brake case study	145
9.4.1	Introduction to the domain	145
9.4.2	Visualising the feature relations	146
9.4.3	Extension considering feature conditions	149
9.5	Conclusions	149

9.1 Introduction

Feature constraints discovery is an important activity in extractive SPL adoption. As described in Section 2.2.2, once the features are identified, domain experts still need to reason about the constraints that may exist among the features. In Section 3.1.4 we presented related work on automatic techniques for this activity and Section 4.3.5 detailed the integrated techniques in BUT4Reuse. However, beyond automatic techniques, domain experts require visualisation paradigms to assist during this activity enabling free exploration for constraints discovery. Unfortunately, there is a:

Lack of support for manual feature constraints discovery

- *It is complex to understand how features are related among them.* As we described in Section 3.3, existing visualisation paradigms in SPLE have not focused enough on solutions to reason about feature constraints and about the existing relations among the features, especially when the objective is to discover missing constraints (hereafter *non-formalized* constraints).
- *Soft constraints discovery should be part of the process.* Soft constraints, presented in Section 2.1.2, are relevant domain information, therefore, domain experts should have the means to formalize this knowledge as part of the discovery process.
- *Stakeholders belonging to different fields of expertise complicate the process.* Feature constraints usually exist among features belonging to different fields of expertise, thus, the stakeholders may not have an overview about how the features relevant to them are related to the others. In these cases, it is challenging to effectively communicate and visualise the constraints.

Contributions of this chapter.

- **The Feature Relations Graphs (FRoGs) visualisation paradigm** to represent the relations among features. For each feature, we are able to display a FRoG which shows the impact, in terms of constraints, of the considered feature on all the other features. The visualisation can be considered as a specialized *radial ego network* where both stakeholder diversity and soft constraints are taken into account.

It is also worth mentioning that, the usage of the FRoGs visualisation is not necessarily restricted to the context of extractive SPL adoption. It can be used in documentation, feature model maintenance or recommendation during product configuration.

The remainder of this chapter is structured as follows: Section 9.2 presents an overview of manual feature constraints discovery. Section 9.3 formalizes the data abstractions and describes our proposed visualisation paradigm. Then, Section 9.4 presents the case study and, finally, Section 9.5 concludes and outline future work.

9.2 Manual feature constraints discovery

FROGs is a visualisation for domain experts to discover constraints during extractive SPL adoption based on mining existing product configurations. Other constraints discovery techniques, such as the structural constraints discovery (Section 4.3.5), automatically analyze the structural dependencies among the elements of a feature. However, FROGs only considers the information from the existing configurations that can be calculated after the feature identification and location activities. Figure 9.1 illustrates how the configurations are obtained. Once the features are known and located, it is possible to identify, for each variant, the features that it contains. In the example of the figure, at the bottom, we can see that the elements associated to F1 and F2 features are those present in **Variant 1**, therefore, **Configuration 1** consists of these two features.

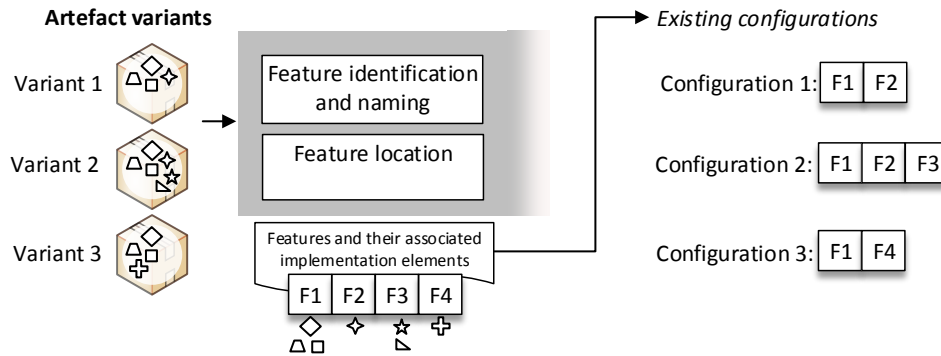


Figure 9.1: Existing configurations obtained as part of the extractive SPL adoption process.

FROGs visualisation enables domain experts to formulate questions regarding a given feature. Continuing with the example of Figure 9.1 and focusing on F3: Given the fact that F3 always appears with F2 in the existing configurations, can we affirm that F3 *requires* F2? The fact that F3 never appears with F4 is because F3 *excludes* F4? or is it just a coincidence and nothing prevents F3 to be with F4? In this example we are not using meaningful feature names but, with meaningful names, domain experts could be able to discover constraints based on their domain knowledge. FROGs visualisation also aims to trigger other kind of questions regarding tendencies found in the existing configurations. This is related to enabling the discovery and formalization of soft constraints.

Given that variability in SPLs arises from different stakeholders' goals [YdPLLM08], in general, constraints are formalized by different stakeholders. Each stakeholder can thus be interested in a partial view of the variability (*stakeholder perspective*). FROGs visualisation aims to facilitate this separation while enabling, at the same time, to reason about potential constraints found across stakeholder perspectives.

FROGs visualisation can be categorized as a *radial ego network* representation which is used in social network analysis. The ego represents the focal node of the graph. In software engineering, ego networks have been used to visualise how the modification of a component could demand the modification of other components [DLL06]. FROGs, for its part, is designed to deal with the specificities of feature constraints in SPLE.

9.3 Mining configurations to create feature relations graphs

In this section we present how the graphs of the FRoGs visualisation are created and how they are used. Concretely, we present the data used to create the visualisation paradigm and how this data is graphically represented. We also describe the interaction possibilities. Finally, we discuss rejected alternatives for a better understanding of our design decisions.

Apart from extractive SPL adoption, FRoGs can be also used for constraints discovery in an operative SPL to discover non-formalized constraints in the FM. The principles of the visualisation are the same, so in next sections we explain FRoGs as a general solution for constraints discovery in SPLE.

9.3.1 Data abstraction and formulas

A configuration is defined as a set of selected features $c_i = \{f_1, f_2, \dots, f_n\}$. Let $S = \{c_1, c_2, \dots, c_m\}$ be the configuration space, i.e., a set with all the possible valid configurations for a given FM. In the case of extractive SPL adoption, if no previous automatic constraints discovery techniques are applied, the FM will consist on independent optional features. Let $EC \subseteq S$ be the set of existing configurations. Our approach assumes that all configurations in the EC set are valid. Also, if duplicated configurations exist, our approach does not take them into account. Indeed, one configuration represents one product variant independently of the amount of units of the product variant that we may have.

As example, and following with the Car example [CSW08] presented in Section 2.1.2, Table 9.1 presents the S set. The Car FM was shown in Figure 2.5 containing the constraints that define the boundaries of S .

Table 9.1: Configuration space for the Car example.

	Manual f_1	Automatic f_2	DriveByWire f_3	ForNorthAmerica f_4
Conf 1		✓		✓
Conf 2	✓			
Conf 3		✓		
Conf 4	✓			✓
Conf 5		✓	✓	✓
Conf 6		✓	✓	

Let consider that EC equals S excluding the last configuration *Conf 6*. In this case EC is expressed as:

$$EC = \{ \begin{array}{l} c_1 = \{ \quad \quad f_2, \quad \quad f_4 \}, \\ c_2 = \{ f_1, \quad \quad \quad \quad \}, \\ c_3 = \{ \quad \quad f_2, \quad \quad \quad \}, \\ c_4 = \{ f_1, \quad \quad \quad f_4 \}, \\ c_5 = \{ \quad \quad f_2, \quad f_3, \quad f_4 \} \end{array} \}$$

Given EC , we can reason on the feature relations in this set. First, let $EC_{f_i} = \{c \in EC : f_i \in c\}$ be the subset of existing configurations that contain the feature f_i . For example, the existing configurations containing the **Manual** feature is defined as $EC_{f_1} = \{c_2, c_4\}$. With this definition of EC_{f_i} , we can calculate the ratio of the occurrence of a feature given the occurrence of another feature. Following existing notation [CSW08] we call this operation f_i given f_j . Equation 9.1 presents the formula which is similar to conditional probability.

$$f_i \text{ given } f_j = \frac{|EC_{f_i} \cap EC_{f_j}|}{|EC_{f_j}|} \quad (9.1)$$

In the Car example, the proportion of **Automatic** given **DriveByWire** (f_2 given f_3) equals 1 since when we have **DriveByWire** in the EC , we always have **Automatic**. In the same way, **Manual** given **Automatic** (f_1 given f_2) equals 0 because when we have **Automatic** we never have **Manual**. Equation 9.2 details how **Automatic** given **ForNorthAmerica** (f_2 given f_4) is calculated to obtain 0.66.

$$f_2 \text{ given } f_4 = \frac{|EC_{f_2} \cap EC_{f_4}|}{|EC_{f_4}|} = \frac{|(c1, c3, c5) \cap (c1, c4, c5)|}{|(c1, c4, c5)|} = \frac{|(c1, c5)|}{|(c1, c4, c5)|} = \frac{2}{3} \approx 0.66 \quad (9.2)$$

These ratios, that are continuous values ranging from zero to one, are mapped to the different potential hard and soft constraints as described in Table 9.2. For example, **Automatic** given **DriveByWire** = 1 is mapped to **DriveByWire** requires **Automatic**. Regarding soft constraints, the thresholds $enc_{threshold}$ and $dis_{threshold}$ are defined and adjusted by domain experts based on their experience. These thresholds are also discussed and set by domain experts in previous related works [CSW08].

Table 9.2: Constraints identification while using feature relations graphs

Condition	Constraint	Notation
$f_j \text{ given } f_i = 1$	f_i requires f_j	$f_i \Rightarrow f_j$
$f_j \text{ given } f_i = 0$	f_i excludes f_j	$f_i \Rightarrow \neg f_j$
$enc_{threshold} \leq f_j \text{ given } f_i < 1$	f_i encourages f_j	$soft(f_i \Rightarrow f_j)$
$0 < f_j \text{ given } f_i \leq dis_{threshold}$	f_i discourages f_j	$soft(f_i \Rightarrow \neg f_j)$

We define a confidence metric for the validity of the mined constraints. The confidence metric is only applicable for relations that are not explicitly formalized in the FM. The intuition of our confidence metric is to calculate the probability of finding one configuration that violates this constraint. Let S_{f_i} be the subset of the valid configurations that contains the feature f_i . $S_{f_i} = \{c \in S : f_i \in c\}$. Equation 9.3 shows how the metric is calculated.

$$Confidence(f_i) = \frac{|EC_{f_i}|}{|S_{f_i}|} \quad (9.3)$$

For example, if we have f_j given $f_i = 1$, the confidence of $f_i \Rightarrow f_j$ is calculated as the percentage of the possible valid configurations containing f_i that exist in EC . The confidence will increase when new configurations containing f_i are created as we will reduce the probability of finding a configuration contradicting $f_i \Rightarrow f_j$. The confidence will reach one when $EC_{f_i} = S_{f_i}$. Calculating the number of valid configurations is a well known problem in automated analysis of FMs [BTRC05]. Given that S_{f_i} can be prohibitively large, the cardinality $|S_{f_i}|$ is calculated reasoning on the formalized feature constraints without the need of an exhaustive enumeration of the set S_{f_i} .

Regarding the stakeholder perspectives, each of them consists in the subset of features that are “owned” by a stakeholder. These subsets must be disjoint in the current version. Stakeholder perspectives can represent conceptual units in SPLs such as *domains* or *subdomains* [BFK⁺99]. Once these conceptual units are identified, the features are distributed in these subdomains [JKLM06]. Following with the Car example, features related to the type of **Gear** are mainly relevant for final *Customers* to choose their cars. However, **DriveByWire** is more the concern of *Engineers* during the construction of the vehicle and the **ForNorthAmerica** feature is relevant for *Commercials* which are interested in sales analysis. Each of these stakeholder perspectives cannot ignore how features belonging to other stakeholders impact their own features.

9.3.2 Graphical aspects

This subsection presents design decisions that are relevant from a visualisation point of view. A FRoG is presented as a circle where the considered feature is displayed in the center. This feature in the center will be called f_c hereafter. The rest of features are displayed around f_c with a constant separation of $\frac{2\pi}{|features|-1}$. This separation allows to uniformly distribute all the features (except f_c) around the circle.

The features are ordered by stakeholder perspectives and circular sectors are displayed for each of them. Before providing more details about the visualisation, Figure 9.2 shows an example of a FRoG for the **Manual** feature of the Car example after mining EC . Specifically, this FRoG shows the impact of **Manual** (f_c in the center) in the rest of the features. For example, **Manual** excludes **Automatic** and **DriveByWire**. We can also see the sectors of the *Customer*, *Engineer* and *Commercial* stakeholders covering their corresponding features.

The zones of a FRoG

Without considering the stakeholder sectors, each FRoG displays five differentiated zones. These zones are associated with a specific color and related to a specific type of constraint. Figure 9.3 shows these zones and how the distance between f_c and the boundary of the circle is calculated according to the f_i given f_c operation.

- *Requires Zone*. This zone is the closest to f_c . Features in the requires zone are “attached” to f_c meaning that, in EC , when we have f_c we always have f_i . The requires zone is reserved to the maximum value of f_i given f_c which is 1. The requires zone, as well as the excludes zone that we will present later, are reserved to only one value and their

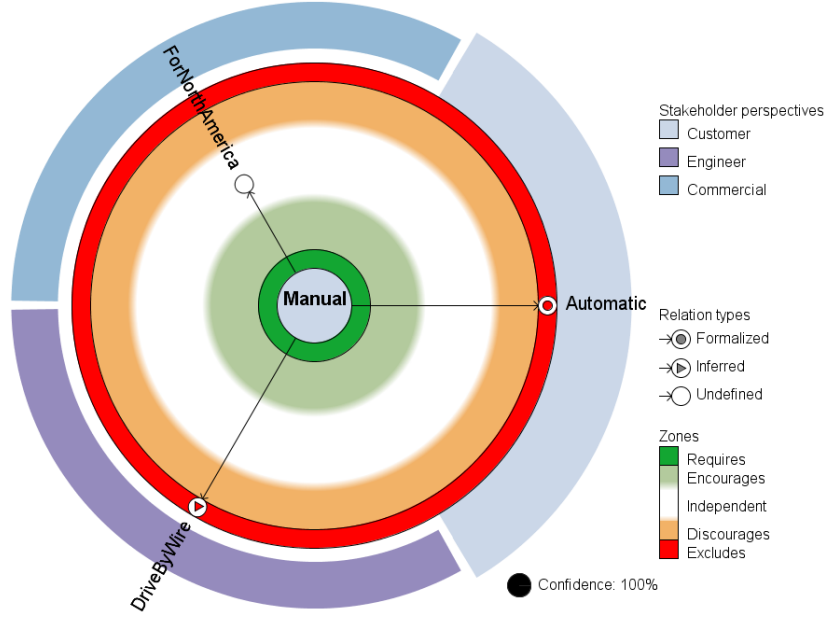


Figure 9.2: Feature relations graph for the Manual feature of the Car example. Manual is in the center. The Automatic and DriveByWire features are in the excludes zone. The excludes constraints between Manual and Automatic are formalized in the FM, and the excludes constraint with DriveByWire is inferred from other constraints. DriveByWire and ForNorthAmerica belong to stakeholders that are not the same as the one from Manual. Manual seems not to have any relation with ForNorthAmerica. The confidence of the constraint Manual excludes Automatic is 100% because, in the existing configurations, we already have all the possible configurations where Manual can be selected.

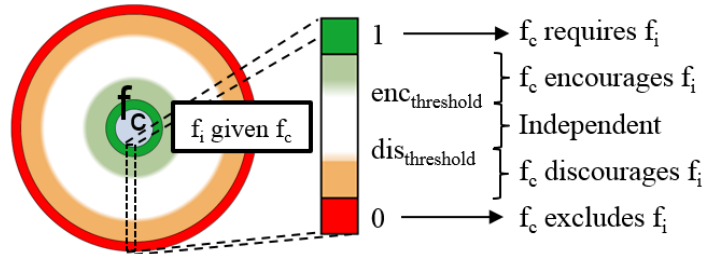


Figure 9.3: Zones of a feature relations graph.

size fits exactly with the diameter of the f_i nodes. The requires zone is displayed with the green color.

- *Excludes Zone.* This zone, displayed with the red color, is in the extreme of the circle. It is the furthest from f_c to illustrate that there is no occurrence of f_c and f_i together in any configuration.
- *Encourages Zone.* This zone includes all f_i that are potentially encouraged by the presence of f_c . The color is a pale green meaning that it is close to the requires zone without reaching it. The encourages zone fades to white in the independent zone that we will present later. The fading occurs at the distance defined by the $enc_{threshold}$.
- *Discourages Zone.* This zone includes all f_i that are potentially discouraged by the presence of f_c . The color is orange meaning that it is close to the excludes zone without reaching it. The discourages zone fades to white in the independent zone at the distance defined by the $dis_{threshold}$.

- *Independent Zone.* This zone, located between the *enc_{threshold}* and the *dis_{threshold}*, is displayed with the white color containing the features that are not impacted by the presence of f_c .

In the FRoG of Figure 9.2, where **Manual** was the f_c , we can observe how both **Automatic** and **DriveByWire** are in the excludes zone. The feature **ForNorthAmerica**, with a f_i given f_c of 0.5, is located in the middle between the requires and excludes zones. In this case, the feature is in the independent zone.

Comments on colors

The usage of colors in a visualisation is an important design decision [SSM11]. FRoG zones are depending on univariate data (f_i given f_c). This data has continuous values. However, as we have shown, these values are mapped to zones in a discrete fashion. Therefore, we have defined boundaries and we associated one color to each of the zones following the color scheme shown before in Figure 9.3.

The green of the requires zone contrasts heavily with the red of the excludes zone. The *separation principle*, that claims that close values must be represented by colors perceived to be closer, is respected with the green and the pale green for the requires and encourages zones. The same principle is respected in the red and orange colors associated to the excludes and discourages zones. The color scheme used in the zones is a *diverging color scheme* given that it illustrates the progression from a central point. The central point is when f_i given f_c equals 0.5. In addition, the traffic lights metaphor, shared by most of the cultural contexts, is used in FRoGs visualisation. The red color for the excludes zone has a connotation of prohibition (f_i cannot be with f_c), while the green of the requires zone has a connotation of acceptance (f_i must be with f_c).

The size of the requires and excludes zones fits exactly with the size of a f_i node. For the other zones, the size depends on the percentage defined in the *enc_{threshold}* and *dis_{threshold}*. The fading zone between the encourages and discourages zones is very small in size. This is aimed to create a visual effect to recall that these boundaries are not as restrictive as the boundaries of the hard constraints represented with a black line. For example, a feature f_i with a formalized excludes hard constraint cannot appear in the discourages zone. Otherwise, it would not be a hard constraint. On the contrary, one feature f_i that has not a hard constraint could “move” over time between the encourages, independent and discourages zone when new configurations are added.

Differentiating formalized, non-formalized and inferred constraints

A FRoG displays the mined constraints by analysing the *EC*. However, some of them (or all in an ideal case) are normally already defined in the FM. In the FRoGs visualisation, for already formalized constraints, we add an inner circle in the node of f_i with the color associated to the type of constraint. Figure 9.4 shows this notation. For example, *Manual* \Rightarrow \neg *Automatic* is an already formalized constraint as shown in the Car FM in Figure 2.5. Therefore, in the FRoG of Figure 9.2, this notation appears in the **Automatic** node. Concretely, it uses the inner circle in red color because it is an excludes constraint.

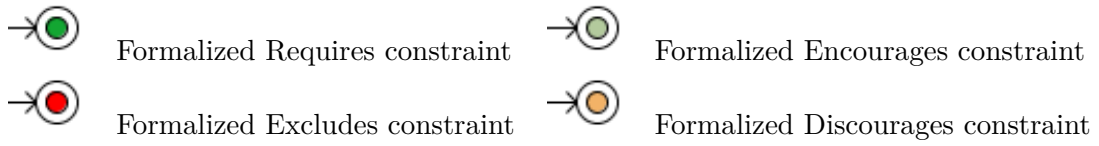


Figure 9.4: Graphical notation for formalized constraints.

It is worth noting that, in the case of formalized soft constraints, the mining process on *EC* could situate f_i in a FRoG zone not corresponding to the defined soft constraint. This could alert the user about a general violation of the formalized soft constraint.

An inferred constraint, is a constraint that is not formalized in the FM but that exists because of logical rules. For example, when $A \Rightarrow B$ and $B \Rightarrow C$, it can be inferred that $A \Rightarrow C$ even if it is not explicitly formalized. Figure 9.2 shows an example of inferred constraint in the *DriveByWire* node. In fact, *Manual* excludes *Automatic*, and *DriveByWire* requires *Automatic*. Given this situation, *Manual* excludes *DriveByWire*.

Figure 9.2 shows the FRoG legend at the right side of the image. In this legend, the “Relation types” category shows the notation for the feature relations. We use the triangle for inferred constraints to differentiate it from the circle of the formalized ones. The triangle metaphor contrasts with the circle and it has the connotation of an arrowhead representing the existence of a rule of inference. If it is neither a formalized nor an inferred constraint, we will refer to it as an undefined relation independently of the zone where the f_i is placed. Undefined relations do not contain any symbol inside the f_i node.

Stakeholder Perspectives

Each stakeholder perspective has an associated color used in the circular sectors of the FRoG. The f_c node has the color of the stakeholder perspective it belongs to. In addition, the circular sector of the stakeholder perspective of f_c has a slightly larger radius. Figure 9.2 showed how the *Manual* feature has the *Customer* color and that the *Customer* sector has larger radius.

Stakeholder perspectives are nominal data and we use a qualitative color scale that does not imply order. Despite that default colors are provided, the stakeholder perspective colors could be changed by users. Figure 9.5 shows the used color scheme. In the rejected alternatives section 9.3.4, we discuss why only grey, purple and blue variations are present in this default color scheme to avoid interfering with the colors from the zones.



Figure 9.5: Default color scheme for the nominal data of stakeholder perspectives.

Displaying the Confidence

The confidence is displayed outside of the FRoG zones. It is presented as a percentage accompanied by a pie chart visualising this percentage. In the bottom-right side of Figure 9.2 we can see a confidence of 100% with a complete black circle. Figure 9.6 shows another example with a confidence value of 5% showing a pie chart with only 5% in black.

9.3.3 Interaction possibilities

FROGs is a visualisation tool allowing free exploration. The navigation and filter capabilities are presented in the next paragraphs focusing on details of our implementation. To illustrate these interaction possibilities, Figure 9.6 shows a screenshot of the tool at which we added dashed sections to highlight the different parts. The “Feature List” part contains all the variable features of the FM. The “FROG” part shows the FROG of the selected feature with the “Confidence” and “Legend” sub parts.

Navigation

On the left side of Figure 9.6 we can see the “Feature List” part. From this list we will be able to select the one that want to be analysed. As we can see in the figure, **PBSAutomatic** is the one selected and displayed in the center of the FROG. Apart from interacting with the list of features, we can interact with any feature node to show its FROG.

Filters

Filters can be applied simultaneously to hide feature nodes belonging to stakeholder perspectives, relation types or FROG zones. Different filter combinations could fulfil different objectives of FROGs usage. Some examples of usage are:

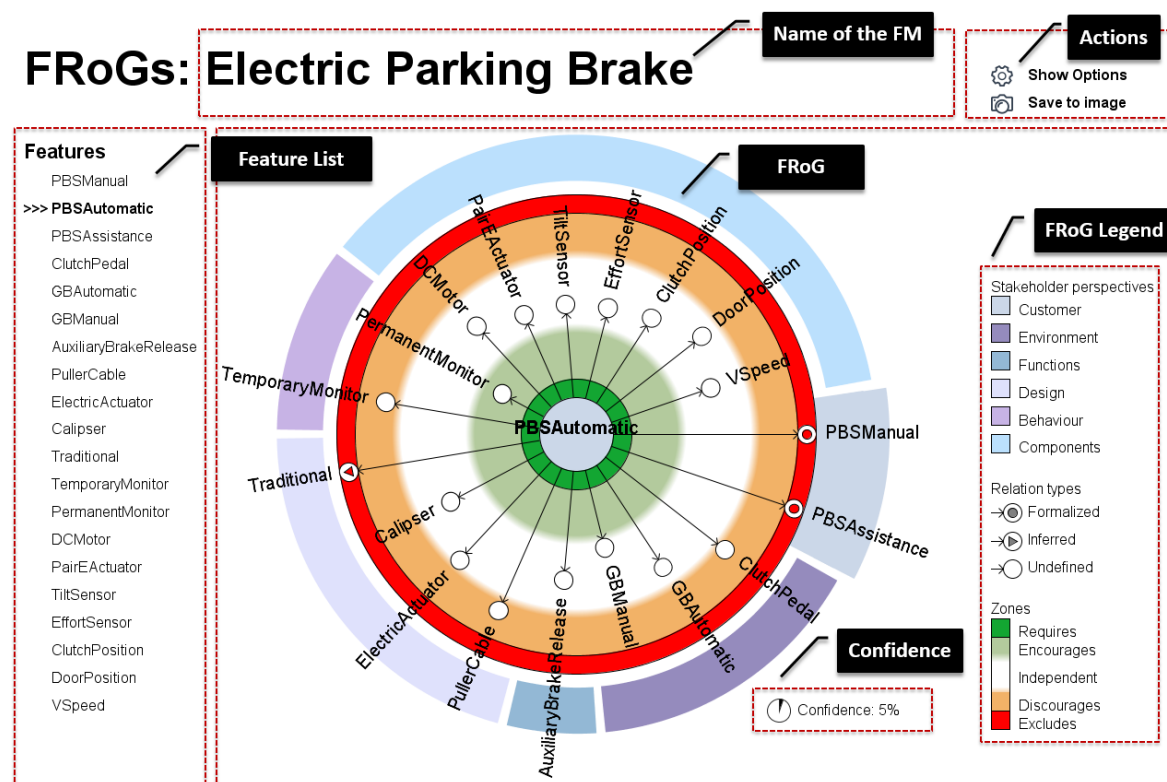


Figure 9.6: Screenshot of FROGs visualisation tool focusing on the PBSPAutomatic feature of the Electric Parking Brake SPL.

- *Discovery of non-formalized constraints:* We focus on potential non-formalized constraints by hiding all features on the independent zone and those constraints that are already formalized or inferred.
- *Documenting constraints:* We focus on constraints by hiding all features on the independent zone.
- *Intra-stakeholder perspectives constraints:* We hide all the features that do not correspond to a given stakeholder perspective.
- *Inter-stakeholder perspectives constraints:* We hide all the features that do not correspond to a given set of stakeholder perspectives.

Adjusting encourages and discourages thresholds

The $enc_{threshold}$ and $dis_{threshold}$ could be adjusted at will. An slider appears when selecting “Show Options” and, interacting with it, automatic feedback in the FRoG is provided by changing the size of the selected zone. If filters are applied in order to hide features in a given zone, hidden nodes could appear or disappear while modifying the thresholds.

Save as image

Another interaction capability of the visualisation tool is saving as image the current FRoG (filtered or not filtered) and the legend to an image file. This image can be used directly for documentation or to communicate some phenomena in feature relations.

Notes about the current implementation

FRoGs visualisation was implemented with Processing [CF12] and we achieved complete visual continuity. f_i given f_j is an algorithm with order $\mathcal{O}(m)$ where m is the number of existing configurations. Therefore, displaying a FRoG is $\mathcal{O}(nm)$ being n the number of features. The number of possible valid configurations is precomputed before starting the visualisation. The needed set of input files for FRoGs is exported using the SPL tool FeatureIDE [TKB⁺14] and the mapping from stakeholder perspectives to features are defined in a configuration file.

9.3.4 Rejected alternatives

During the design of FRoGs, we asked the opinion of the industrial partner as well as other users with knowledge on SPLE. We report in this subsection the reasons to reject some visualisation design alternatives.

Colors of Requires and Excludes zones

Initially, we used pale green and pale red for the requires and excludes zones. Users reported that more intense green and red colors for these zones are preferred to catch more attention on hard constraints.

Default colors for stakeholder perspectives

Qualitative scales for nominal data are publicly available as for example the color scheme provided by ColorBrewer 2.0 [Col]. Figure 9.7 shows this color scheme that we used in the beginning. However, users complained about the similarities between some of the colors of the stakeholder perspectives' sectors and the red of the excludes zone. For instance, the fourth color of Figure 9.7 was a cause of confusion. We decided to exclude colors similar to red, orange as well as similar to green and pale green.



Figure 9.7: Rejected color scheme for the nominal data of stakeholder perspectives.

Complex relation types

The current version of FRoGs only has the possibility to see the impact of the presence of f_c in the presence of the other features f_i . In order to display other phenomena in the relations of a given feature f_c , FRoGs implementation was flexible to show the impact of the presence or absence of f_c in the presence or absence of other features f_i . It was also possible to show the impact of the presence or absence of the rest of the features f_i in relation with f_c . These aspects gave raise to eight possible FRoGs for each feature. Figure 9.8 shows four of them that could be obtained from a given f_c . Current version of FRoGs visualisation corresponds to the first type of Figure 9.8.

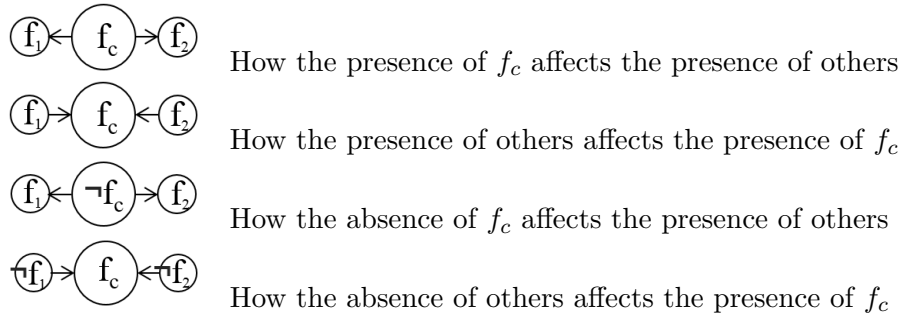


Figure 9.8: Different relations regarding presence or absence of features, and the direction of the relation regarding the FRoG central feature.

The other four combinations could also be displayed even if they are the “inverse” of one of the previous four types (f_i given $f_c = 1 - (\neg f_i)$ given f_c). Also, it is worth commenting that the direction of the arrows from f_c to f_i changes in types one and two, and three and four. To obtain the distances to show the impact of other features on f_c , it is enough to reverse the f_i given f_j operation: For type one is f_i given f_c while for type two is f_c given f_i .

This functionality in the options of the visualisation tool confused the users. It is not easy to think about presence and absence of features in combination with the direction of the arrows. Our case study did not need these kinds of relation visualisation for documentation nor for constraint discovery so we decided to remove it to simplify FRoGs usage.

9.4 Electric Parking Brake case study

We present FROGs evaluation on an industrial case study in the automotive industry concerning the Renault's Electric Parking Brake (EPB) SPL [DMSD13, MDSD14]. In this section, we will first describe the case study and subsequently discuss the results. We will focus on constraints discovery when an available FM might have non-formalized constraints.

9.4.1 Introduction to the domain

The EPB system is a variation of the classical, purely mechanical, parking brake, which ensures vehicle immobilization when the driver brings the vehicle to a full stop and leaves the vehicle. The case study is focused on the variability in the Bill of Materials (BOM). The BOM is related to the logical and physical components of the EPB. Figure 9.9 presents its FM containing 20 variable features. The possible valid configurations of this FM (S) amount to 2976 while the existing configurations that were provided (EC) are 200.

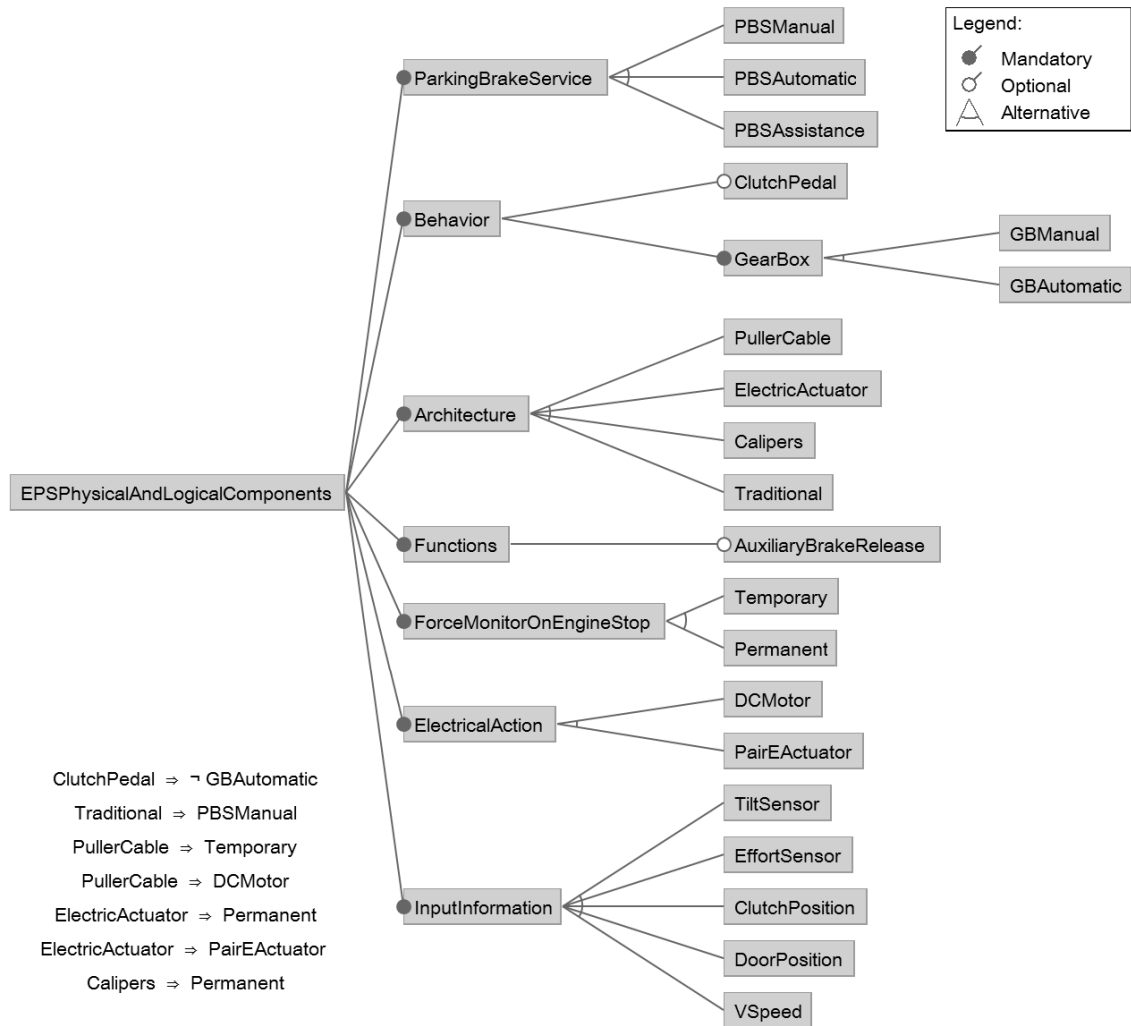


Figure 9.9: Feature model for physical and logical components of the Electric Parking Brake system.

The variability in this SPL is described by Renault according to six stakeholder perspectives that represent different subdomains of the EPB's variability. These stakeholder perspectives are *Customer*, *Environment*, *Functions*, *Design*, *Behavior* and *Components*. We present a brief description of these perspectives and their associated features:

Customer: Customer visible variability is handled by the product division. For customers, the parking brake proposes three types of services. The manual brake, implemented in the `PBSManual` feature, is controlled by the driver either through the classical lever or a switch. The automatic brake, `PBSAutomatic`, is a system that may enable or disable the brake itself depending on an automatic analysis of the situation. Finally, the assisted brake, `PBSAssisted`, comes with extra functions which aid the driver in other situations such as assistance when starting the car on a slope.

Environment: For vehicle environment, the variability concerns the gear box where we distinguish two possible variants: `GBManual` and `GBAutomatic`. The presence of a `ClutchPedal` is also an optional feature.

Functions: The feature `AuxiliaryBrakeRelease` is related to an optional function of the system that includes an auxiliary brake release mechanism.

Design: There are different alternatives concerning design decisions of the EPB impacting the technical solution. For architectural design, there are four feature alternatives: `PullerCable`, `ElectricActuator`, `Calipser` and `TraditionalPB`.

Behavior: After the vehicle has stopped, braking pressure is monitored during a certain amount of time for the single DC motor or permanently monitored for other solutions. The Behaviour includes thus the `TemporaryMonitor` and `PermanentMonitor` features.

Components: The EPB variability also concerns the physical architecture/components. This consists in the presence of different means of applying the brake force: electric actuators mounted on the calipers or a single DC motor and puller cable. The latter is the traditional mechanical parking brake. Also, the type of sensors available may vary. This perspective thus includes the following features: `DCMotor`, `PairEActuator`, `TiltSensor`, `EffortSensor`, `ClutchPosition`, `DoorPosition`, and `VSpeed`.

As discussed with a domain expert of the EPB SPL, FRoGs could help during the design phase to visualise the impact of different configurations of features (*Customer* variability or *Environment*) on the final solution (variability in *Components*). Also, it can be used before the design (product planning) to choose the possible components for future products from several information sources (stakeholders).

9.4.2 Visualising the feature relations

We obtained 20 FRoGs, one per feature, after automatically mining the 200 existing configurations of the EPB case study. Regarding soft constraints, the values used for *enc_{threshold}* and *dis_{threshold}* are 25% and 75% respectively. We present the number of constraints that

can be identified by visualising the FROGs in Table 9.3. For example, the second row in the table corresponds to the FROG of `PBSAutomatic` shown in Figure 9.6. We can see that this FROG displays two hard constraints and one inferred. Also, it displays four potential soft constraints (three discourages and one encourages). Before experimenting with domain experts, we manually checked the correctness of the visualised information. Concretely, we checked that the formalized hard constraints and inferred constraints of the FM appear in the displayed FROGs.

Table 9.3: Number of the identified constraints using FROGs

FROG central feature	Hard	Inferred	Soft
<code>PBSManual</code>	2	0	2
<code>PBSAutomatic</code>	2	1	4
<code>PBSAssistance</code>	2	1	3
<code>ClutchPedal</code>	1	1	4
<code>GBAutomatic</code>	2	0	4
<code>GBManual</code>	1	0	4
<code>AuxiliaryBrakeRelease</code>	0	0	4
<code>PullerCable</code>	5	2	0
<code>ElectricActuator</code>	5	2	0
<code>Calipers</code>	4	1	0
<code>Traditional</code>	4	2	0
<code>TemporaryMonitor</code>	1	2	2
<code>PermanentMonitor</code>	1	1	1
<code>DCMotor</code>	1	1	3
<code>PairEActuator</code>	1	1	3
<code>TiltSensor</code>	0	0	5
<code>EffortSensor</code>	0	0	4
<code>ClutchPosition</code>	0	0	4
<code>DoorPosition</code>	0	0	4
<code>VSpeed</code>	0	0	4
Total	32	15	55
Average	1.6	0.75	2.75
Standard Deviation	1.67	0.79	1.68

We now discuss the analysis that domain experts carried out using FROGs and how it helps in the comprehension of feature relations in the EPB SPL. For instance, analysing the FROG associated with the `EffortSensor` feature (see Figure 9.10), it is observed that it has no hard constraints to any other features. That means that it has no relevant impact on the presence or absence of any feature. However, the feature `PullerCable` (see Figure 9.11) has a great impact on other features as the `PullerCable` FROG shows seven features on the excludes and requires zones.

In the EPB case study, any non-formalized hard constraint was discovered neither in the excludes nor requires zones. This means that, potentially, the FM is not missing hard constraints. However, thanks to FROGs, as presented in Table 9.3, a total of 55 potential soft constraints were found while mining the existing configurations. By visualising this, domain experts are able to think about two possibilities: a) whether it is an actual soft constraint that should be formalized or b) FROG displayed a fact based on the existing configurations but it is not an actual soft constraint.

The FROGs also represented a visualisation paradigm to understand the relations between stakeholder perspectives. From the 47 hard or inferred constraints, 20 of them are hard constraints relating features belonging to different stakeholder perspectives.

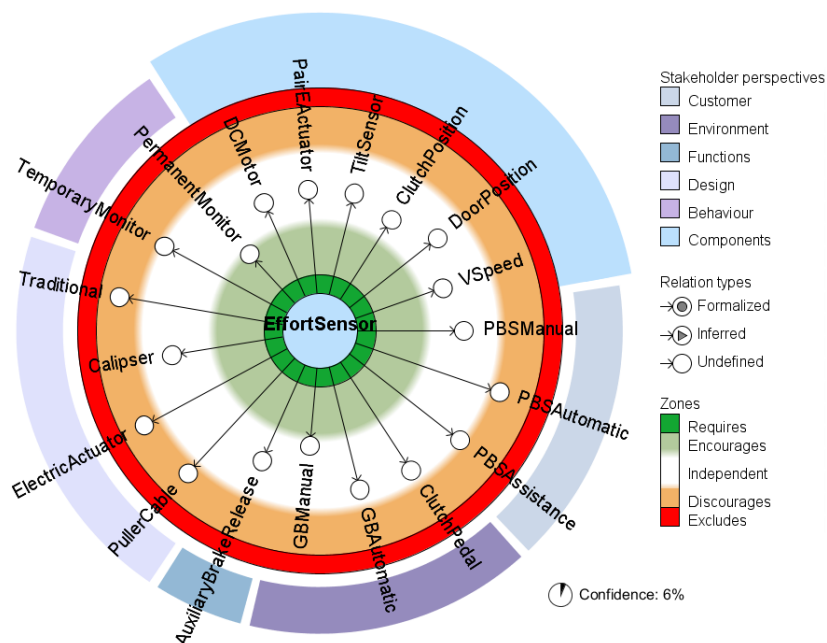


Figure 9.10: FRoG of *EffortSensor* feature without filters showing that it does not representatively affect other features.

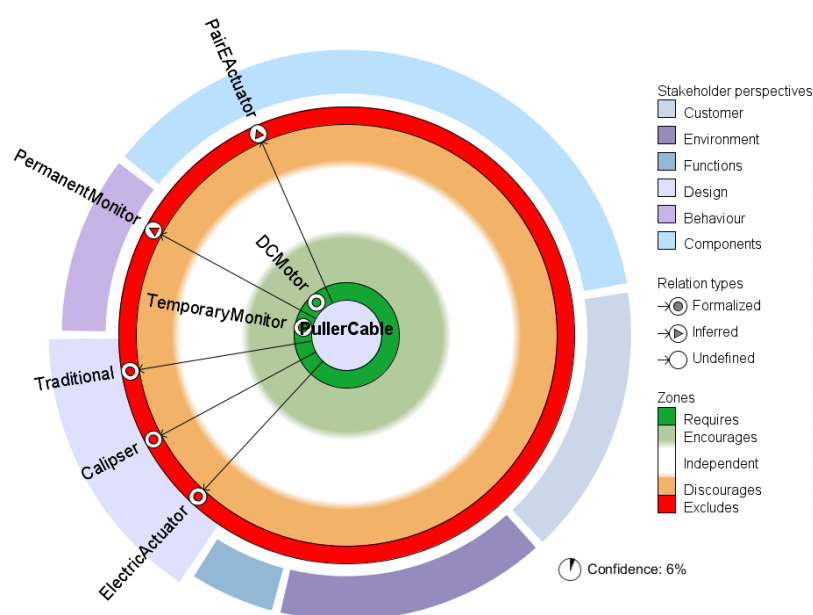


Figure 9.11: FRoG of *PullerCable* feature with a filter hiding Independent zone features to illustrate great impact on other features.

For example, Figure 9.11 shows four inter-stakeholder perspectives hard constraints: *TemporaryMonitor*, *PermanentMonitor*, *DCMotor* and *PairEActuator*. The other three *Traditional*, *Calipser* and *ElectricActuator* belong to the same stakeholder perspective as *PullerCable*. Regarding soft constraints, the 55 soft constraints that are in total are among inter-stakeholder perspectives. Figure 9.10 shows four *discourages* soft constraints regarding *EffortSensor*: *Traditional*, *ElectricActuator*, *PullerCable* and *PBSAutomatic*.

9.4.3 Extension considering feature conditions

Discussing with our industrial partner, we identified an interest to allow more than one feature in the center in the form of a condition. This way, instead of knowing the relation of a feature to the others, it is possible to visualise the relation of a condition of feature selections to the rest of the features. This allows to visualise, for example, how a set of feature selections from a given stakeholder perspective affects the features of another stakeholder perspective. As discussed before, the industrial partner showed interest in visualising the impact of *Customer* and *Environment* features in the features related to the *Components* stakeholder perspective.

Figure 9.12 shows a FRoG with two features in the center within the condition **PBSManual & ClutchPedal**. All the principles for constructing a FRoG is maintained for this extension of the FRoGs paradigm. The only change is that, in these cases, in f_i given f_j , f_j is replaced with the condition. In the current implementation we do not show any notation for relation types (formalized, non-formalized nor inferred).

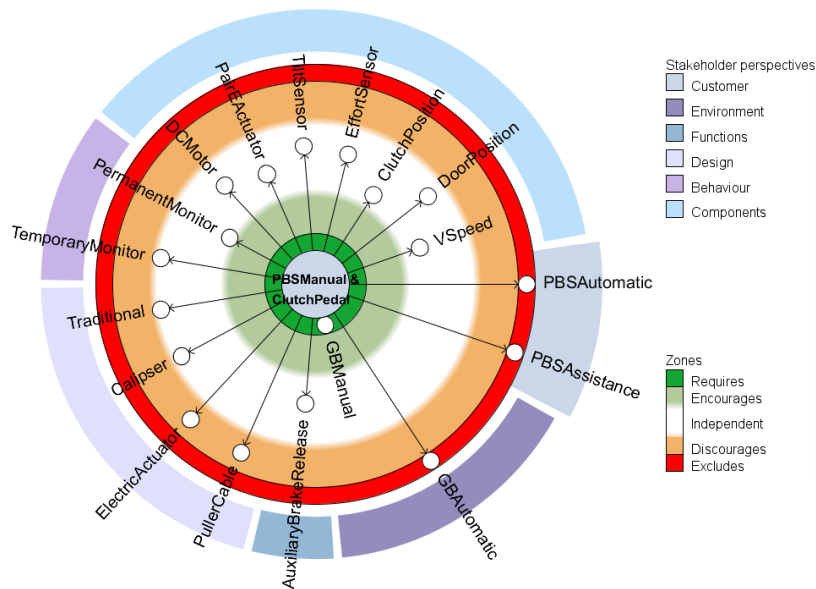


Figure 9.12: Visualising the impact of the selection of two features in the rest of the features.

9.5 Conclusions

Feature constraints discovery is one important activity in extractive SPL adoption. Concretely, feature constraints are at the core of FMs and identifying and maintaining them is a challenging task in SPLE. We presented a paradigm called FRoGs to help domain experts visualise the feature relations mined from existing configurations in an interactive way. This can help to potentially discover and formalize new hard and soft constraints. In addition, FRoGs can be used to document, at the domain level, each feature by displaying its relations among the

rest of features. We demonstrate in this chapter the usability of FRoGs on a real-world case study from a major manufacturer.

This work opens two main research directions. First, regarding to visualisation, when the SPL is operative, FRoGs can be improved to consider the time dimension. Indeed, the time dimension in the creation of each existing configuration could help users to understand the dynamics of feature relations and to reason about feature obsolescence or usage over time. Second, independently from the visualisation aspect, providing heuristics for determining the default values for the encourages and discourages thresholds is a relevant subject.

Part V

CONFIGURATION SPACE ANALYSIS
FOR ESTIMATING USER
ASSESSMENTS

10

END USER ASSESSMENTS IN SOFTWARE PRODUCT LINES: THE HSPLRANK APPROACH

This chapter is based on the work that has been published in the following paper:

- Jabier Martinez, Gabriele Rossi, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Estimating and predicting average likability on computer-generated artwork variants. In *Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings*, pages 1431–1432. ACM, 2015

Contents

10.1 Introduction	154
10.2 Motivating scenarios for estimation and prediction	155
10.2.1 Subjectivity in SPL-based computer-generated art	155
10.2.2 An SPL for digital landscape paintings	156
10.2.3 Variability and user experience in UI design	158
10.3 The HSPLRank approach	159
10.3.1 Variant reduction	160
10.3.2 Variant assessment	161
10.3.3 Ranking computation	163
10.3.4 Confidence levels for ranking items	166
10.4 Conclusions	168

10.1 Introduction

In many application domains of SPLE, product variants are intended to intensively interact with humans as presented in Section 2.3. In these Human-Computer Interaction (HCI) scenarios, products usability plays an important role in customers satisfaction. Estimate and predict user assessments for the different products is complex but crucial at the same time for 1) understanding user perception along the configuration space of the SPL and 2) maximizing the chances to select the best products for the targeted market. However, there are:

Many barriers to understand user opinions in SPLE contexts

In Section 2.3, we mentioned the three main barriers for understanding user expectations about the products within the SPL configuration space. These barriers are:

- *The users cannot assess all possible products.* For instance, less than three hundred optional features already permit to configure more products than the number of atoms in the universe. Given this combinatorial explosion, evaluating all products is impossible or prohibitively expensive as already highlighted by the SPL testing community [McG07, dCMCda14].
- *Human assessments are subjective by nature.* Socio-cultural factors, personal preferences or previous experiences in similar systems, affect the way a user rates a product. Therefore, mechanisms for summarizing different user opinions are needed to draw conclusions about products usability or, more generally, about the user experience.
- *Getting user assessment is resource expensive.* The fastest time-to-market in SPLE is achieved by eliminating human assessments as much as possible [McG07]. However, eliminating usability evaluations in products with HCI components is not always possible. Contrary to automated evaluations, HCI evaluation requires humans to experiment with the system which is a time consuming task. In addition, user fatigue is a relevant issue in usability evaluations when a user is exposed to several systems. When the user is exhausted or bored, the assessment quality and confidence degrade.

Contributions of this chapter.

- **Human-centered SPL Rank (HSPLRank):** This chapter introduces the theoretical and practical aspects of an approach for estimating and predicting user assessments within an SPL configuration space by leveraging user assessments in a limited number of variants.

This chapter is structured as follows: Section 10.2 discusses scenarios where estimation and prediction of user assessments is relevant. Section 10.3 details the phases of HSPLRank, our approach to overcome the presented challenges. Finally, Section 10.4 presents the conclusion. The evaluation of HSPLRank in two case studies is presented in Chapter 11.

10.2 Motivating scenarios for estimation and prediction

This section presents and discusses SPL application domains where estimating and predicting is specially relevant. Also, we introduce one case study in order to show a concrete example.

10.2.1 Subjectivity in SPL-based computer-generated art

With the whole myriad of software alternatives that exist today, the adoption of a software product is eventually dependent on users' subjective perception, sometimes beyond the offered functionalities. Being able to apprehend, estimate and predict this perception on the product as a whole will be an important step towards efficient production. Unfortunately, subjective perception of a software product is hard to formalize. It cannot even be computed with a simple formula based on the perception of its components: melting good ingredients indeed does not necessarily produce a good recipe. Systematically predicting the subjective appreciation of software is therefore an important research direction [HB12, Li12].

An extreme case where appreciation is pertinent, is when the intention of the product is just about “beauty”: this is the case of computer-generated art. By studying how computer-generated art products meet user perception of beauty, we can infer insightful techniques for estimating the perception of other types of software which may involve other HCI aspects. The practice of computer-generated art, also known as generative art, involves the use of an autonomous system that contributes to the creation of an art object, either in its whole, or in part by reusing pieces of art from a human artist, or using predefined algorithms or transformations [BE09]. This art genre is trending in the portfolio of many artists and designers in the fields of music, painting, sculpture, architecture or literature [Edw11, SL93, Per85, Fla98, Gre05, RM08]. We consume computer-generated art systems in our daily life as in videogames, cinema effects, screen-savers or visual designs.

In general, the autonomous systems for computer-generated art rely to some degree to a randomization step in the generation algorithms. However, when no relevant stochastic component is introduced and the creation is not limited to a unique art object, computer-generated art allows to derive different art objects in a predefined and deterministic fashion, giving rise to a *family of art products*. In Section 3.6.2 we mentioned some SPLs of art related products. Usually, because of the combinatorial explosion of all possible art objects, not all of them will actually be created. Besides, a number of them may not reach the desired aesthetic quality. Exploring the art product family to find the “best” products is thus challenging for computer-generated art practitioners. In this field, they have proposed the paradigm of evolutionary computer-generated art which consists in the application of natural selection techniques to iteratively adapt the generated artworks to aesthetic preferences [RM08].

We can take into account human feedback for ranking the art objects in a process that explores the collective understanding of beauty. This process is strongly related to the notions of group intelligence which discuss how large numbers of people can simultaneously converge

upon the same point [Sur05]. The case of art is interesting as it constitutes the worst case scenario for the homogeneity of opinions due to the subjective essence of art. Thus, we focus on user assessments on art product families and how it can help artists to understand people perceptions of their art products. In this scenario, the challenge is to answer the following question:

- Can we empirically study whether we can predict the like/dislike human perception of an artwork variant built by assembling perceivable components?

If the objective is to find the optimal products and rank computer-generated art product variants based on human feedback, this question leads to the following sub question:

- Given the combinatorial explosion of configurations and the limited resources, how can one identify and select the optimal subset of products that are relevant for human assessment?

One must keep in mind that, as we only consider a subset of products for human assessment, most of the products have not been assessed yet. In addition, user feedback is subjective by nature. Hence, another sub question for ranking product variants is:

- Given a subset of assessed product variants, how can one infer the user assessment of the non assessed products? How can one aggregate user assessments to calculate estimations even for the already assessed products?

HSPLRank, that will be explained later, is an approach that can be used for leveraging user assessments towards *ranking* all the possible art products based on estimations. We present an introduction of our case study in this scenario dealing with high subjectivity.

10.2.2 An SPL for digital landscape paintings

We have actively collaborated with Gabriele Rossi, an Italian art painter based in Paris, with whom we were able to conduct a large study on computer-generated art built by composing portions of paintings. We developed this system using SPLE techniques. In the last years, Gabriele has been drawing abstract representations of landscapes with quite a recognizable style of decomposing the canvas in different parts: a **Sky** part, a **Middle** part and the **Ground**. This variability is further enhanced by the fact that the **Sky** and **Ground** are mandatory while the **Middle** part is optional. For the realization of the computer-generated art system, the artist created different representative paintings for each part. For the **Sky** part, he painted 10 representative sky paintings hereafter noted S_i , with $i \in [1..10]$. Similarly, he painted 9 paintings of the middle part (noted M_i , where M_{10} means the absence of middle part), and 10 paintings of the ground part (noted G_i).

Another variability dimension identified by the artist concerns the perception of the composition. Indeed, this perception can change if any instance of any part is flipped horizontally, thus adding an optional property for configurations (**Flip**). For example, a given ground part may have more brightness in its left or right side, adding a compositional decision for where

to place this brightness in the whole painting. The artist also stated that each of the parts could take more or less space in the canvas. We therefore included an optional property for **ExtraSize**. For example, if a painting should have the sky visible in only a small section of the canvas, the middle and ground parts must be of larger size.

The variability in this domain can be expressed through a FM which exposes the different configurations that can be selected to yield painting variants. It was thus easy to introduce the artist with different FM concepts and to discuss the different elements of paintings in terms of features. This led to the establishment of a FM for his painting style that is presented in Figure 10.1.

A specific configuration, i.e., a given selection for sky, middle and ground, is assembled by superposition of the different parts. In Figure 10.2 we illustrate how different paintings in the style of the artist can be obtained by flipping one or more painting parts or/and increasing the size of elements. We have implemented a compositional derivation tool with Processing [CF12] which, given a configuration from the FM of Figure 10.1, we can generate a digital landscape painting based on the reusable assets of painting parts provided by the artist.

Using this SPL, the objective is to provide a solution for the challenges described in Section 10.2.1. Before introducing this solution we describe another relevant scenario for estimation and prediction of human assessments.

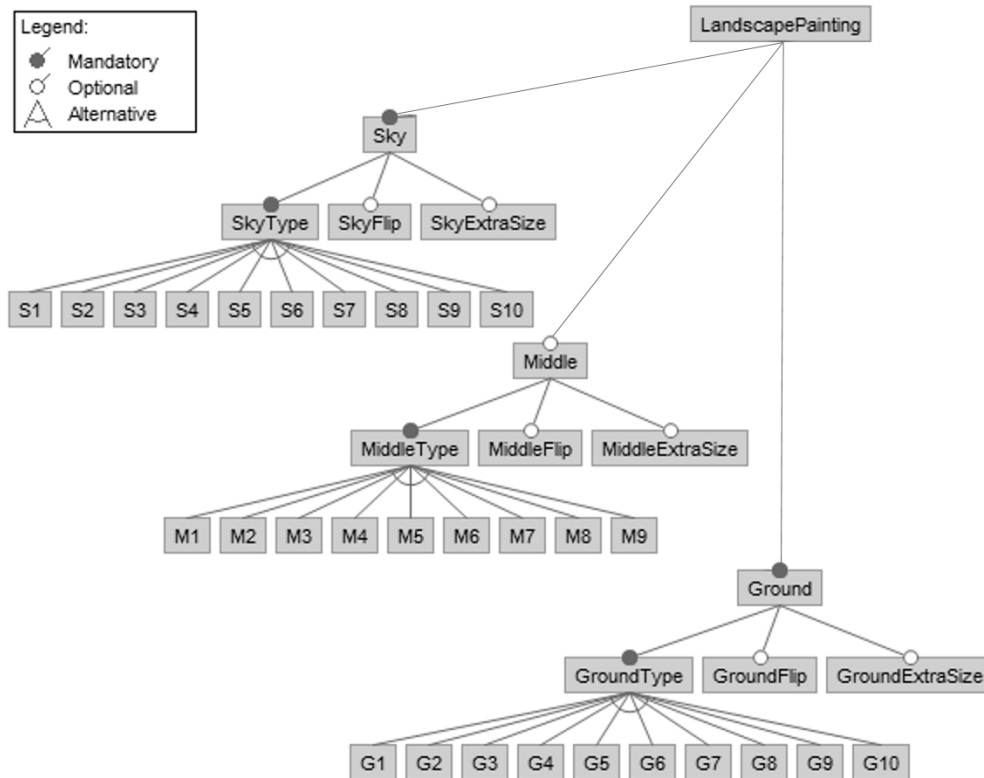


Figure 10.1: Feature model of the landscape paintings.

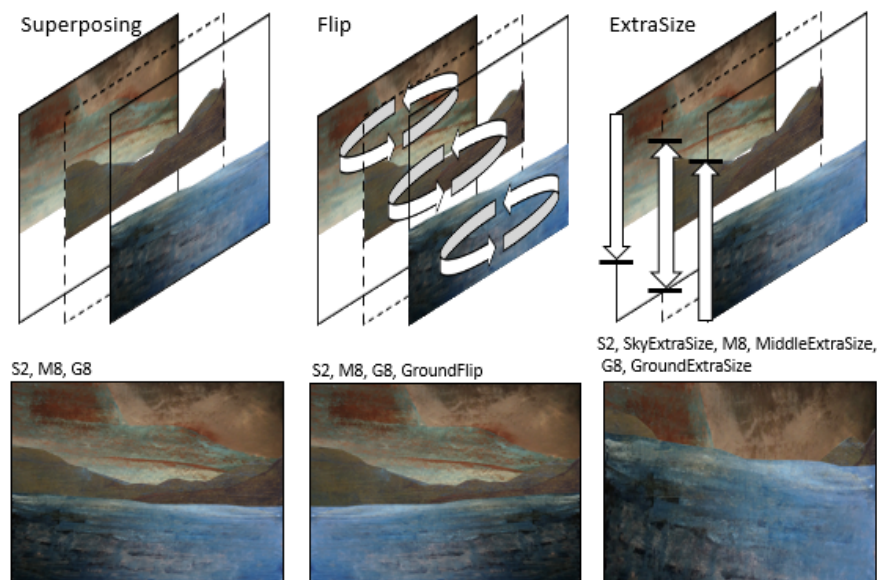


Figure 10.2: Painting derivation process and examples.

10.2.3 Variability and user experience in UI design

User involvement is one of the key principles of the user-centered design method [DIS09]. It is nearly impossible to design an appropriate UI without involving end users throughout the product design and development. Practitioners are often required to produce several variants of the same product, each of them presenting a different look and feel and different user interaction patterns. These variants are presented to end users who provide feedback for identifying the most adapted variant for their context of use.

Variability management in UIs serves to formalize the possible variants to be evaluated. The design, from scratch, of several UI variants is not practical when, eventually, all variants share among themselves some commonalities that can be leveraged in a systematic reuse process. Thus, it is essential to capture the variability of the design choices among potential variants. Many works have initiated first steps in the management of HCI-specific variability using SPL-based approaches [PHDB12, PBD10].

As discussed in Section 10.1, the large amount of possible configurations presents many challenges for achieving UI variant evaluations in a feasible way, we need to try to reduce the number of variants to evaluate, perform the variant assessment and finally select the best one according to estimations. It is worth to mention that by evaluation, we mean not only evaluating usability of the HCI system but also general user experience.


We focus on early binding variability [KLD02] where design alternatives are chosen at design-time. Therefore, this scenario is not focused on customization options which are done by end users themselves within the UI, nor to self adaptive systems which corresponds to run-time binding of variability.

In order to ensure that final product variants are of good quality, in this scenario we should answer the following question:

- How to plan evaluations for a limited number of users and in a reasonable amount of time against a large number of viable products in order to select relevant variants?

Consequently, this leads to two research sub questions:

- How to reduce the number of evaluations each user is involved in?
- How to rapidly converge with a rather small number of evaluations according to a given context/situation?

 In Section 11.3 of next chapter, we present a specific case study in this scenario dealing with the design of a Contact List UI.

10.3 The HSPLRank approach

In order to apply HSPLRank for the estimation and prediction of user assessments, we assume the existence of an operative SPL. Concretely, the input to apply our approach is an SPL where the derived products include HCI components. Figure 10.3 presents an overview of HSPLRank. On top of the SPL, HSPLRank is built on three phases: Variant reduction, variant assessment and ranking creation. Each of them will be presented in next subsections but, before we get into the details, we summarize them:

- **Variant reduction** is a phase driven by domain experts aiming to reduce the configuration space of the SPL. This is possible by injecting soft constraints regarding usability aspects. Soft constraints were presented in Section 2.1.2. The subset of valid configurations satisfying soft constraints is called the viable configuration space (i.e., configurations which have sufficient usability).
- **Variant assessment** is a phase driven by a pool of users to explore the configuration space during a predefined amount of time or for a finite number of assessments. This phase relies on an Interactive Genetic Algorithm (IGA) [ES03, Tak01] which is seeded with configurations from the viable space. This algorithm evolves a set of configurations using the result of the assessments of the users pool. The user assessment is performed using the product from the configuration assigned by the IGA. All assessments obtained during the exploration of the configuration space are used to feed a data set which is used in the next phase. Data set items are composed of a pair of a configuration and the score of the user assessment.
- **Ranking creation.** Using a data mining technique, HSPLRank estimates and predicts the score of any possible configuration enriched with confidence metrics about the estimation. This technique is leveraged to produce a ranking.

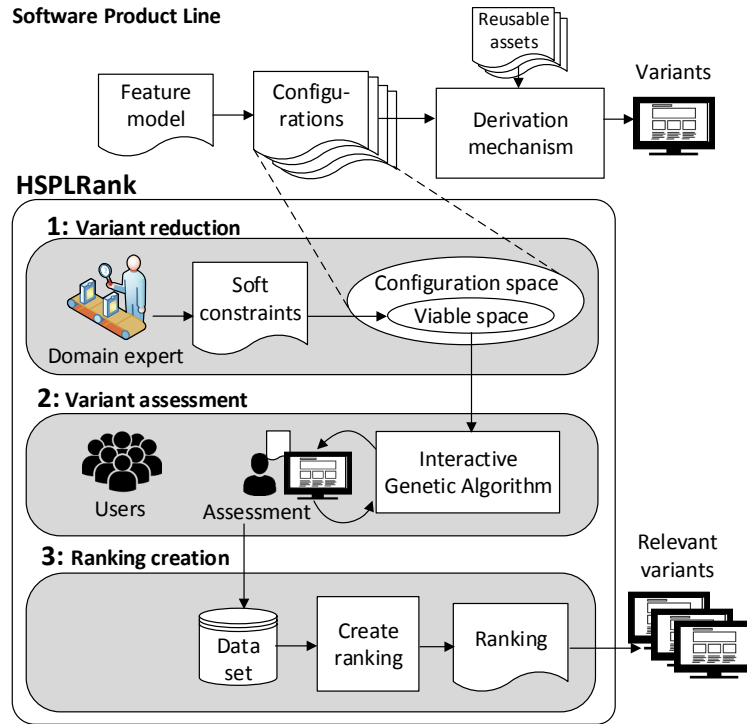


Figure 10.3: HSPLRank is built on top of an SPL and it consists of the variant reduction, variant assessment and ranking creation phases.

After the application of HSPLRank, the most relevant variants in a given scenario can be finally selected by the domain experts.

10.3.1 Variant reduction

Given that the number of possible configurations can grow exponentially with the number of features, we propose first to discard the configurations which represent variants with very poor expected quality. This quality level is based on *expert judgement* of the expected impact of features or feature combinations in the final product. In order to formalize this domain knowledge, we use soft constraints, presented in Section 2.1.2, to describe the *viable* configuration space. Therefore, viable configurations represent variants with the minimum acceptable quality to be tested by end users.

Figure 10.4 shows an illustrative diagram of the configuration space. As we mentioned, the configuration space represents the set of all possible configurations that can be produced with the FM (i.e., the configuration space is the set of valid configurations which satisfies hard constraints). The viable configuration space represents the product variants which are of sufficient expected quality and that satisfy all the constraints, both hard and soft constraints.

In an scenario of UI design, our approach aims at requiring user assessments restricted to *a priori* usable UIs. As soft constraints are currently based on expert judgement, usability



Figure 10.4: Configuration space and viable space.

experts are involved in order to produce them. The usability experts analyze the usability-related impact of feature combinations in a given SPL derivation scenario of UIs. Following with the E-Shop example presented in Section 2.1.2 (FM shown in Figure 2.1a) and considering a scenario of an E-Shop with a large amount of items in the catalogue, the usability expert can determine that not including a **Search** functionality in the UI results in a poor usability of the E-Shop. Table 2.1 presented the configuration space of the E-Shop that consist of eight possible configurations. By including the constraint $soft(\text{Search})$, the viable space is reduced to four configurations as only **Conf 2, 4, 6 and 8** satisfy the soft constraint. For this scenario of large catalogues, only this viable space will be used as the input for the variant assessment phase.

Soft constraints define the boundaries of what is supposed to be usable in a given scenario. As consequence, the characteristics of the case study are decisive because the soft constraints are not necessarily the same in all cases. For instance, for an E-Shop with only two items, the **Search** functionality is possibly counterproductive. In this case, the soft constraint will be $soft(\neg \text{Search})$. Soft constraints can be unsatisfied in a final product but, according to the usability expert, most probably it will be a non usable variant. In other words, soft constraints reduce, *a priori*, most of the non-viable products but we can still produce UIs that may result, *a posteriori*, in bad user experiences or in unexpected good solutions.

Given that the viable space could remain very large, we still require a reasonable way for variant assessment with end users. Next section explains the assessment phase and how it benefits from this variant reduction phase.

10.3.2 Variant assessment

For users assessment, HSPLRank proposes the use of search based techniques. Concretely, we have chosen to implement an evolutionary approach in order to select the variants for user assessment. Evolutionary algorithms apply the principles of natural selection to approximate optimal solutions in multi-dimensional search spaces [ES03]. In the context of Human-centered SPLs, a solution is a configuration, the dimensions are the features, and we measure the adaptation to the environment by exposing the derived product to user evaluation. Traditionally, in genetic algorithms, the fitness function is automatically calculated using a formula or an automatic computation. In our case we rely on an IGA [Tak01] where the fitness function, which is the genetic algorithm operator that drives the evolution, is not automatically calculated but provided interactively by users. Each generation of the evolution consists of a population of configurations. After a given number of generations, the evolution

favours the better adapted solutions which have more chances to propagate their chromosomes (i.e., feature selections). Therefore, the genetic algorithm tries to guide the search to regions of the configuration space with relevant variants.

The initialization of the first generation has a great impact on the results of a genetic algorithm [FA12, MRSB13]. HSPLRank proposes, for the first generation, only to include viable products which were enclosed in the previous variant reduction phase. In the genetic algorithms field, this technique is known as *seeding*. Population seeding techniques consist in initializing the population with solutions that are expected to be good solutions in promising regions of the search space. We further propose to try to guarantee the diversity of the seeded initial population by random selection of individuals from the viable space. Figure 10.5 illustrates an example of implementing an IGA for HSPLRank. We can observe the pool of users and the population through the generations. Figure 10.5 corresponds to the settings used in the case study that will be presented in Section 11.2.

For the generations following the initial population, we propose two alternatives: 1) we assume that soft constraints, stated by usability experts, must be always respected or 2) we allow to explore configurations beyond the viable space. If we assume that usability expert constraints are correct, we can strictly restrict the new generations to the viable space by avoiding non-viable products in all generations. However, we can also consider unexpected cases by allowing the exploration (starting from generation number two) of *a priori* non viable configurations. End users may eventually find that a combination which does not satisfy some soft constraint is well adapted for them. Therefore, in the realization of HSPLRank it can be decided between restricting the exploration phase of the genetic algorithm to the viable space or using the viable space only for seeding.

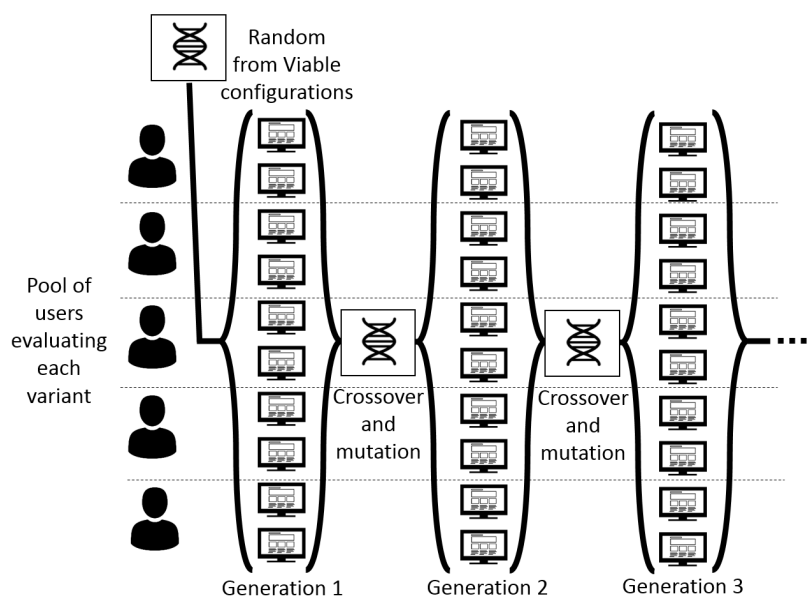


Figure 10.5: Variant assessment driven by the interactive genetic algorithm.

10.3.3 Ranking computation

The phase for the ranking computation takes as input the data set produced by the previous evolutionary phase, and as output we estimate and predict the score of the assessment of any possible configuration. Through this, we create a ranking for all possible configurations including those that are not in this initial data set.

First, a similarity measure for computing a *similarity distance* between two configurations in the domain must be used. We then design a protocol for aggregating feedback from different users for a set of *similar* products. This protocol works by defining a *similarity radius* which will be relied on to compute the *weighted mean score* for each possible configuration. We further investigate the suitability of the methods for defining the radius as well as for the approaches for computing the weighted mean. Finally, with the hypothesis that the computed score of each configuration should equate the human expected perception, we create a ranking of all possible configurations. We also provide confidence metrics for each of the ranked items. We detail the operators of the ranking computation phase:

Similarity distance: Given two configurations C_i and C_j , we aim at formally computing a value for the similarity distance between them. The notion of similarity distance was already studied in the software engineering literature, specially by the SPL testing community [HAB13, HPP⁺14]. One example discussed in these works is the use of the Hamming distance [Ham50]. The space defined by the set of SPL features can be considered as a binary string where each position corresponds to a given feature. In this binary string, 0 stands for selected feature and 1 for not selected feature. Then, the Hamming distance can be calculated for any pair of configurations by counting the minimum number of substitutions required to change one string into the other. Hamming distance for calculating configurations similarity was previously used in the SPL domain [Al-15]. Another example is the Jaccard distance which is a set based similarity distance where we consider that a configuration is a set of selected and not selected features. Then, the Jaccard similarity is calculated as the size of the intersection of the sample sets divided by the size of the union [HPP⁺14].

It is worth mentioning that the use of similarity metrics such as Hamming or Jaccard, in the context of HSPLRank, imply the assumption that when two products are similar in the way they were assembled, they will be appreciated similarly. Apart from these generic configuration similarity metrics, other approaches can be based on ad hoc domain-specific similarity functions between configurations or between the products derived from these configurations. HSPLRank does not impose a similarity distance method. In next chapter, in Section 11.2 dealing with the Paintings SPL, we use an ad hoc distance function to measure the distance between two paintings. In Section 11.3 regarding the case study in UI design, we use the Hamming distance. In the ad hoc distance function, we give different weights for the features, meaning that differences in the selection of some features are considered less relevant for the similarity of two configurations while other features are considered more relevant. The Hamming approach results in values from 0 (the two configurations are the same) to the maximal value which is the total number of features (completely different).

Similarity radius: To be part of the *neighborhood* of a given configuration, any configuration must be inside a *similarity radius*. This radius allows to restrict the products that will be considered similar enough for inferring information from one to the other.

Figure 10.6 illustrates the example of a configuration C_c (small white central point) and all configuration instances of the data set (black points) placed according to their relative distance to C_c . When we consider a distance r to define the neighborhood (circumference line), we note that some instances remain in the radius of C_c . The scores of the assessments of these instances which are inside the threshold of the similarity radius (the numbers on the black points) are used for the calculation of the weighted mean that represents the inferred score for C_c . The ones that are outside the similarity radius are considered dissimilar and no information can be inferred regarding the white central point. This Figure corresponds to a real configuration of the Paintings case study that will be presented in Section 11.2.

Weighted mean score: In order to assess the collective agreement on the score of a given configuration, we consider its neighborhood and compute the mean score in a similarity radius. HSPLRank considers this computed score as the expected level of appreciation by users. For computing reliable mean values, a weight is assigned for each instance of the data set. This weight depends on their proximity with the configuration C_c whose score wants to be estimated.

Equation 10.1 serves to compute the weighted mean score \bar{s}_c for configuration C_c where s_i and w_i respectively represent the score and the associated weight of each of the N configurations C_i that are in the data set and that are within the similarity radius.

$$\bar{s}_c = \frac{\sum_{i=1}^N w_i \cdot s_i}{\sum_{i=1}^N w_i} \quad (10.1)$$

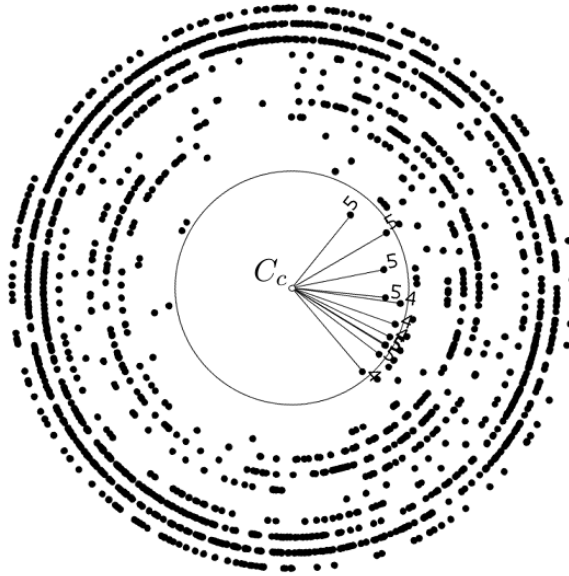


Figure 10.6: Similarity radius of a given configuration C_c .

Figure 10.7 illustrates four approaches for assigning a weight to the scores of configurations in the similarity radius. In all these weighting approaches, we consider $w_i = 0$ when $d_{c,i} > r$, where $d_{c,i}$ is the similarity distance between configurations C_c and C_i , and r the value of the similarity radius. In the first approach (Figure 10.7a), all scores within the radius are weighted equally independently of the distance between configurations. It is equivalent to a standard average computation. In the second approach (Figure 10.7b), a linear distribution of the weights is implemented. The third approach (Figure 10.7c) exponentially maximizes the weights for configurations that are close to C_c . Finally, the fourth approach (Figure 10.7d), modifies the standard average approach by slightly reducing the weights of configurations that are farthest from C_c .

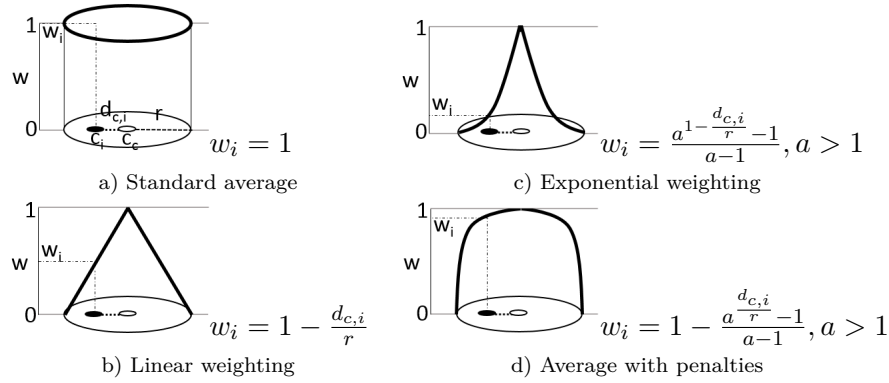


Figure 10.7: Different options to calculate the weight.

Empirical selection of the approach settings: In order to select the radius and the weighting approach, we explore different combinations to identify the one that minimizes the error rate. We rely on a 10-fold cross-validation scenario. We shuffle the instances in the data set and then we split it into 10 folds. 9 of these folds are used as configurations with real scores based on user feedback, and 1 fold is used for testing the inference of score for the missing ones. The error rate is computed based on the difference between the expected value (the computed weighted mean score) and the actual user feedback score for each instance of the test set. The evaluation is a numeric prediction so we selected the *mean absolute error* (MAE) as error rate metric. Equation 10.2 shows how MAE is calculated, where T is the number of instances in the test set, \bar{s}_i is the weighted mean score of C_i computed with the training set (which represents the expected score) and s_i is the score of C_i in this instance of the test set (the actual score).

$$\text{mean absolute error} = \frac{\sum_{i=1}^T |\bar{s}_i - s_i|}{T} \quad (10.2)$$

Based on the empirical investigation of the different radius and weighting approaches using the 10-fold, we tune the ranking computation parameters to reliable values. Concretely, we aim at minimizing MAE rates while maximizing the coverage of configurations. Regarding the latter, besides the MAE values, we are also interested in knowing how the choice of different radius values impacts the coverage of the test set. If the radius is small there is a possibility

that, for some configurations, the neighboring is empty, thus preventing the computation of any mean score. Because such instances are not taken into account for the computation of the MAE, it is important to know the average percentage of the test sets that is covered. We calculate the test set coverage for each radius as the number of configurations from the test set from which we can estimate a score (i.e., the training set has assessments in the neighborhood), divided by the size of the test set.

Ranking creation: We exhaustively compute the weighted mean for all possible configurations. In the cases where a configuration C_c has no neighbor within its similarity radius, no score is computed and it is ignored in the final ranking. In our case studies, an exhaustive calculation for estimating all configurations was computationally feasible. In other cases, where exhaustive calculation might not be feasible, other approaches will be needed to limit the amount of configurations to analyse in order to make the approach more scalable at this phase. For instance, one research direction could be the use of non-euclidean centroid calculations.

10.3.4 Confidence levels for ranking items

The obtained ranking is based solely on the weighted mean scores computed with the data set instances within the similarity radius. However, to yield a comprehensible ranking, it is important to assign a confidence level for each computed score. We define three main metrics for measuring confidence. Concretely, we focus on the average distance between a given configuration and its neighbors, and on the number of neighbors that were relied upon for computing the mean.

Neighbors similarity confidence: The first metric explores the average distance with the neighbors. Figure 10.8 illustrates the importance of this metric: on the left, a unique neighbor, with a score value of 5 (e.g., let be 5 the maximum score), is shown to be very close to the center (C_c). On the right, we see another case with a unique neighbor which is farther from the center. In both cases, the weighted mean is 5, however, intuitively, one is more confident about the accuracy of the mean score for the left than for the right case. The instance that was assessed in the case of the left is more similar to the configuration that we want to estimate.

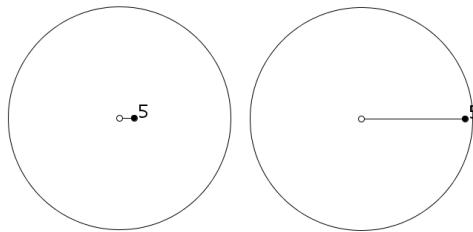


Figure 10.8: Illustration of the importance of *Neighbors similarity confidence*. The weighted mean is 5 in both cases but, in the case of the left, a more similar configuration was assessed by a user.

We define a neighbors similarity confidence as in Equation 10.3 where N is the number of data instances within the radius of C_c (i.e., neighbors). $nsimc_c = 1$ when all data instances are in the center, i.e., all configurations in the radius are identical to the configuration for which

the score is inferred. Also, when $N = 0$, this metric is not applicable. For the calculation of $nsimc$ in the case studies of next chapter, we will use the linear weighting approach. For the example scenarios of Figure 10.8 and using a similarity radius of 1.5, the C_i in the left of the figure is at $d_{c,i} = 0.2$ from C_c , and the $nsimc$ is thus 87%. In the right of the figure, the C_i is at $d_{c,i} = 1,4$ so the $nsimc$ is 7%.

$$nsimc_c = \sum_{i=1}^N \frac{w_i}{N} \quad (10.3)$$

Neighbors density confidence: The second metric concerns the density of neighbors in the similarity radius. Figure 10.9 illustrates the importance of this metric: on the left side, a unique data instance, with a score value of 5, is shown within the radius; on the right side a different case is presented where several data instances are present in the similarity radius, all with a score value of 5. In both cases, the weighted mean has a value of 5, but intuitively again, the confidence is greater for the second case as we have more data to support the estimation.

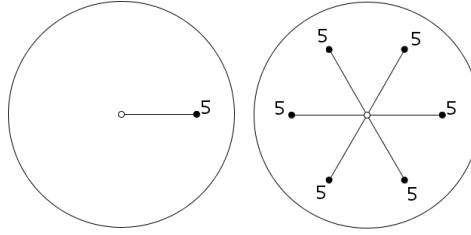


Figure 10.9: Illustration of the importance of *Neighbors density* confidence. The weighted mean is 5 in both cases but, in the case of the right, we have more user assessments.

The neighbors density confidence is computed as in Equation 10.4 where N represents the number of data instances within the radius of C_c , max is the highest value of N found in all the possible configurations, and the function $withNeighbors(i)$ returns the number of instances from the data set containing exactly i neighbors within their radius.

$$ndenc_c = \sum_{i=1}^N withNeighbors(i) / \sum_{i=1}^{max} withNeighbors(i) \quad (10.4)$$

With this metric we obtain a neighbors density confidence of 0.5 when the number of data instances within the radius of the configuration corresponds to the median of $withNeighbors(i)$.

Global confidence: Finally we define a global confidence metric, shown in Equation 10.5, that takes into account the previous metrics by assigning weights. For the case studies of next chapter, we decided to put more weight to $ndenc$ giving emphasis to the number of instances used to compute the mean. We made the design decision of setting w_{ndenc} to 0.75 and w_{nsimc} to 0.25 in the computations presented in next chapter.

$$gconf_c = w_{ndenc} \cdot ndenc + w_{nsimc} \cdot nsimc \quad (10.5)$$

10.4 Conclusions

Leveraging human perceptions in the context of SPLE is an emerging challenge dealing with an important aspect of products success: user expectations on product variants. We present the HSPLRank approach towards estimating and predicting human assessments on variants for the creation of a ranking of all the possible configurations. This ranking, enhanced with confidence metrics for each ranking item, has the objective to serve as input for selecting the most relevant variants. HSPLRank contains three phases at which we use 1) the injection of extra domain constraints (soft constraints) to enclose the viable variants, 2) an interactive genetic algorithm for the initial data set creation and 3) a tailored data mining interpolation technique for reasoning about the data set to infer the ranking and the confidence metrics.

We have presented two scenarios for HSPLRank: The first one is related to an scenario that can be considered as the worst software project case dealing with human assessments subjectivity, which is the case of computer-generated artworks production. The second one is related to variability in UI design. A computer-generated art was presented and it will be evaluated in next chapter together with the presentation and evaluation of another case study on UI design.

11

SOFTWARE PRODUCT LINE CASE STUDIES FOR ESTIMATION AND PREDICTION

This chapter is based on the work that has been published in the following paper extended with the Contact List case study:

- Jabier Martinez, Gabriele Rossi, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Estimating and predicting average likability on computer-generated artwork variants. In *Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings*, pages 1431–1432. ACM, 2015

Contents

11.1 Introduction	170
11.2 Paintings case study	170
11.2.1 HSPLRank realization	171
11.2.2 Evaluation	175
11.3 Contact List case study	176
11.3.1 An SPL for Contact List design alternatives	176
11.3.2 HSPLRank realization	180
11.3.3 Evaluation	184
11.4 Discussion	188
11.5 Threats to validity	190
11.6 Conclusion	191

11.1 Introduction

In previous chapter, Chapter 10, we introduced HSPLRank, an approach to rank the configurations of an SPL to identify relevant products to a given end user context. In Section 10.2 we presented two scenarios where its usage is relevant which are SPL-based computer generated art and UI design. The objective of this chapter is to detail the results of the case studies that we conducted as well as presenting a detailed discussion about our experiences with HSPLRank and its threats to validity.

Contributions of this chapter.

- **The Paintings case study:** A large scale case study in SPL-based computer generated art in collaboration with a professional painter as described in Section 10.2.2.
- **The Contact List case study:** A case study in UI design concerning the implementation of an SPL for Contact List systems.

This chapter is structured as follows: Section 11.2 presents the results of the paintings case study. Section 11.3 introduces the Contact List case study and the results of applying HSPLRank. Then, Section 11.4 discusses important general topics of our experience in the two case studies. Section 11.5 details the threats to validity and Section 11.6 concludes this chapter.

11.2 Paintings case study

This section explains the realization of HSPLRank and evaluate the results in the case study presented in Section 10.2.2 dealing with an SPL for digital landscape paintings.

Objectives and Settings

The objectives for this case study are:

- Study the error rates of the estimations provided by HSPLRank to evaluate its soundness.
- Qualitatively study the opinion of the artist regarding the relevant variants obtained through HSPLRank.

We proposed a validation on the feasibility of HSPLRank after applying it to a real-world installation for computer-generated art. The installation for collecting user feedback was available to the public as part of an art festival at Théâtre de Verre at Paris in 2014. This provided the location for the HSPLRank variant assessment phase with the attendees. The artistic project of evolutionary paintings is called AdherentBeautyⁱ. For further evaluations, the experiment was repeated but instead of using a collective of persons, it was operative only for one person each time (individualized experiments).

ⁱVideo explaining the art installation: <https://vimeo.com/139153572>

11.2.1 HSPLRank realization

Phase 1: Variant reduction

In this scenario, the artist was the domain expert which decided not to introduce any restriction to the viable space. That means that the configuration space is equal to the viable space and that the IGA will not be seeded. This was motivated by not adding prejudgements about what the attendees should like or dislike in the paintings. On the contrary, in Section 11.3.2 that concerns a case study on UI design, the variant reduction phase will be used.

Phase 2: Variant assessment

In order to address the combinatorial explosion of possible paintings, as proposed by HSPLRank, we rely on an IGA which explores the possible configuration space trying to reach optimal or suboptimal solutions. The IGA permits to create more data set instances in the regions that are more adapted to the fitness function. If we have more density in these regions we will have more confidence about user expectations regarding the most appreciated configurations.

In our case study, the fitness function for the IGA is based on the assessment captured using the device shown at the bottom of Figure 11.1. This device implements a physical 5-point scale. The values range from 1 (*strong dislike*) to 5 (*strong like*). When a user votes, the displayed painting (as shown at the top of Figure 11.1) vanished and the next painting of the genetic algorithm population is displayed. When all paintings from the population have been assessed, a new population is yielded based on the calculations of the genetic algorithm, and the exploration towards optimal paintings continues, until it is manually stopped at the end of the session. Thus, in our installation, the data set is constructed during a unique session with users.



Figure 11.1: Displayed painting and voting device with a five points scale.

Algorithm 2 shows the implementation of the IGA for this case study. In genetic algorithms terminology, this algorithm is a non-elitist, generational, panmictic IGA, and the *Data* section of the algorithm shows how the genotype of a landscape painting phenotype was designed. Concretely, Sky type, middle and ground positions are assigned with values representing all possible parts. In the case of the middle part, which is optional, value 9 represents its

absence in the composed painting. Further, for this special case, the flip and extra size features are irrelevant, and thus a special treatment in the operators of the genetic algorithm is performed.

At line 1, the *initialization operator* creates a pseudo-random initial population where we force all sky, middle and ground parts to appear in the initial population. The evolution starts at line 2 until it is manually stopped. During the evolution, from line 3 to line 6 each member of the population is evaluated using an *evaluation operator* based on user assessment. The *parent selection operator* is then based on a fitness proportionate selection (line 7). At line 8, the *crossover operator* is based on one-point crossover with the peculiarity that it is not possible to select the last two positions to force to crossover the sky, middle and ground parts. Then the *mutation operator* used at line 9 is uniform with $p = 0.1$. Such a high mutation factor is meant to prevent a loss of motivation from users (user fatigue) by reducing the likelihood that they will keep assessing similar products from the population, while thus enabling us to explore new regions. Finally, at line 10, the *survivor selection operator* is based on a complete replacement of the previous generation with the new generation.

Algorithm 2 Interactive genetic algorithm for data set creation in the Paintings case study.

input: population = 20 members, Genetic representation of a member = 9 positions: SkyType, SkyFlip, SkyExtraSize; MiddlePart, MiddleFlip, MiddleExtraSize, GroundType, GroundFlip, GroundExtraSize; Type value from 0 to 9, Flip and ExtraSize values from 0 to 1; Example: 801510210
output: data set of user assessments

```

1: population ← initializePopulation()
2: while ManualStopNotPerformed do
3:   for  $member_i \in \{population\}$  do
4:      $member_i.fitness \leftarrow \text{getUserFeedback}(member_i)$ 
5:     registerDataInstance( $member_i$ )
6:   end for
7:   parents ← parentSelection(population)
8:   offspring ← crossover(parents)
9:   offspring ← mutate(offspring)
10:  population ← offspring
11: end while

```

The installation was operative for 4 hours and 42 minutes and 1620 votes were collected. The IGA and its exploration process towards optimal products led to 1490 paintings being voted once, 62 paintings being voted twice and only 2 being voted three times. No data was gathered to make distinctions about different user profiles nor any control mechanism was used to limit the number of votes per person. On average we were able to register 5.74 votes per minute from around 150 people of different ages and sociocultural backgrounds who voted for one or more paintings.

Phase 3: Ranking creation

General approaches for the calculation of the similarity distance between two configurations were presented in Section 10.3.3. Instead of using general approaches, we designed and defined a domain-specific ad hoc distance calculated through the algorithm that we present in Algorithm 3. Other approaches might have been used. In our case study, to compare two configurations, we start by assuming that they are the same ($distance = 0$). The distance between them increases by 1 point for each different painting part. If a part is the same in

both configurations, we check the flip feature and, in case of dissimilarities, we increase the distance by 0.2. Finally, and whether the parts are identical or not, we increase the distance by 0.2 if a part has an extra size in one configuration and not in the other: extra size is independent to the part as it has an impact on the whole composition. To account for the fact that the optional middle part may not be present, we check that part P is not null.

Algorithm 3 Distance function for formalizing the similarity between two painting configurations.

input: Two configurations of paintings C_i and C_j
output: The distance $d_{i,j}$ between C_i and C_j

```

1:  $d_{i,j} \leftarrow 0$ 
2: for  $P \in \{S, M, G\}$  do
3:   if  $P_i \neq P_j$  then
4:      $d_{i,j} = d_{i,j} + 1$ 
5:   else if  $P \neq null \ \&\& \ isFlipped(P_i) \neq isFlipped(P_j)$  then
6:      $d_{i,j} = d_{i,j} + 0.2$ 
7:   end if
8:   if  $P \neq null \ \&\& \ isEnlarged(P_i) \neq isEnlarged(P_j)$  then
9:      $d_{i,j} = d_{i,j} + 0.2$ 
10:  end if
11: end for
12: return  $d_{i,j}$ 

```

Using this algorithm, when the three parts are different and every part has extra size in one configuration and not in the other, we reach the maximum distance between two configurations which is 3.6. Thanks to this definition of distance, we can now reason about the neighborhood of a configuration in the space.

After the variant assessment phase, we took advantage of the user assessments in the data set to create the ranking. Using the 10-fold cross-validation explained in Section 10.3.3, Figure 11.2 shows the performances of different combinations of radius values and weighting approaches. The graph reveals that we start covering all instances of the test set with a similarity radius that in this case is 1.3. In Figure 11.2, we marked with a circle our selected parameters. We set the radius to 1.5 and selected the standard average weighting approach for the weighted mean score computation. A radius of 1.5 further fits artist's developed intuition of the maximum distance for two paintings to be considered similar paintings.

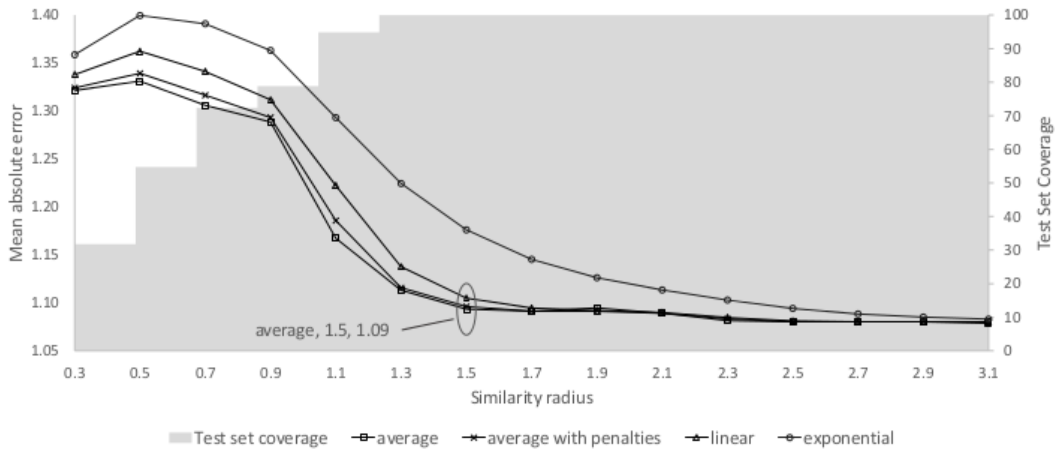


Figure 11.2: Mean absolute error for different radius values and weight calculation approaches.

We computed the weighted mean score for all 59,200 possible configurations of our case study and ranked them accordingly. Figure 11.3 depicts the paintings that were derived from the top 10 configurations with highest weighted mean scores ($wmean$). The highest $wmean$ was established at 4.75 for only one configuration. None of these configurations in the top 10 were part of the data set created in Phase 1 when collecting user feedback. For example, the third best configuration score was obtained based on the $wmean$ of 11 different configurations with scores computed from actual user feedback. Figure 11.4 depicts the bottom 5 configurations of the ranking. We observed that bottom configurations in the ranking had in general less confidence given the effect of the IGA that tried to avoid these regions.

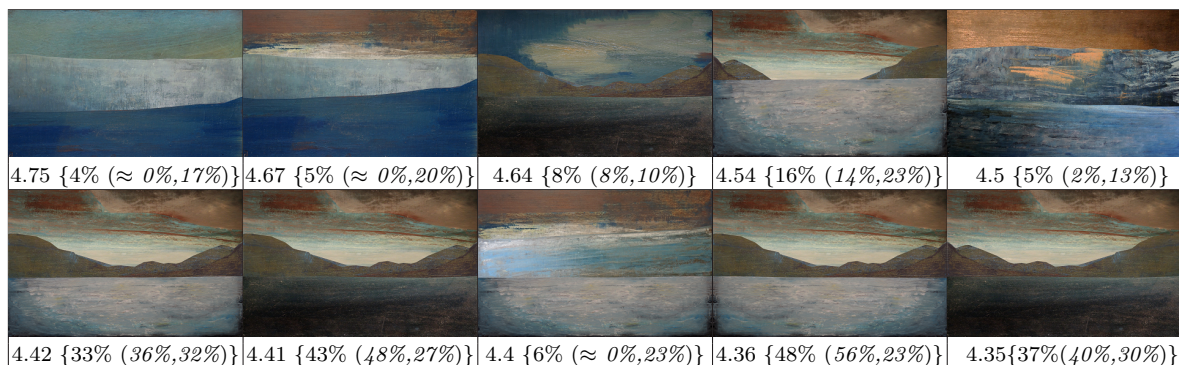


Figure 11.3: Top 10 ranking items defined by $wmean \{ gconf\% (ndenc\%, nsimc\%) \}$.

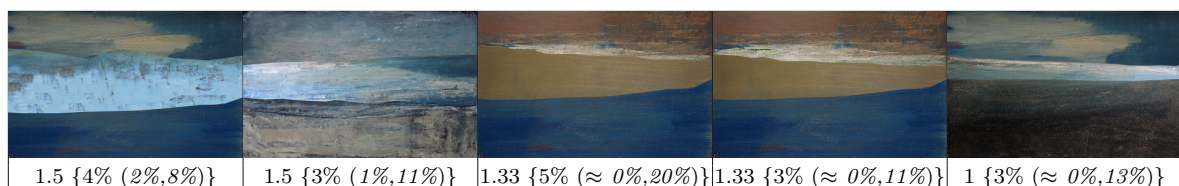


Figure 11.4: Bottom 5 ranking items defined by: $wmean \{ gconf\% (ndenc\%, nsimc\%) \}$.

We also used $gconf$ to filter and reorder the ranking items. Figure 11.5a shows the product that was derived for the configuration with the highest $gconf$ (82%). This configuration got a $wmean$ of 3.27 and holds the rank 14,532. The configuration with the highest $gconf$ for configurations that are liked (i.e., $score > 4$) holds the position 102 and the painting is shown in Figure 11.5b. Similarly, we identify the highest $gconf$ for configurations that are disliked (i.e., $score < 2$). It holds the rank 59,145 and it is shown in Figure 11.5c.

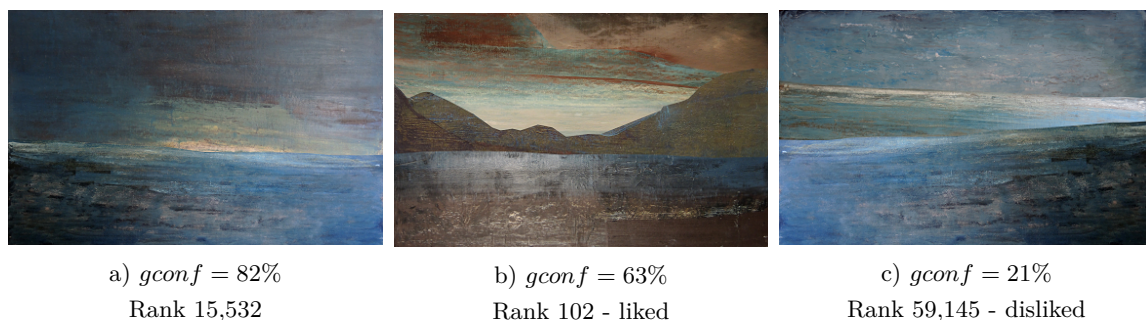


Figure 11.5: Key paintings with high global confidence.

11.2.2 Evaluation

We present a quantitative evaluation using a collective of persons and a qualitative evaluation from the artist’s perspective. We completed the evaluation with another quantitative evaluation repeating the whole approach with separated individuals.

Controlled assessment

The objective of the controlled assessment is to study the error rates in how the scores were estimated. In Section 11.2.1, we have already performed a 10-fold cross-validation to empirically select an optimal value for the similarity radius and the weighting approach. Now we also estimate the error rates using the whole data set with real user assessments to draw both training and test sets. The goal is to predict the score of a configuration and assess against the actual scores provided by users. We also compute the *resubstitution error* which is an optimistic case for evaluating classification approaches. Table 11.1 provides the MAE for 10-fold and resubstitution to evaluate the accuracy of the approach. The results suggest that any prediction has a margin of error around 1. In a scale of five this is a substantial number, however, this means that if the actual score of the painting is 4 (*like*), the margin of error is between 3 (*normal*) and 5 (*strong like*). We consider this, in conjunction with the confidence metrics, to be a good performance when attempting to capture collective understanding of beauty within the boundaries of our FM.

Table 11.1: Controlled assessment results.

	Resubstitution	10-Folds average
Mean absolute error	1.0523	1.0913

Artist perspective

The artist obtained and analyzed the ranking and the confidence metrics using the approach. He claimed that the collective as a whole stated their scores in a very coherent fashion according to the parameters of traditional and classical painting principles of perspective and contrast. For example in Figure 11.5b, the brightness in the sky found its counterpart in the brightness of the sea but only in the left side because of the mountain on the right. People understood that they were dealing with landscapes and they disliked the ones that tended to be flat or that they did not respect some of these principles.

The objective of the installation was to explore his painting style in a feasible way to leverage user feedback. The resulting ranking was very interesting to understand people’s sensibility about the possible configurations. The ranking showed him liked configurations with high global confidence that he had never considered and that he liked them too. The results of HSPLRank exposed him to novelty that, as added value, he considered that they have some guaranties of acceptance when exposing them to the public. For example, before this exercise, he painted mountains or sea but never together in the same composition. He considered that he has learnt many things about his own painting style as well as about the perception of the people about it.

Individualized evaluation

Regarding the previous controlled assessment, it was not feasible to bring the whole collective back for evaluating the created ranking. However, by conducting the experiment with only one person we can create a ranking with his or her own assessments and then evaluate the validity of our estimations with the person on site. The objective was to evaluate if the ranking successfully discriminated between the liked and disliked for the perception of each user. We selected 10 persons for this evaluation. Each user was voting in a session of 20 minutes. In average this duration corresponded to a data set of 16 populations (320 paintings). Once the ranking was created we took 10 liked and 10 disliked that were not shown during the evolutionary phase. Specifically, we took the first 10 paintings with a weighted mean score from 4.5 to 5 that had the highest global confidence. In the same way, we took the first 10 with a weighted mean score from 1 to 1.5 that had the highest global confidence. After splitting these 20 paintings randomly, we obtained an average of 91% accuracy in the prediction between like and dislike. The results suggest that the predictions in the extremes of the ranking are accurate in the case of individualized estimation.

11.3 Contact List case study

Contact Lists are widely used HCI applications to obtain personal information such as telephone numbers or email addresses. We can find them on mobile phones for personal use, communication systems for elderly people, corporate intranets or web sites. Despite of sharing the same objective, the final UI implementations are very diverse. In this case study we focus on corporate contact information.

11.3.1 An SPL for Contact List design alternatives

Figure 11.6 presents the FM defining the variability of the Contact List application domain. The FM was created by HCI experts from the Luxembourg Institute of Science and Technology (LIST) with whom we collaborated in this case study. This FM encodes knowledge about the interface design defining a configuration space of 1365 valid configurations. UI design choices, even for this apparently simple case, give raise to voluminous configuration spaces.

The **ContactList** variability is decomposed into three main features: **List** depicting the possible choices to be made in terms of widgets for representing the list, **Master Detail Interface** which states the global layout of the application and **Details Grid** which sets the layout for the detailed information of a person. The **ListType** variability defines the different alternatives of List widgets: **DropDownList** is a select box showing only one item when inactive, **ListView** is a classic navigation list and **TileList** is a list of thumbnails represented as tiles. The **Indexed** optional feature separates and ranks the list items by the first letter of the name. The **Filter** optional feature adds a search functionality implemented through a text box that automatically filters the list items according to the text introduced by the user. The **ListItem** consists of the **Name** of the person or the **Photo**, or both. Four

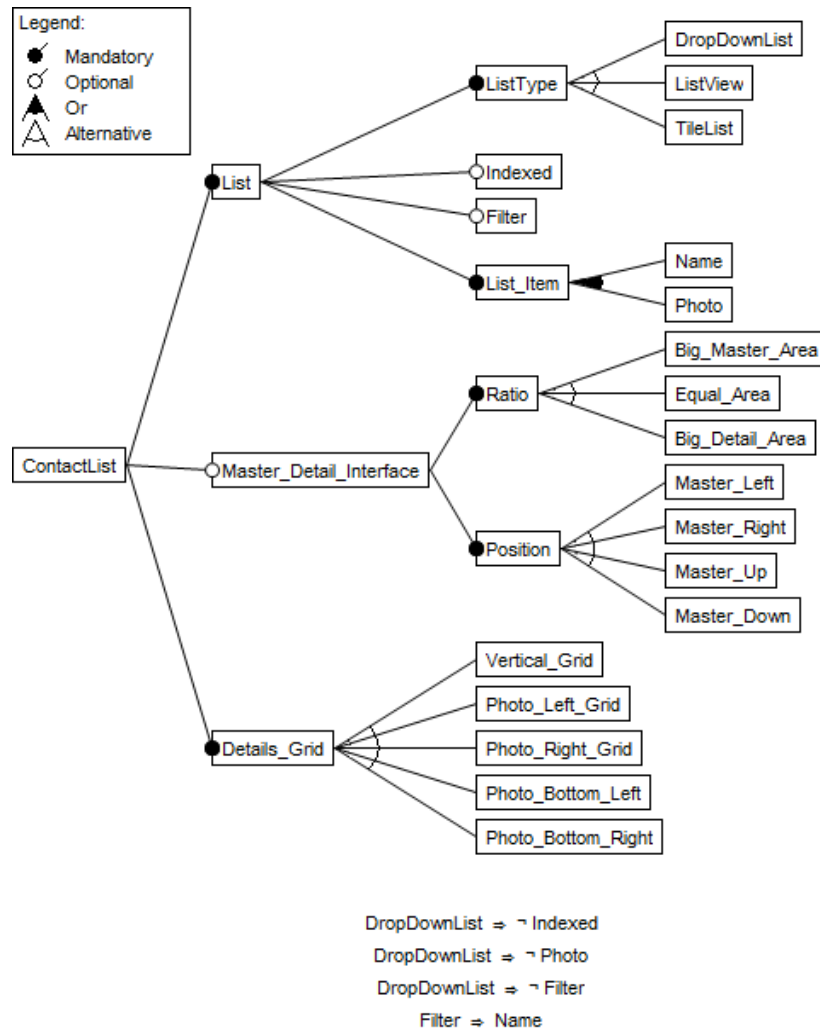


Figure 11.6: Contact List feature model.

cross-tree constraints are shown in the feature diagram which are related to these features. Concretely, the **DropDownList** feature excludes **Indexed**, **Photo** and **Filter** features. Also, the **Filter** feature requires the **Name** in the **ListItem** feature.

The **Master Detail Interface** is an optional feature that split the screen in two parts: the master and the detail. The master contains the list while the details interface, after a selection in the master, shows the corresponding contact information. There is variability concerning the **Ratio** of the screen split and the **Position** of the master interface. Finally, the **Details Grid** feature represents different alternatives to organize the contact information on the screen (e.g., telephone number, address etc.) as for example including all information in one column or determine the position of the textual information with respect to the photo.

We implemented the SPL using the Variability-aware Model-Driven UI design framework [SVGF15] based on AME (Adaptive Modeling Environment) [GFSV14] which is able to derive, through source code generation, any configuration of the presented FM. The target framework for the derived products is the JQueryMobile web frameworkⁱⁱ.

ⁱⁱJQueryMobile web framework: <https://jquerymobile.com>

We present screenshots to show the diversity of UIs that can be obtained. We decided to anonymize these screenshots to avoid displaying personal information. Figures 11.7 and 11.8 present UI variants from which we enumerate their corresponding features. Figure 11.7 shows an example that includes **ListView** with only the **Name** (see left side of the figure). The list view is neither **Indexed** nor has a **Filter** feature. It has **Master Detail Interface** with **Equal Area** and **Master Left** given that the screen is split in two identical parts with the list (master) at the left and the details grid at the right. The details are displayed with **Photo Right Grid**.

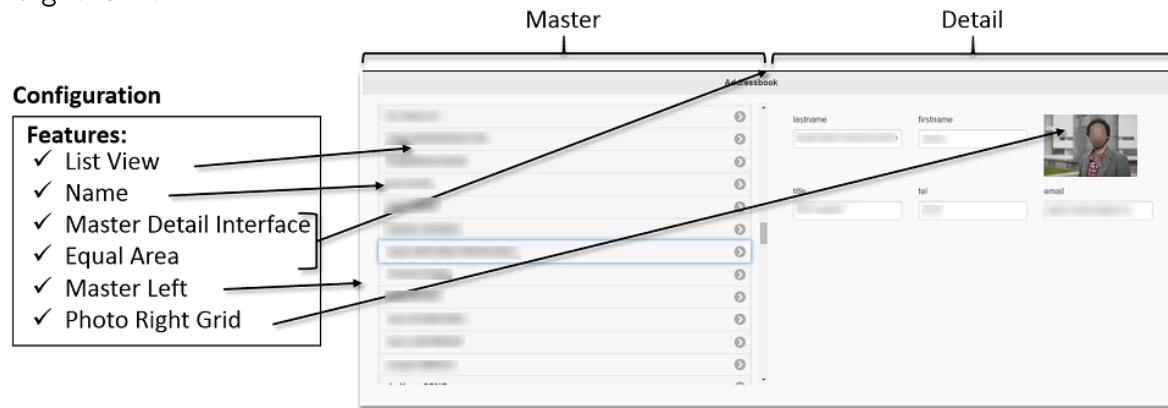
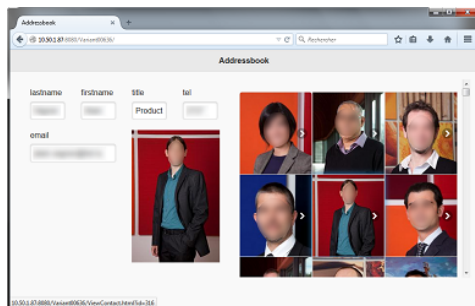
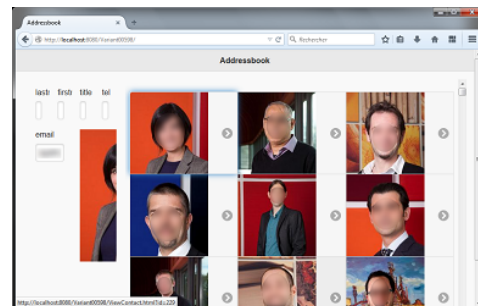


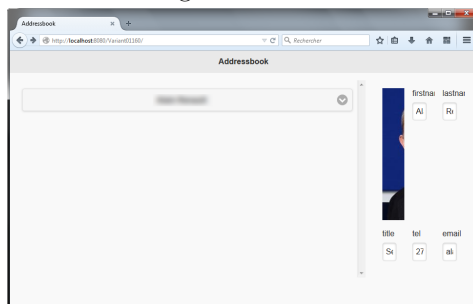
Figure 11.7: Configuration and screenshot of its associated Contact List UI variant.



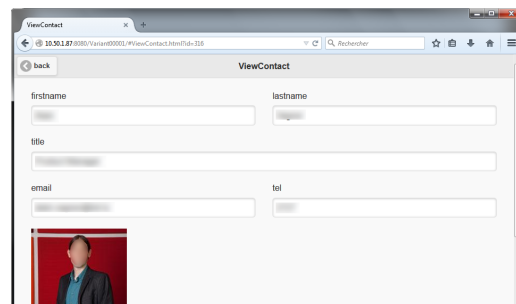
(a) Tile List, Photo, Master Detail (with Equal Area and Master Right) and Details Grid with Photo Bottom Right.



(b) Tile List, Photo, Master Detail (with Big Master Area, Master Right).



(c) DropDownList, Name, Master Detail (with Big Master Area, Master Left).



(d) Master Detail Feature (The list variability is not illustrated in this figure) and Details Grid with Photo Bottom Right.

Figure 11.8: Screenshots of derived variants from the Contact List SPL and enumeration of their associated features.

Figures 11.8a and 11.8b show how the **TileList** is realized (see right side of the figure) and Figure 11.8c how the **DropDownList** is displayed (see left side of the figure). Figure 11.8d shows a UI variant whose configuration does not have a master detail. It only displays on the screen either master or detail (note the presence of the back button at the top left of the screenshot for coming back to the master). In the case of master detail, the ratio indicates whether we have a big master interface with a small details interface (e.g., Figure 11.8c) or vice-versa. Alternatively, we can have the split into two equal parts (Figure 11.7 and 11.8a).

The **Position** variability is related to a horizontal or vertical split of the screen and whether the master is in one side or the other (in Figure 11.8b the master and detail have been swapped). If the **Master Detail Interface** feature is not selected in a configuration, the window split will be replaced during the navigation: one first window for selecting the person to be displayed and the other one for seeing the details. Finally, the **Details Grid** feature represents different alternatives to organize the contact information on the screen. For instance, in Figure 11.8a the grid is four columns and two rows whereas in Figure 11.8d it is two columns and four rows.

Objectives and settings

In order to know if HSPLRank enables to select relevant configurations from user assessments, we evaluated two hypotheses:

1. The variant assessment selects better configurations than a randomized algorithm for a given number of iterations. We quantitatively evaluated the improvement of the user scores through the IGA compared to random selection within the configuration space. We further investigated the diversity of the population along the generations to show the convergence of the IGA towards relevant UI designs.
2. The top positions of the ranking created with HSPLRank are configurations that usability experts confirm as relevant UI designs. We qualitatively discussed the findings with a usability expert and we checked if the top ranked configurations are close to configurations elicited by usability experts.

We deployed HSPLRank using the Contact List SPL in two organizations, the LIST, and the Interdisciplinary Centre for Security, Reliability and Trust (SnT) of the University of Luxembourg. The objective was to design their web-based corporate contact lists. We have set up independent experiments in these two different organizations in order to have more than one experimental result. In order to make the Contact List more realistic to the participants context, we fed the contact list database using the corresponding real contact information as publicly available in the organization websites. We excluded persons without photo. The contact list at LIST contained 276 persons and the contact list at SnT contained 154 persons. The difference in the number of persons is not significant for the tasks at hand since we did not measure the users time performance.

11.3.2 HSPLRank realization

Phase 1: Variant reduction

In order to reduce the number of variants, a usability expert from LIST was involved in defining the soft constraints. These soft constraints are used in both LIST and SnT organizations to define the viable space. The usability expert has 5 years of experience on usability analysis but no knowledge about SPL. We explained him the Contact List FM and the variability-aware UI design models as well as the formalisms and the concept of soft constraint. Then we let him try our configuration interface and SPL derivation in order to have a first grasp of the UI variability and the representation of the different features in JQueryMobile. Then he discussed the possible usability problems and established the following soft constraints for targeting these corporate contact lists:

- *soft*(\neg DropDownList): The drop-down list does not seem appropriate as it introduces an additional interaction step which is the expansion of the list items.
- *soft*(Index \vee Filter): It is required at least one functionality to facilitate the search. Searching persons on a list composed uniquely of name or photo is easier if the elements are ordered (indexed) or if we can search for a person name.
- *soft*(TileList \Rightarrow Photo): By essence, tiles are made for graphical representation of elements. As a result, having only a name in a tile list is not recommended.

From an initial configuration space of 1365 possible configurations, the introduction of the presented soft constraints reduced the viable configuration space to 715. This represents a 48% reduction of the possible configurations. Even with such a reduction of configurations, it is still too expensive to perform user assessments for all the 715 remaining configurations.

Phase 2: Variant assessment

We followed the recommendations of Nielsen regarding the minimal number of end users to involve in iterative user tests [NL93]. As such, we decided to run user tests with 5 participants in order to collect their feedback. According to the predictions of the Poisson model [NL93], involving 5 users gives rise to an expected probability of reporting 85% of the usability issues. Given that usability problems impact their assessments, we consider that 5 users allow to evolve the UIs at an optimal cost regarding the number of involved users.

At LIST, the age of the participants was between 26 and 36 with a mean of 32. Their role in the organization ranged from PhD students and R&D engineers to researchers. 4 out of 5 declared having previous experiences in UI design and 3 out of the 5 already designed UIs in an industrial context. At SnT, the age of the participants was between 25 and 33 with a mean of 31. Their role in the organization were 4 PhD students and 1 post-doc. They had limited knowledge of HCI design except one who has been also confronted to UI design in an industrial context for a short period of time. Participants from both organizations do not have sufficient practical years of experience to be considered as professional UI designers.

However, they already have been confronted with real-world problems in UI design and they have extensive experience on interaction with computers as end users.

We made the decision of using 10 configurations per generation of the IGA. Figure 10.5, presented in previous chapter, illustrates the iterations of the IGA to produce the different generations with 5 users and 10 configurations per generation. A bigger number of configurations will represent more diversity in the initial population but it will require a greater number of user assessments before obtaining the benefits of an evolutionary approach. On the contrary, a smaller number of population members will represent a very limited diversity. We considered 10 configurations to evaluate a complete generation as a number that keeps the balance between diversity and effort. Therefore, in each generation, the 5 users are provided with 2 variants to evaluate.

To carry out the study, we split the evaluation protocol into two different phases common to both groups. In the first phase, we introduced the test to the users, we explained the main steps of this test and we gave them a questionnaire to evaluate their profile and skills regarding UI design. In the second phase, each user had access to a web application that was driving the tests. For each UI assessment, the user received the task to accomplish that consisted in obtaining information from a particular person (randomly selected by our system) in the contact list. Then, the UI was displayed and the user was able to interact with it. At the end, the user finished the test by filling up a satisfaction form. This form, apart from standard usability related questions [Bro96], encompassed a global satisfaction note on a scale from 1 to 7 to capture the global impression. An extra free-text field was included to report comments and usability issues. When the test of one UI was completed, the system proposed the second user interface to test.

One important decision to implement the genetic algorithm is how to represent the individuals. We used an array, as in the previous case study presented in Section 11.2.1, but this time we consider a binary array. Figure 11.9 shows an example of the chromosome of an individual that conforms to the Contact List SPL. The phenotype consists of the non-abstract features of the FM. Concretely, the leaves of the FM and the **Master Detail Interface** feature (see Figure 11.6) are coded on a binary string of 20 bits. The features are the fixed indexes of the array where the value 1 means that the feature is activated and 0 that it is not. Representing a FM configuration chromosome as an array of bits is a common practice in the use of genetic algorithms in SPLE [EBG12, HPP⁺14].

DropDownList	0	1	0	0	1	1	1	1	0	1	0	1	0	0	0	0	0	1	0	0
ListView																				
TileList																				
Indexed																				
Filter																				
Name																				
Photo																				
MasterDetailInterface																				
BigMasterArea																				
EqualArea																				
BigDetailArea																				
MasterLeft																				
MasterRight																				
MasterUp																				
MasterDown																				
VerticalGrid																				
PhotoLeftGrid																				
PhotoRightGrid																				
PhotoBottomLeft																				
PhotoBottomRight																				

Figure 11.9: Example of the chromosome of an individual of the Contact List SPL.

The details of the implemented IGA is shown in Algorithm 4. First, the population is randomly initialized at line 1 taking into account the defined soft constraints (i.e., only viable configurations). After this, the evolution starts at line 2 until the stop condition (line 15) is satisfied. In our case we used the termination condition when reaching a fixed number of generations. We set it to 6 generations that correspond to a total of 60 variant assessments (12 UI variants per user). We decided that this amount will be sufficient and that more generations will be not feasible in terms of user-fatigue and time consumption. These 60 configurations correspond to only the 4.4% of the configuration space.

From line 3 to 5, each member of the population is assigned to one user for assessment. In our case study, each of the 5 users is assigned to 2 members to cover the whole population. The user feedback for all the population is obtained from line 6 to 9. Once the fitness of the whole population is set, we can proceed to the parent selection for the next generation. The *parent selection operator* is based on a fitness proportionate selection (line 10). At line 11, the *crossover operator* is based on the half uniform crossover scheme [Sys89]. The crossover, as well as the mutation operators of the GA, can end up with invalid configurations because of FM constraints. Existing works have solved this by penalizing the fitness function or trying to recover the configuration to a valid state. In our case, at line 12 and 14 we repair the offspring if hard constraints are violated. The *mutation operator* used at line 13 is uniform with $p = 0.1$. This mutation factor is meant to prevent a loss of motivation from users (i.e., user fatigue) by reducing the likelihood that they will keep assessing very similar UI configurations from the population, while thus enabling us to explore new regions of the configuration space. Finally, at line 15, the *survivor selection operator* is based on a complete replacement of the previous generation with the new generation.

The variants assessment phase was conducted independently for the two organizations collecting two separated data sets. The results of this phase will be presented in next section.

Algorithm 4 Interactive genetic algorithm for data set creation in the Contact List case study.

```

input: Genetic representation of a configuration = 20 bits, Population = 10 configurations, Users = 5
output: Data set of user assessments
1: population  $\leftarrow$  initializePopulation()
2: while stopConditionNotSatisfied() do
3:   for  $conf_i \in$  population do
4:      $conf_i.assignedUser \leftarrow$  assignUser(users,  $conf_i$ )
5:   end for
6:   for  $conf_i \in$  population, in parallel do
7:      $conf_i.fitness \leftarrow$  getUserFeedback( $conf_i$ )
8:     registerDataInstance( $conf_i$ )
9:   end for
10:  parents  $\leftarrow$  parentSelection(population)
11:  offspring  $\leftarrow$  crossover(parents)
12:  offspring  $\leftarrow$  repair(offspring)
13:  offspring  $\leftarrow$  mutate(offspring)
14:  offspring  $\leftarrow$  repair(offspring)
15:  population  $\leftarrow$  survivorSelection(offspring)
16: end while

```

Phase 3: Ranking creation

For the similarity distance between configurations, we used the Hamming distance, presented in Section 10.3.3. Figures 11.10 and 11.11 show the results of the 10-fold schema in the two organizations. A similarity radius with values less than 4.1 fails to cover the whole test set in both cases, and the exponential weighting approach outperforms the other weighting approaches. In this case study, we decided to use 4.1 as similarity radius and exponential weighting for the estimation of the configurations allowing the ranking creation.

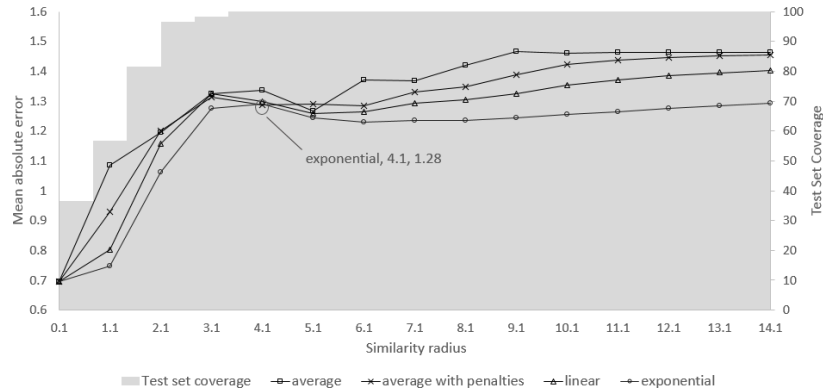


Figure 11.10: LIST: Mean absolute error for different radius values and weight calculation approaches.

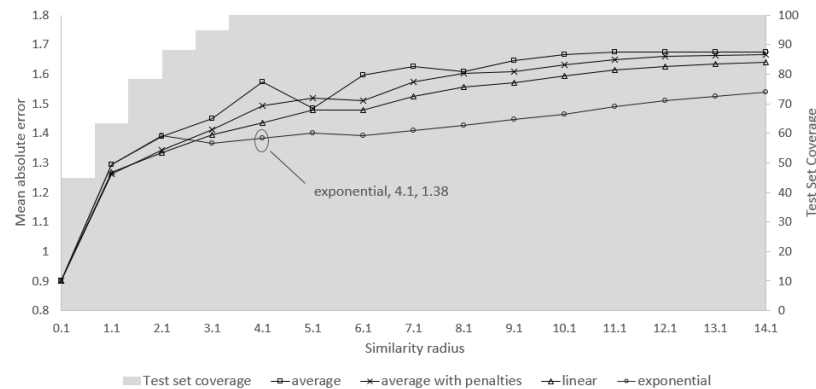


Figure 11.11: SnT: Mean absolute error for different radius values and weight calculation approaches.

A manual examination of the top-ranked UI designs for both organizations revealed three relevant UI designs shown in Figure 11.12.

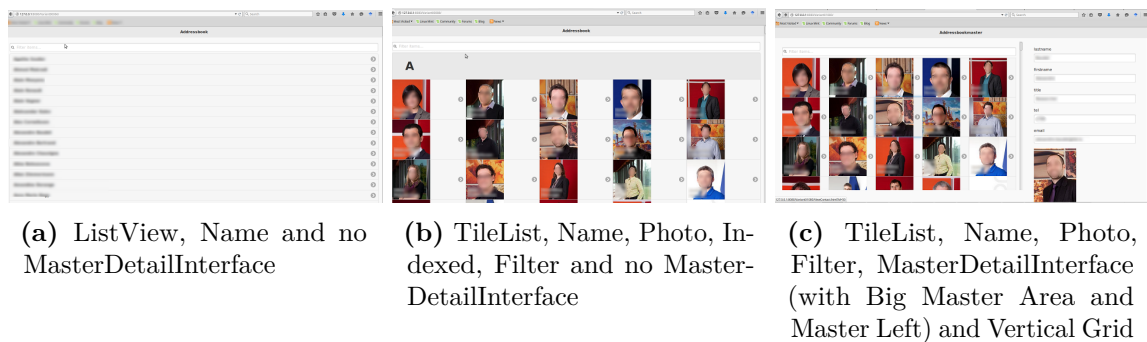


Figure 11.12: Screenshots of relevant variants found using HSPLRank.

11.3.3 Evaluation

This section presents the results after applying HSPLRank and discusses the objectives of this case study presented in Section 11.3.1. Figure 11.13 presents the results of the IGA for the two organizations. On the horizontal axis we have the different generations and the vertical axis is the mean of the scores of the user assessments for this generation. We show the score mean along the six generations including the standard deviations. An ascendant progression means that for each new generation, globally, the UI variants are being better appreciated by the pool of users. In Figure 11.13a we observe a quick ascension until generation four while in Figure 11.13a we observe the ascendant progression starting at generation two.

Despite that we do not have the explanation for the descending effect in LIST case for generations five and six, we consider that it is caused by the experience that the users had in UI design. The score mean improved quickly from generation one to four by filtering really inappropriate variants, and then decreased a little bit because of their capacity to criticize the proposed variants. They may not evaluate the variant itself but its capability to be different from what they already evaluated. Furthermore, some of these critics were related to non-variability related issues as we will discuss in Section 11.4. Other possible explanation could be user fatigue.

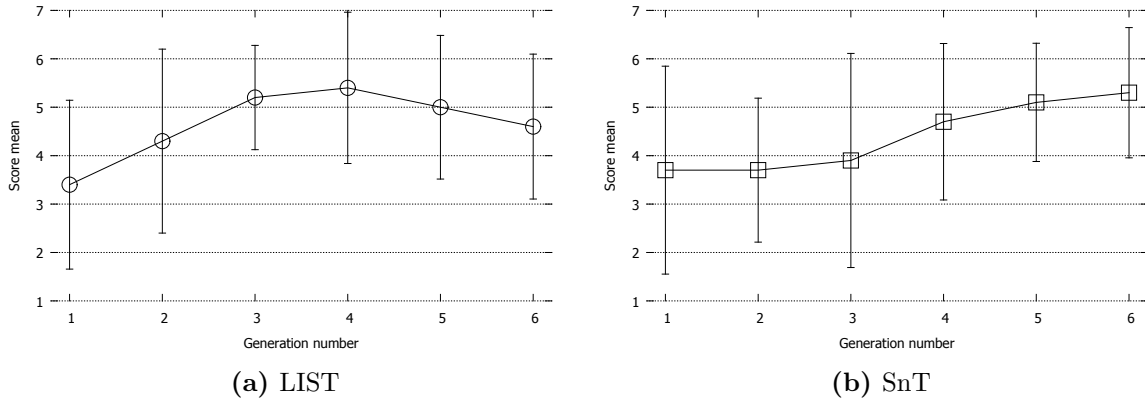


Figure 11.13: Results of the genetic algorithm evolution in the two organizations.

In order to observe if the IGA tends to converge, Figure 11.14 shows the progression of the generations in the two dimensional space of mean score and diversity. We calculated the genotype diversity along the different generations ($g1$ to $g6$) as the average of the Hamming distance of all pair of configurations in the generation. The diversity decreases if we approach to the left side of the horizontal axis. For example, we can observe how the diversity is not increasing more than its value at $g1$ which is the randomly created population. For LIST, as shown in Figure 11.14a, $g4$ has both the lowest diversity and the maximum mean score. In the case of SnT, as shown in Figure 11.14b, the last generation ($g6$) has both the lowest diversity and the maximum mean score. In the LIST case, the user pool was able to reach better variants for them earlier.

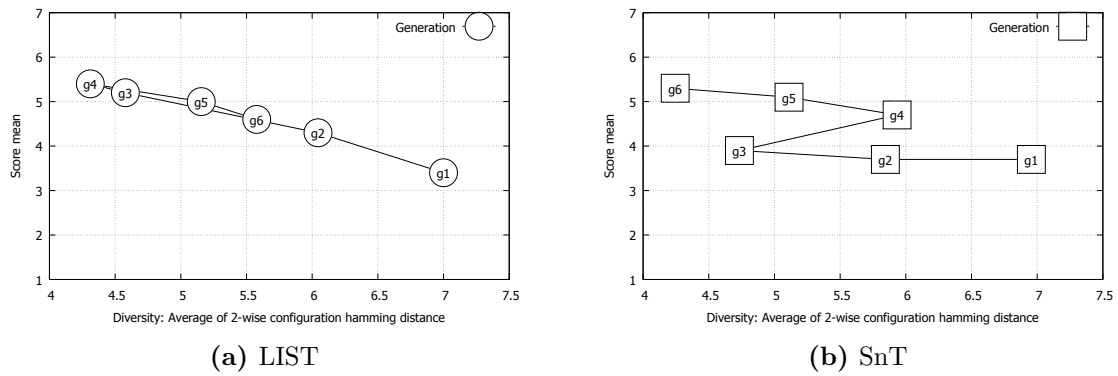


Figure 11.14: Generations progress in terms of mean score and diversity.

UI quality improvement

Regarding the first hypothesis, we evaluate if our process based on evolutionary techniques selects better variants than a randomized algorithm for a given number of iterations. We repeated the experiments with the same participants using random selection. In this approach, for each generation, 10 configurations were automatically selected from the viable space which is the same size of the population that we used for the IGA. Basically, for the random selection, we used the same operator as the one used for seeding in the IGA. Despite that we still call each group of 10 random configurations a generation, no genetic information was propagated from one generation to the next.

Figure 11.15 shows the results of the genetic algorithm and the random selection in order to compare them. The most important observation is that random selection failed to obtain a global score mean greater than 5 in any of the generations while the genetic algorithm did achieve it. We can see how the genetic algorithm outperforms the random selection approach except in the first two generations where the effect of evolution is still trying to find relevant regions of the configuration space.

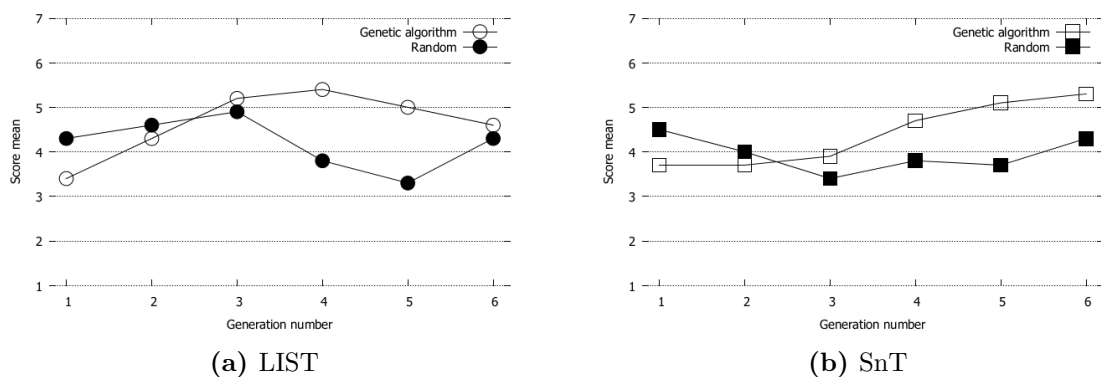


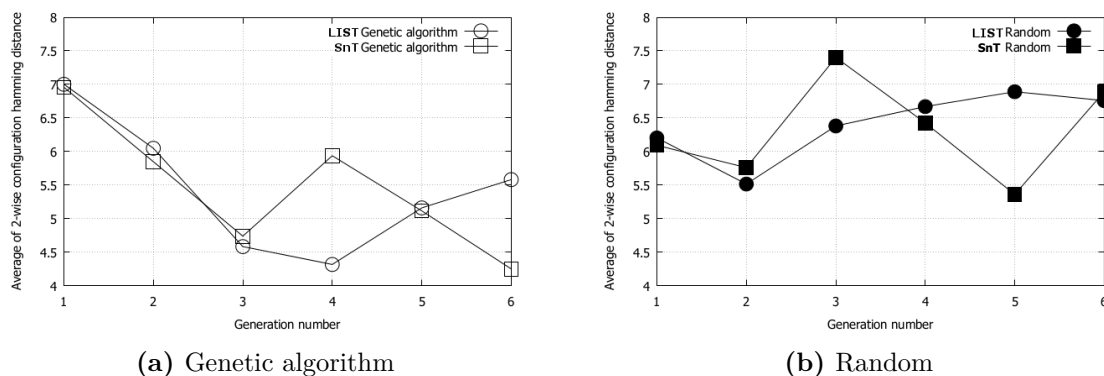
Figure 11.15: Comparing variant selection based on genetic algorithm and random.

Table 11.2 presents the representative improvements obtained in the two independent experiments by comparing the global score mean. The global score mean is the mean of the assessment scores in all the generations. The genetic algorithm approach has a global score mean which is around 0.5 points better (i.e., 0.45 in LIST and 0.55 points in SnT).

Table 11.2: Global Score Mean evaluation.

	GA	Random
LIST	4.65	4.20
SnT	4.40	3.95

We have seen that the proposed genetic algorithm got better results over the generations than the random approach, and now we show that the algorithm tries to converge in this search of better UI configurations. To show this, we calculated the diversity of the members of each generation. If the diversity has a tendency to decrease, it is a sign of convergence. Figure 11.16 shows the graph of the results at organization LIST and SnT for both the genetic algorithm and the random process. We can see how the random approaches in both organizations do not decrease the diversity while, for these 6 generations, we observe how the genetic algorithm performs better than the random approach to reduce the diversity. The random approach failed to decrease the diversity to values lower than 5 while this was achieved by the genetic algorithm approach. As result, we can conclude that, compared to the random approach, we both increase the global mean score and we reduce the diversity along the generations. These two aspects allow the genetic algorithm to try to converge to optimal or suboptimal solutions which means to relevant UI designs.

**Figure 11.16:** Genotype population diversity.

Analysis of the relevant variants by a usability expert

In order to confirm our second hypothesis we required a usability expert with nine years of experience to assess that the better variants found by HSPLRank satisfy usability criteria. This expert is independent in order to provide an impartial assessment. He does not belong to the team that developed the considered project, nor participated during the variant assessment, nor defined the soft-constraints in the variant reduction. We summarize the expert qualitative evaluations on the three HSPLRank relevant variants shown in Figure 11.12:

- The first variant, shown in Figure 11.12a, is the simplest list with no master detail. It satisfies many usability criteria [SB97] such as low workload, explicit control, homogeneity/consistency or compatibility with traditional contact applications. The search bar and the simplicity of the UI allows the end user to go directly to what he/she is looking for. However, as drawback, it is not possible to browse through the contacts' photos or to do a visual research if the name of the person is unknown.

- The second variant, shown in Figure 11.12b, has a better appearance (aesthetic consideration) and has more information (i.e., photo and index). It also complies well with Scapin and Bastien's usability criteria [SB97]. Notably the adaptability criteria is well implemented here: the application can be convenient to the different situations of use (e.g., on large screen and small screen display, etc.). However, it seems visually overloaded. Reducing the number of persons displayed in the list can be an option. Another important point noticed is that the users can just play with the UI (e.g., browse through colleague photos) and be distracted from the prescribed task.
- The third variant, shown in Figure 11.12c is very close to the previous one, except for the master/detail pattern. It also complies with most of the usability criteria. The list of persons is more compact than the previous variant (Figure 11.12b) giving a better impression. The information is accessible directly without the need to navigate which is a plus for large screens but not necessarily the best solution. In the configuration with a Master Detail interface, the layout is important, and in this variant the vertical grid fits perfectly with this layout.

The usability expert claimed that the relevant variants that have emerged from applying the HSPLRank approach satisfy most of usability criteria.

Comparing HSPLRank results with usability expert choices

We asked the usability expert to create configurations of the Contact List using strictly expert judgement. HSPLRank approach is intended to explore design alternatives using the users as the stakeholders to make the last decisions about the correct alternatives. However, we wanted to compare the alternative that a usability expert will determine against the obtained relevant variants with our approach.

Regarding the different configuration possibilities, the usability expert suggests, according to his opinion, two UI variants that are the two best possible configurations regarding global user experience (i.e., including usability, but also aesthetic and functional aspects). The configuration of these two variants are the following: 1) **TileList, Indexed, Filter, Photo, Name** and 2) **ListView, Indexed, Filter, Name**. The first variant is a complete UI with aesthetic aspects first while the second is simpler and efficient in its design. For the contact information detail layout (**Detail Grid**), two are possible: the **Vertical Grid** which gives the best organization of information for large screen size and the **Photo Left** which gives the best organization for smaller screen size.

As we can observe, the configurations elicited by the expert are very similar to the relevant ones shown in Figure 11.12. We calculated the Hamming distance between each of the best configurations given by HSPLRank and the ones provided by the usability expert. We found that the minimal distance between the expert variants and the two best configurations of HSPLRank is 1. For the last one, the distance is 7. This can be explained by the inclusion of a **MasterDetail** feature in this configuration that increases the Hamming distance by 3. As conclusion, the results of HSPLRank do not diverge from the usability expert judgement but in our case, the expertise emerges from the assessments of a set of potential end users.

11.4 Discussion

This section discusses certain general aspects about our experiences with HSPLRank in the paintings and the contact list case studies.

Quality of the Soft constraints

In the paintings case study, it was decided that no soft constraints will be introduced. However, in UI design, trying to formalize usability issues in the variability is a difficult manual task. The variants reduction phase performed by usability experts seems to need further support. In our current approach this is a manual task which may not be scalable on very large FMs. The usability expert defined the soft constraints presented at Section 11.3.2 but, according to *a posteriori* analysis of the results, the inclusion of other soft constraints could have prevented some features or feature combinations that were highly undesired. For example, a soft constraint setting the feature **Name** as mandatory. When the feature **Name** was not present, 83% percent of the users were not able to complete the task because they needed either to know the person or to guess the person name through the photo. As result, they could not finish the task which had a great negative impact on their assessment.

The inclusion of the soft constraint *soft*(**Name**) would have reduced the viable configuration space from 715 to 585 possible configurations which is a 18% reduction. In addition, the initial population of the IGA would have ignored these configurations. Other relevant features that affected negatively to the user scores were the absence of **Filter** and layout issues. If both *soft*(**Name**) and *soft*(**Filter**) were added in the variants reduction phase, the viable configuration space would be reduced to 390 configurations which is a 45% reduction.

Non-variability factors during assessment

Figure 11.17 presents a categorization of user comments on issues found in the contact list usability. The presented usability issues, at both LIST and SnT, are the ones that conditioned their scores in the variants assessment process. We observe how 21% of the comments in the LIST experiment are related to non-variability factors. These segments are shown in Figure 11.17 in black. We call non-variability factors to elements of the variants that can not change because they are not included in the FM and, therefore, they can not be presented differently in the products implementation. Given that the users assess the variants as a whole, they are not aware about the elements that can change. However, these elements can impact the results of the evolution as the users can negatively assess a variant due to elements that do not have the possibility to change in next generations.

From this 21% of non variability related issues, some examples are the size of the picture in the list. They consider that the pictures should be smaller. Another example is that they would like auto-focus in the search field when the user opens the Contact List UI. Some of these reported issues can be fixed independently of variability, like the auto-focus for the **Filter** feature. Others could be registered and added as variability for further evaluation (e.g., the picture size big or small). The usability expert also suggested some enhancements

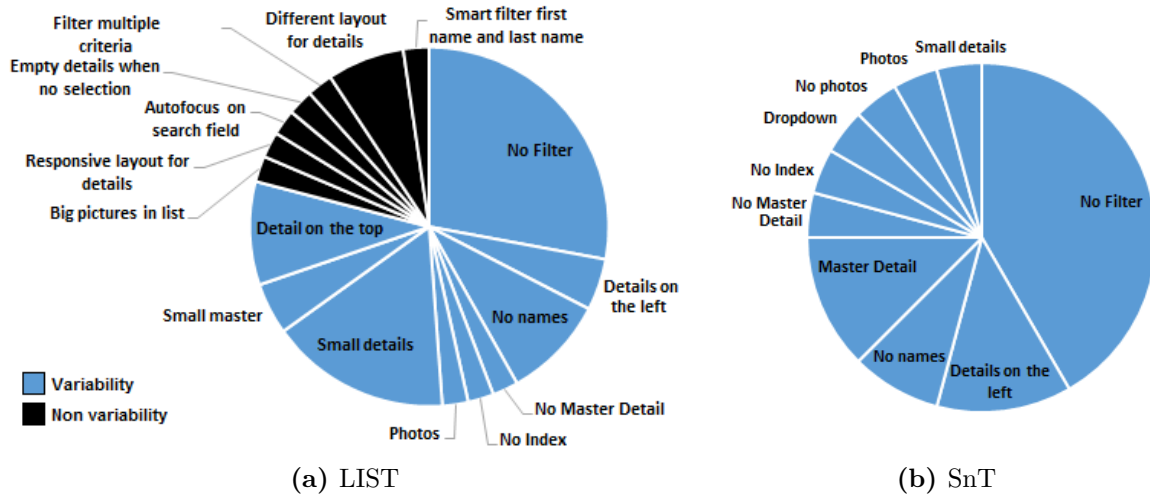


Figure 11.17: Usability issues reported by users.

that were not present in the planned variability (e.g., the order of the telephone and mail fields in the detail view).

At SnT, all the reported usability issues are related to variability. In the current evaluation of HSPLRank, we can not state if it had an impact. How to manage the reported issues is out of the scope of HSPLRank. However, we suggest to consider and study the impact of non variability related issues in future works applying IGAs in Human-centered SPLs to further understand its impact.

The importance of feature combinations

Due to actual user feedback, the IGA favoured some features that appeared frequently in products with high scores. Figure 11.18 illustrates the impact of the IGA to the middle parts of the paintings case study. For example we obtained less user assessments in configurations that have M_5 as the evolution found it less adapted while we have more data set instances containing other middle parts. However, as we can see in the figure, the score distribution for these Middle parts are quite similar. This also occurs for **Sky** and **Ground** features as well as for **Flip** and **ExtraSize** features of each part. This shows that no feature by itself has a great impact on the voting.

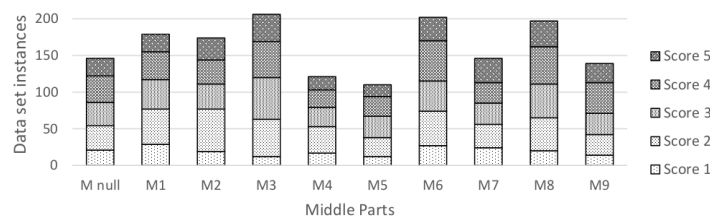


Figure 11.18: Relating middle parts with the number of data set instances and the scores.

We applied data mining attribute selection algorithms in an attempt to discriminate between relevant and irrelevant features. Concretely, we evaluated the worth of each feature by computing the value of the chi-squared statistic with respect to the class (i.e., a score in the range of 1 to 5). The ranked features showed that the alternatives for **Ground**, **Sky** and

Middle parts were more relevant features than `MiddleFlip`, `GroundExtraSize`, `GroundFlip`, `SkyFlip`, `MiddleExtraSize` or `SkyExtraSize`. After applying *a priori* association discovery algorithms to discover relations between features and the scores, we found no rules even by setting very low levels of confidence. We then used a classification model to explore the issue attempting to predict the score of a combination of features. We relied on decision trees which led to incorrectly classifying 80% of instances in the test sets. Linear regression implementations also performed poorly.

These findings suggest that the collective understanding of relevant variants in this context was truly built upon highly subjective opinions and that only combinations of features were relevant in user assessments. On the contrary, in the contact list case study, as discussed in Section 11.4 regarding the importance of appropriate soft constraints, there were some features, like `Name of Filter`, that had a great impact on the user assessments.

11.5 Threats to validity

In this section we categorize the threats to the validity of HSPLRank.

Dealing with subjective assessments

The inherent subjectivity of ratings is an important threat. As presented by Martinez *et al.* [MYH14], one threat is the non-linearity of the scores scale. HSPLRank considers the rates as numbers in order to summarize the variant assessments for calculating the score means. In rating scales, for example from one to five, depending on the person, the distance from one to two may not be the same as the distance from two to three. This is related to personal and cultural factors. The conversion of numerical values to nominal values is an alternative for the ranking creation phase that is worthy to explore and compare [MYH14].

Also, using the contact list case study as example, one user can justify that photos distract him from the task of finding someone while other user find it useful or appealing. This fact is related to different opinions which are contradictory. Even the same person assessing the same UI in two different moments can report different scores. Another problem is the already mentioned user fatigue. It should be possible to mitigate these threats by increasing the number of users in our tests and by establishing a better distribution of the testing workload between users. Embedding the IGA in an online crowdsourcing platform can be envisioned.

HSPLRank operators

During the variant assessment and ranking creation phases of HSPLRank there are automatic operators that condition the results of the approach. We should investigate different similarity distance metrics between two configurations. For example, for the paintings case study, algorithms based on image difference metrics or on distance matrices for each of the features could be explored. In our opinion, general similarity distance metrics like the Hamming distance can be used but it is worthy to explore domain-specific ones. We should investigate different genetic algorithm operators for the same case studies to try to find the optimal

settings. Also, we should compare our results to other approaches not relying on genetic algorithms. Also, other methods for empirically selecting the similarity radius and approaches for weighting the scores should be investigated.

The principles and techniques of the presented approach are repeatable for any case study dealing with user feedback on SPL variants. However, this approach will not scale in the ranking creation phase for FMs that can produce large amount of possible configurations. To solve this, instead of calculating the weighted mean score for all the possible configurations, we will investigate on non-euclidean centroid-based approaches or filtering mechanisms to restrict the calculation to a feasible amount of configurations.

Generalization of the findings

Another threat to validity is the need of more experiments that can further support our claims and confirm our hypothesis. Notably, we still need to understand the impact of non-variable elements. Variability in UI design can be manifold and in the experiments presented in our case study we only tackled some facets of this variability (i.e., layout, widgets) that are related to design alternatives. However, variability can be defined for different interaction devices, interaction contexts, graphical frameworks, etc. We consider that our approach can be used for any kind of UI variability but more experiments will be needed in this sense. Also, our case studies may be considered as medium size experiments in terms of the size of the configuration space.

11.6 Conclusion

HSPLRank can be applied in SPLE application domains where HCI components play an important role in the final products. In an SPL-based computer-generated art scenario, artists can use HSPLRank to understand people perception of their work, to inspire and refine their style, and eventually help them in the decision making process to select the variants that have more guarantees of collective acceptance. In a UI design scenario, design alternatives are assessed for trying to select the most adapted. We apply and validate the approach on two Human-centered SPL: an SPL-based computer-generated art system dealing with landscape paintings, and a contact list SPL where we aim to find the best UI design alternative for two different institutions.

During the case studies we have evaluated that 1) the interactive genetic algorithm performs better than random selection of the variants to be assessed, 2) the estimations and predictions provided by HSPLRank are acceptable using a 10-fold cross-validation and using individuals to assess variants that were not in the data set, and 3) domain experts confirms that variants exposed by HSPLRank are actually relevant.

The results of the presented case studies are promising and, as further work, enhanced versions of the approach could be applied to other case studies. There are also open research directions to extend HSPLRank. For example, it is interesting to learn users preferences *during* the execution of an IGA. When the prediction of HSPLRank is able to estimate

the user assessment with high confidence, we can skip it and use the next artefact variant. There are already some works in this direction, Liapis *et al.* [LMTY13] focused on creating computational models of user preferences. Also, Hornby and Bongard [HB12] and Li [Li12] proposed to learn to predict user aesthetic preferences. These will reduce the need for user involvement or at least it will allow to speed-up the evolutionary process.

Part VI

CONCLUSIONS

12

CONCLUSIONS

Contents

12.1 Summary	196
12.2 Open research directions	197

12.1 Summary

Software Product Line Engineering (SPLE) is a mature approach to manage a family of product variants with proven benefits in terms of quality and time-to-market. Unfortunately, SPLE demands a profound culture shift to an scenario where products should not be developed with a single-product vision. SPLs promote a centralized system that manages variability and exploits reusable assets to satisfy a wide range of customers. Before considering the high up-front investment of SPL adoption, it is often the case that companies have artefacts created with ad-hoc reuse practices, such as copy-paste-modify, to quickly respond to different customers at the expense of future maintenance costs for the artefact family as a whole. With the objective to mine and leverage artefact variants, Parts II, III and IV contributed to the advancement of extractive SPL adoption.

In Part II we presented Bottom-Up Technologies for Reuse (BUT4Reuse) which is a generic and extensible extractive SPL adoption framework that helps in chaining the technical activities for obtaining a feature model and reusable assets. Given that SPLs are being used in many application domains beyond source code artefacts, we presented the principles of adapters to support different artefact types. We described the ones available and a detailed description of the use of BUT4Reuse in an scenario with model variants which is considered a relevant domain in software engineering. Our approach, named Model Variants to Product Lines (MoVa2PL) enables to extract Model-based SPLs.

Part III is dedicated to help researchers in the field of extractive SPL adoption by providing the means for intensive experimentation. Concretely, we focused on providing a benchmark for feature location techniques using Eclipse variants (EFLBench). Therefore, we provided a method to obtain the ground truth from any set of Eclipse variants. The benchmark integration in the extensible BUT4Reuse framework makes easy to launch and compare feature location techniques. In addition, we provided a method for automatic generation of variants enabling the parametrization of some characteristics of the benchmarks. We also presented a method to identify family of variants of Android applications in large repositories (AppVariants) for experimentation with feature identification techniques. By applying our method we were able to discuss recurrent cases found when performing feature identification, such as libraries reuse, feature-based generated applications, content-driven variability and device-driven variability.

In Part IV we focused on assistance for domain experts by presenting two visualisation paradigms to support extractive SPL adoption activities. The first one is related to feature naming where word clouds are leveraged to summarize the implementation elements inside the identified blocks (VariClouds). The second one is a visualisation to help during feature constraints discovery that is built by mining the existing configurations (FRoGs). The usage of this visualisation can go beyond extractive SPL adoption and can be used for analysing feature relations (e.g., soft constraints) or as support for product configuration.

In Part V, we tackled an important topic in SPLE not directly related to SPL adoption. We presented an approach to predict and estimate end user assessments in artefact variants in order to rank the configuration space and help in selecting the most relevant artefacts (HSPLRank). We conducted experiments with an SPL-based computer generated art system dealing with high subjectivity in assessments as well as with an SPL of user interface design alternatives.

12.2 Open research directions

In the conclusions of each chapter we already suggested research directions which can be considered short-term goals. In this section, we present the long-term goal which considers the synergies among the different parts of this thesis. The challenge is to advance towards:

Extractive SPL adoption in the wild assisted by visualisation and end user assessment

In Chapter 7, we presented the identification of families of artefacts in the wild. These families are interesting candidates for extractive SPL adoption because of their shared implementation elements. However, with more advanced techniques, we should be able to consider not only similarly implemented artefacts, but also those that, belonging to the same domain, were implemented independently from each other. We presented in Section 4.5 that semantic comparisons have limitations. The analysis and unification of artefacts in overlapping domains has been studied, e.g., in the field of domain-specific languages [DCB⁺15].

Even if semantic comparisons find interesting software parts, there is still many technical difficulties in their integration (e.g., different interfaces or architectures). To cope with these challenging scenarios, we can bring to extractive SPL adoption the advances in *automated software transplantation* [BHJ⁺15]. The objective would be to transplant feature implementations from a donor in the SPL by extracting reusable assets. This might be tackled through a combination of extractive and reactive SPL adoption [AMC⁺07, Kru01]. In this context, finding and *analysing previous integrations* of a similar feature in other products could help during the integration process of this feature in the SPL. These techniques were already explored for advising in library migration in single systems [TFPB14]. *Mining version control systems*, which have been proposed to identify and locate features [LRC15], can help also to analyse the changes needed in previous integrations of features. This envisioned process is complex. As a consequence, providing *dedicated interactive visualisation paradigms* for domain experts is a challenging task.

In industrial scenarios, when merging companies of the same domain, each one may have its own software product, which gives rise to long integration projects. In some cases, as in the reported case of eight companies in the home rental market [KCB08], they may be willing to extract an SPL instead of building a “one-size-fits-all” solution. Apart from this scenario, we can also leverage artefact variants from public software repositories to extract an SPL targeting a specific domain. We consider that, mining repositories to start extractive SPL adoption in a given domain, create opportunities to provide highly customizable products.

As an illustrative example, Github or markets of mobile apps host hundreds of *task list* (“to-do list”) products showing great diversity in terms of features. Apart from adding tasks, some have the feature to mark tasks as repeatable (e.g., weekly), add a map location, add a date, assign colors, associate alarms or speech-to-text to create tasks, to name a few. Extracting a to-do list SPL would enable to create tailored solutions. Finally, it could happen that, having an operative SPL, we want to analyse the domain in the wild to discover and integrate features that might be of our interest. Remarkably, this opens new research directions in terms of SPL scoping [Sch02].

In this vision, user assessments are important at the beginning of the extractive SPL adoption as well as during the process. Firstly, we need to gather information about the products that will be used for extracting the features. Concretely, user assessments about these products will give us confidence about the *mined artefacts quality* [KGB⁺14] and will help in prioritizing feature donors. In this direction, we can analyse user reviews or project development activity. Secondly, automated software transplantation is strongly based in the existence of automatic tests, which are sometimes difficult to obtain. Transplantation was evaluated from one product to another but, in SPLE, feature interactions must be taken into account representing new challenges for SPL testing. Because of this complexity, during the extractive SPL adoption process, user assessments can be exploited through *crowdsourcing campaigns* with *interactive evolutionary approaches* in the background to guide in creating valid products. Assuming that valid products emerge because of the evolutionary process, the user assessments would then be based on usability issues. Therefore, in our opinion, in the present dissertation we have dealt with all the pieces towards the extractive SPL adoption of the future, i.e., harmonization, experimentation, visualisation and estimation.

LIST OF PAPERS, TOOLS & SERVICES

Papers included in the dissertation:

- 2016:
 - Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Name suggestions during feature identification: The VariClouds approach. In *Proceedings of the 20th International Conference on Software Product Lines, SPLC 2016, Beijing, China, September 19-23, 2016*
 - Li Li, Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Mining families of Android applications for extractive SPL adoption. In *Proceedings of the 20th International Conference on Software Product Lines, SPLC 2016, Beijing, China, September 19-23, 2016*
 - Jabier Martinez, Tewfik Ziadi, Mike Papadakis, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Feature location benchmark for software families using Eclipse community releases. In *ICSR*, volume 9679 of *Lecture Notes in Computer Science*, pages 267–283. Springer, 2016
- 2015:
 - Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Bottom-up adoption of software product lines: a generic and extensible approach. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 101–110. ACM, 2015
 - Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Automating the extraction of model-based software product lines from model variants (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 396–406. IEEE Computer Society, 2015
 - Jabier Martinez, Gabriele Rossi, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Estimating and predicting average likability on computer-generated artwork variants. In *Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings*, pages 1431–1432. ACM, 2015
- 2014:
 - Jabier Martinez, Tewfik Ziadi, Raúl Mazo, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Feature relations graphs: A visualisation paradigm for feature constraints in software product lines. In *Second IEEE Working Conference on Software Visualization, VISSOFT 2014, Victoria, BC, Canada, September 29-30, 2014*, pages 50–59, 2014
 - Jabier Martinez, Tewfik Ziadi, Jacques Klein, and Yves Le Traon. Identifying and visualising commonality and variability in model variants. In *Modelling Foundations and Applications - 10th European Conference, ECMFA 2014, Held as Part of STAF 2014, York, UK, July 21-25, 2014. Proceedings*, pages 117–131, 2014

Papers not included in the dissertation:

- Jabier Martinez, Jan Malburg, Tewfik Ziadi, and Görschwin Fey. Towards analysing feature locations through testing traces with BUT4Reuse. In *Workshop on Design Automation for Understanding Hardware Designs (DUHDe)*, 2015
- Amine Lajmi, Jabier Martinez, and Tewfik Ziadi. DSLFORGE: textual modeling on the web. In *Demos@MoDELS*, volume 1255 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2014

Papers published prior PhD:

- Jabier Martinez and Anil Kumar Thurimella. Collaboration and source code driven bottom-up product line engineering. In *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 2*, pages 196–200. ACM, 2012
- Haitham S. Hamza, Jabier Martinez, and Carmen Alonso. Introducing product line architectures in the ERP industry: Challenges and lessons learned. In *Software Product Lines - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Workshop Proceedings (Volume 2 : Industrial Track)*, pages 263–266. Lancaster University, 2010
- Jabier Martinez. Generation language - enabling scalability for product realisation. In *Software Product Lines - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Workshop Proceedings*, pages 211–216. Lancaster University, 2010
- Jabier Martinez, Cristina Lopez, Aitor Aldazabal, Jason Mansell, and Marta del Hierro. Plum (product line unified modeller). *13th International Software Product Line Conference, SPLC, Tool demo*, 2009
- Jabier Martinez, Jessica Díaz, Jennifer Pérez, and Juan Garbajosa. Software product line engineering approach for enhancing agile methodologies. In *Agile Processes in Software Engineering and Extreme Programming, 10th International Conference, XP 2009, Pula, Sardinia, Italy, May 25-29, 2009. Proceedings*, volume 31 of *Lecture Notes in Business Information Processing*, pages 247–248. Springer, 2009
- Jabier Martinez, Cristina Lopez, Estibaliz Ulacia, and Marta del Hierro. Towards a model-driven product line for web systems. *5th Model-Driven Web Engineering Workshop, MDWE*, pages 1–15, 2009
- Begoña Losada, David Lopez, and Jabier Martinez. Intergram, an user-centered design process. In *9th European Conference for the Advancement of Assistive Technology, AAATE 2007, October 3-5, San Sebastian, Spain*, pages 786–790, 2007

Software developed during PhD:

- *BUT4Reuse*: Bottom-up Technologies for Reuse: Extractive SPL Adoption
<http://but4reuse.github.io>

Services:

- Workshop organizer
 - Reverse Variability Engineering workshop (REVE)
4 editions at SPLC 2016, SPLC 2015, SPLC 2014 and CSMR 2013
 - Knowledge-Oriented Product Line Engineering workshop (KOPLE)
3 editions at SPLC 2012, SPLC 2011 and SPLASH OOPSLA 2010
- Program Committee Member
 - SPLASH 2016 OOPSLA Artifacts Program Committee (AEC)
 - INFOS 2016 10th International Conference on Informatics and Systems
 - SPLASH 2015 OOPSLA Artifact Evaluation Committee (AEC)
 - VARY workshop (Variability for you) at MoDELS 2012
- Session chair: International Conference on Software Reuse (ICSR) 2016 Tool demonstrations
- Conference co-reviewer: MoDELS 2016, ICSE SEIP 2016, SBES 2016, ICWS 2015, COMPSAC 2014, MoDELS 2014
- Workshop co-reviewer: 4th Workshop on View-Based, Aspect-Oriented and Orthographic Software Modeling (VAO) 2016
- Invited reviewer: AMECSE 2016, HICSS-46 2013 and Springer Requirements Engineering journal: Special Issue on Requirements Engineering in Software Product Lines, 2013
- Contributor
 - Tutorials for the Repository of teaching material for product lines and variability of the FAMILIAR project. <http://teaching.variability.io>
 - Enhancements, bug reports and fixes for FeatureIDE Eclipse plug-in for Feature-Oriented Software Development. <https://github.com/FeatureIDE>
- Project proposal: The BUT4Reuse framework will be part of the REVaMP² project (ITEA 3 Call 2): Round-trip Engineering and Variability Management Platform and Process.
<https://itea3.org/project/revamp2.html>
- General public
 - Luxembourg Science Festival 2015 workshop organizer: Evolving dinosaurs, evolving with computers. <https://www.science-festival.lu>
 - RTL TV show participation: Mister Science, logistics challenge. <http://science.lu/de/content/logistik-challenge-mr-science-vs-livreur>

BIBLIOGRAPHY

- [AAG⁺14] Mathieu Acher, Mauricio Alf  rez, Jos   A. Galindo, Pierre Romenteau, and Benoit Baudry. Vivid: a variability-based tool for synthesizing video sequences. In *18th International Software Product Lines Conference - Companion Volume, SPLC '14, Florence, Italy, September 15-19, 2014*, pages 143–147, 2014.
- [ABB⁺02] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, J  rgen W  st, and J  rg Zettel. *Component-based Product Line Engineering with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [ABBJ14] Mathieu Acher, Benoit Baudry, Olivier Barais, and Jean-Marc J  z  quel. Customization and 3D printing: a challenging playground for software product lines. In *18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014*, pages 142–146, 2014.
- [ABC⁺15] Mathieu Acher, Guillaume B  can, Beno  t Combemale, Benoit Baudry, and Jean-Marc J  z  quel. Product lines can jeopardize their trade secrets. In *FSE*, 2015.
- [ABKLT16] Kevin Allix, Tegawend   F Bissyand  , Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *MSR*, 2016.
- [ABKS13] Sven Apel, Don S. Batory, Christian K  stner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013.
- [ABS09] Muhammad Sarmad Ali, Muhammad Ali Babar, and Klaus Schmid. A comparative survey of economic models for software product lines. In *35th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2009, Patras, Greece, August 27-29, 2009, Proceedings*, pages 275–278. IEEE Computer Society, 2009.
- [ACC⁺14] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. Extraction and evolution of architectural variability models in plugin-based systems. *Software and System Modeling*, 13(4):1367–1394, 2014.
- [AK09] Sven Apel and Christian K  stner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, July/August 2009.
- [AKL09] Sven Apel, Christian Kastner, and Christian Lengauer. FEATUREHOUSE: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering, ICSE '09*, pages 221–231, Washington, DC, USA, 2009. IEEE Computer Society.
- [Al-15] Mustafa Al-Hajjaji. Scalable sampling and prioritization for product-line testing. In *Software Engineering & Management 2015, Dresden, Germany*, pages 295–298, 2015.
- [AM13] Ra’Fat Al-Msie’Deen. Mining Feature Models from the Object-Oriented Source Code of a Collection of Software Product Variants. In *ECOOP: European Conference on Object-Oriented Programming*, pages 1–10, Montpellier, France, July 2013.
- [AMC⁺07] Vander Alves, Pedro Matos, Leonardo Cole, Alexandre Vasconcelos, Paulo Borba, and Geber Ramalho. Extracting and evolving code in product lines with aspect-oriented programming. *Trans. Aspect-Oriented Software Development*, 4:117–142, 2007.

- [AmHS⁺14] Ra'Fat AL-msie'deen, Marianne Huchard, Abdelhak-Djamel D. Seriai, Christelle Urtado, and Sylvain Vauttier. Concept lattices: A representation space to structure software variability. In *Information and Communication Systems (ICICS), 2014 5th International Conference on*, pages 1–6, April 2014.
- [Apa10] Apache. Opennlp. <http://opennlp.apache.org>, 2010.
- [AS16] Imad Eddine Araar and Hassina Seridi. Software features extraction from object-oriented source code using an overlapping clustering approach. *Informatica*, 40(2), 2016.
- [ASC⁺06] Vander Alves, Gustavo Santos, Fernando Calheiros, Vilmar Nepomuceno, Davi Pires, Alberto Costa Neto, and Paulo Borba. Beyond code: Handling variability in art artifacts in mobile game product lines. In *SPLC*, 2006.
- [ASD⁺11] Raian Ali, Carlos Solís, Fabiano Dalpiaz, Walid Maalej, Paolo Giorgini, and Bashar Nuseibeh. Social software product lines. In *RESC*, pages 14–17. IEEE, 2011.
- [ASH⁺13a] Ra'Fat Al-Msie'deen, Abdelhak Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. Feature location in a collection of software product variants using formal concept analysis. In *Proc. of Intern. Conf. on Soft. Reuse, ICSR 2013*, pages 302–307, 2013.
- [ASH⁺13b] Ra'Fat Al-Msie'deen, Abdelhak-Djamel Seriai, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Mining features from the object-oriented source code of software variants by combining lexical and structural similarity. In *ICSR*, 2013.
- [ASH⁺13c] Ra'Fat Al-Msie'deen, Abdelhak-Djamel Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. Mining features from the object-oriented source code of a collection of software variants using formal concept analysis and latent semantic indexing. In *SEKE*, 2013.
- [AV14] Wesley Klewerton Guez Assunção and Silvia Regina Vergilio. Feature location for software product line migration: a mapping study. In *18th International Software Product Lines Conference - Companion Volume, SPLC '14, Florence, Italy, September 15-19, 2014*, pages 52–59, 2014.
- [BABBN15] Guillaume Bécan, Mathieu Acher, Benoit Baudry, and Sana Ben Nasr. Breathing Ontological Knowledge Into Feature Model Synthesis: An Empirical Study. *Empirical Software Engineering*, page 51, 2015.
- [BBNAB14] Guillaume Bécan, Sana Ben Nasr, Mathieu Acher, and Benoit Baudry. WebFML: Synthesizing Feature Models Everywhere. In *SPLC*, 2014.
- [BCDM14] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. Modisco: A model driven reverse engineering framework. *Information & Software Technology*, 56, 2014.
- [BD10] Benjamin Biegel and Stephan Diehl. JCCD: a flexible and extensible API for implementing custom code clone detectors. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 167–168. ACM, 2010.

-
- [BE09] Margaret A. Boden and Ernest A. Edmonds. What is generative art? *Digital Creativity*, 20(1-2):21–46, 2009.
- [Beu10] Danilo Beuche. Modeling and building software product lines with pure::variants. In *SPLC Workshops*, page 296, 2010.
- [BFK⁺99] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean-Marc DeBaud. Pulse: A methodology to develop software product lines. In *Proceedings of the 1999 Symposium on Software Reusability, SSR '99*, pages 122–131, New York, NY, USA, 1999. ACM.
- [BHJ⁺15] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 257–269, 2015.
- [BHP03] Stan Böhne, Günter Halmans, and Klaus Pohl. Modelling dependencies between variation points in use case diagrams. In *in Proceedings of 9th Intl. Workshop on Requirements Engineering - Foundations for Software Quality*, pages 59–69, 2003.
- [BJS09] Goetz Botterweck, Mikolás Janota, and Denny Schneeweiss. A design of a configurable feature model configurator. In *VaMoS*, volume 29 of *ICB Research Report*, pages 165–168. Universität Duisburg-Essen, 2009.
- [BLC16] Manuel Ballarín, Raúl Lapeña, and Carlos Cetina. Leveraging feature location to extract the clone-and-own relationships of a family of software products. In *Software Reuse: Bridging with Social-Awareness - 15th International Conference, ICSR 2016, Limassol, Cyprus, June 5-7, 2016, Proceedings*, volume 9679 of *Lecture Notes in Computer Science*, pages 215–230. Springer, 2016.
- [BLR⁺15] 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, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 16–25. ACM, 2015.
- [BM11] Jaime Barreiros and Ana Moreira. Soft constraints in feature models. In *International Conference in Software Engineering Advances.*, 2011.
- [BMAC05] David Benavides, Pablo Trinidad Martín-Arroyo, and Antonio Ruiz Cortés. Automated reasoning on feature models. In *CAiSE*, pages 491–503, 2005.
- [BMMM08] Xavier Blanc, Isabelle Mounier, Alix Mougénou, and Tom Mens. Detecting model inconsistency through operation-based model construction. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, pages 511–520, New York, NY, USA, 2008. ACM.
- [Bos01] Jan Bosch. Software product lines: Organizational alternatives. In Hausi A. Müller, Mary Jean Harrold, and Wilhelm Schäfer, editors, *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada*, pages 91–100. IEEE Computer Society, 2001.
- [BP10] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.

- [BRN⁺13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. A survey of variability modeling in industrial practice. In *VaMoS 2013*, 2013.
- [Bro96] John Brooke. SUS - A quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.
- [BSC10] David Benavides, Sergio Segura, and Antonio Ruiz Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.
- [BSR04] Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.
- [BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.
- [BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *LNCSE, Advanced information systems engineering: 17th international conference, CAISE 2005*. Springer, 2005.
- [CA05] Krzysztof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proceedings*, volume 3676 of *Lecture Notes in Computer Science*, pages 422–437. Springer, 2005.
- [CDFP97] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Trans. Software Eng.*, 23(7):437–444, 1997.
- [CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming - methods, tools and applications*. Addison-Wesley, 2000.
- [CF12] Andres Colubri and Ben Fry. Introducing processing 2.0. In *SIGGRAPH Talks*, page 12. ACM, 2012.
- [CFSS16] Marsha Chechik, Michalis Famelis, Rick Salay, and Daniel Strüber. *Perspectives of Model Transformation Reuse*, pages 28–44. Springer International Publishing, Cham, 2016.
- [CGR⁺12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. Cool features and tough decisions: A comparison of variability modeling approaches. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, VaMoS '12*, pages 173–182, New York, NY, USA, 2012. ACM.
- [CHSL11] 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*, pages 321–330. ACM, 2011.
- [CMS99] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman, editors. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann Publishers Inc., 1999.
- [Col] Colorbrewer. Colorbrewer2, color advice for cartography. <http://colorbrewer2.org>.

-
- [CP06] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings*, pages 211–220, 2006.
- [CR10] Kunrong Chen and Vaclav Rajlich. Case study of feature location using dependence graph, after 10 years. In *The 18th IEEE International Conference on Program Comprehension, ICPC 2010, Braga, Minho, Portugal, June 30-July 2, 2010*, pages 1–3, 2010.
- [CSW08] Krzysztof Czarnecki, Steven She, and Andrzej Wasowski. Sample spaces and feature models: There and back again. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*, pages 22–31, 2008.
- [CVF11a] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. Extracting software product lines: A case study using conditional compilation. In *CSMR*, 2011.
- [CVF11b] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. Extracting software product lines: A case study using conditional compilation. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering, CSMR '11*, pages 191–200, Washington, DC, USA, 2011. IEEE Computer Society.
- [DCB⁺15] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: a meta-language for modular and reusable development of dsls. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, Pittsburgh, PA, USA, October 25-27, 2015*, pages 25–36. ACM, 2015.
- [dCMMCdA14] Ivan do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana de Almeida. On strategies for testing software product lines: A systematic literature review. *Information and Software Technology*, 56(10):1183 – 1199, 2014.
- [dCMMdA12] Ivan do Carmo Machado, John D. McGregor, and Eduardo Santana de Almeida. Strategies for testing products in software product lines. *ACM SIGSOFT Softw. Eng. Notes*, 37(6), 2012.
- [DDH⁺13] Jean-Marc Davril, Edouard Delfosse, Negar Hariri, Mathieu Acher, Jane Cleland-Huang, and Patrick Heymans. Feature model extraction from large collections of informal product descriptions. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, pages 290–300, 2013.
- [DDL⁺90] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *JASIS*, 41(6):391–407, 1990.
- [DHJ⁺08] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. Clone detection in automotive model-based development. In *ICSE 2008*, 2008.
- [DIS09] ISO DIS. 9241-210: 2010. ergonomics of human system interaction-part 210: Human-centred design for interactive systems. *International Standardization Organization (ISO)*. Switzerland, 2009.

- [DLL06] Marco D'Ambros, Michele Lanza, and Mircea Lungu. The evolution radar: Visualizing integrated logical coupling information. In *Proceedings of the International Workshop on Mining Software Repositories*, 2006.
- [DMSD13] Cosmin Dumitrescu, Raul Mazo, Camille Salinesi, and Alain Dauron. Bridging the gap between product lines and systems engineering: An experience in variability management for automotive model based systems engineering. In *Proceedings of the 17th International Software Product Line Conference, SPLC '13*, pages 254–263, New York, NY, USA, 2013. ACM.
- [Dor13] Alan Dorin. Aesthetic selection and the stochastic basis of art, design and interactive evolutionary computation. In *GECCO*, pages 311–318, 2013.
- [EBG12] Faezeh Ensan, Ebrahim Bagheri, and Dragan Gašević. Evolutionary search-based test generation for software product line feature models. In *Advanced Information Systems Engineering*, volume 7328 of *Lecture Notes in Computer Science*, pages 613–628. Springer Berlin Heidelberg, 2012.
- [Ecl14a] Eclipse. Requirements Engineering Platform. <http://eclipse.org/rmf/pror/>, 2014.
- [Ecl14b] Eclipse. The visualiser, AJDT: AspectJ Development Tools. <http://www.eclipse.org/ajdt/visualiser>, 2014.
- [Ecl16a] Eclipse. Eclipse integrated development environment. <http://eclipse.org>, 2016.
- [Ecl16b] Eclipse. EMF Compare. <http://www.eclipse.org/emf/compare/>, 2016.
- [Ecl16c] Eclipse. EMF Diff/Merge: a diff/merge component for models. <http://eclipse.org/diffmerge>, 2016.
- [Edw11] Michael Edwards. Algorithmic composition: Computational thinking in music. *Commun. ACM*, 54(7):58–67, 2011.
- [Eib08] A.E. Eiben. Evolutionary reproduction of dutch masters: The Mondriaan and Escher evolvers. In *The Art of Artificial Evolution*, Natural Computing Series, pages 211–224. Springer Berlin Heidelberg, 2008.
- [ES03] Agoston E. Eiben and Jim E. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.
- [FA12] Gordon Fraser and Andrea Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012*, pages 121–130, 2012.
- [FBHC15] Jaime Font, Manuel Ballarin, Oystein Haugen, and Carlos Cetina. Automating the variability formalization of a model family by means of common variability language. In *SPLC*, pages 411–418, 2015.
- [FCS⁺08] Eduardo Figueiredo, Nélío Cacho, Cláudio Sant'Anna, Mario Monteiro, Uirá Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Cutigi Ferrari, Safoora Shakil Khan, Fernando Castor Filho, and Francisco Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE*, 2008.

-
- [FFBA⁺14] Joao Bosco Ferreira Filho, Olivier Barais, Mathieu Acher, Jérôme Le Noir, Axel Legay, and Benoit Baudry. Generating Counterexamples of Model-based Software Product Lines. *Software Tools for Technology Transfer (STTT)*, July 2014.
 - [fJ15] WordNet Similarity for Java. Wordle word clouds generator. <https://code.google.com/p/ws4j/>, 2015.
 - [Fla98] Gary William Flake. The computational beauty of nature, computer explorations of fractals, chaos, complex systems, and adaptation. *The MIT press*, 1998.
 - [FLLE14] 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 International Conference on Software Maintenance and Evolution (ICSME), 2014*, pages 391–400, 2014.
 - [Fre83] Peter Freeman. Reusable software engineering: Concepts and research directions. *ITT Proceedings of the Workshop on Reusability in Programming*, 129:137, 1983.
 - [GBZ16] Loïc Girault, Cédric Besse, and Mikal Ziane. Puck: an architecture refactoring tool. <https://pages.lip6.fr/puck>, 2016.
 - [GFSV14] Alfonso García Frey, Jean-Sébastien Sottet, and Alain Vagner. AME: an adaptive modelling environment as a collaborative modelling tool. In *Proceedings of the 2014 ACM SIGCHI symposium on Engineering interactive computing systems*, pages 189–192. ACM, 2014.
 - [GKHB10] Nicolas Gold, Jens Krinke, Mark Harman, and David Binkley. Issues in clone classification for dataflow languages. In *the 4th International Workshop on Software Clones, IWSC '10*, 2010.
 - [Goo01] Michael Good. MusicXML for Notation and Analysis. In *The virtual score: representation, retrieval, restoration*, volume 12 of *Computing in Musicology*, pages 113–124. The MIT Press, Massachusetts, 2001.
 - [GRDL09] Paul Grünbacher, Rick Rabiser, Deepak Dhungana, and Martin Lehofer. Model-based customization and deployment of eclipse-based tools: Industrial experiences. In *Intern. Conf. on Aut. Sof. Eng. (ASE)*, pages 247–256, 2009.
 - [Gre05] Gary Greenfield. Evolutionary methods for ant colony paintings. In *Applications of Evolutionary Computing*, volume 3449 of *LNCS*, 2005.
 - [GW97] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1997.
 - [HAB13] Hadi Hemmati, Andrea Arcuri, and Lionel C. Briand. Achieving scalable model-based testing through test case diversity. *ACM Trans. Softw. Eng. Methodol.*, 22(1), 2013.
 - [Ham50] Richard W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 26(2):147–160, 1950.
 - [HB12] Gregory S. Hornby and Josh C. Bongard. Accelerating human-computer collaborative search through learning comparative and predictive user models. In *GECCO*, 2012.
 - [HFH⁺09] Mark A. Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1), 2009.

- [HKW08] Florian Heidenreich, Jan Kopcsek, and Christian Wende. Featuremapper: mapping features to models. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 943–944, 2008.
- [HLE13] Evelyn Nicole Haslinger, Roberto Erick Lopez-Herrejon, and Alexander Egyed. On extracting feature models from sets of valid feature combinations. In *Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*, pages 53–67. Springer, 2013.
- [HMA10] Haitham S. Hamza, Jabier Martinez, and Carmen Alonso. Introducing product line architectures in the ERP industry: Challenges and lessons learned. In *Software Product Lines - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Workshop Proceedings (Volume 2 : Industrial Track)*, pages 263–266. Lancaster University, 2010.
- [HØ14] Øystein Haugen and Ommund Øgård. BVR - better variability results. In Daniel Amyot, Pau Fonseca i Casas, and Gunter Mussbacher, editors, *System Analysis and Modeling: Models and Reusability - 8th International Conference, SAM 2014, Valencia, Spain, September 29-30, 2014. Proceedings*, volume 8769 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2014.
- [HPP⁺13a] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Multi-objective test generation for software product lines. In *SPLC*, pages 62–71, 2013.
- [HPP⁺13b] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. PLEDGE: a product line editor and test generation tool. In *17th International Software Product Line Conference co-located workshops, SPLC 2013 workshops, Tokyo, Japan - August 26 - 30, 2013*, pages 126–129. ACM, 2013.
- [HPP⁺14] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Trans. Software Eng.*, 40(7):650–670, 2014.
- [HWC13] Øystein Haugen, Andrzej Wasowski, and Krzysztof Czarnecki. CVL: common variability language. In *SPLC*, 2013.
- [IBK11] Paul Istoean, Nicolas Biri, and Jacques Klein. Issues in model-driven behavioural product derivation. In *VAMOS*, 2011.
- [IRW16] Nili Itzik, Iris Reinhartz-Berger, and Yair Wand. Variability analysis of requirements: Considering behavioral differences and reflecting stakeholders’ perspectives. *IEEE Trans. Software Eng.*, 42(7):687–706, 2016.
- [Ite14] Itemis. Yakindu statechart tool. <http://statecharts.org>, 2014.
- [JDB07] Hans Peter Jepsen, Jan Gaardsted Dall, and Danilo Beuche. Minimally invasive migration to software product lines. In *SPLC*, 2007.
- [JDH09] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. Clonedetective - a workbench for clone detection research. In *ICSE 2009*, 2009.
- [JDHW09] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *ICSE 2009*, 2009.

-
- [JHF⁺12] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. Generating better partial covering arrays by modeling weights on sub-product lines. In *MoDELS*, pages 269–284, 2012.
- [JKLM06] Isabel John, Jens Knodel, Theresa Lehner, and Dirk Muthig. A practical guide to product line scoping. In *Software Product Lines, 10th International Conference, SPLC 2006, Baltimore, Maryland, USA, August 21-24, 2006, Proceedings*, pages 3–12. IEEE Computer Society, 2006.
- [Jon07] Karen Spärck Jones. Automatic summarising: The state of the art. *Inf. Process. Manage.*, 43(6):1449–1481, 2007.
- [KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proc. of the 30th Inter. Conf. on Soft. Eng. (ICSE)*, pages 311–320, 2008.
- [KBGE11] Steffen Koch, Harald Bosch, Mark Giereth, and Thomas Ertl. Iterative integration of visual insights during scalable patent search and analysis. *IEEE Trans. Vis. Comput. Graph.*, 17(5), 2011.
- [KC] Charles W. Krueger and Paul C. Clements. Systems and software product line engineering with BigLever software gears. In *SPLC 2012: Volume 2*.
- [KCB08] Charles W. Krueger, Dale Churchett, and Ross Buhrdorf. Homeaway’s transition to software product line practice: Engineering and business results in 60 days. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*, pages 297–306. IEEE Computer Society, 2008.
- [KCH⁺90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, 1990.
- [KDO14] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. Variability mining: Consistent semi-automatic detection of product-line features. *IEEE Trans. Software Eng.*, 40(1):67–82, 2014.
- [KDRPP09] Dimitrios S. Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models, CVSM ’09*, pages 1–6, Washington, DC, USA, 2009. IEEE Computer Society.
- [KFBA09] Rainer Koschke, Pierre Frenzel, Andreas P. J. Breu, and Karsten Angstmann. Extending the reflexion method for consolidating software variants into product lines. *Software Quality Journal*, 17(4):331–366, 2009.
- [KGB⁺14] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining github. In *MSR*, 2014.
- [KLD02] Kyo Chul Kang, Jaejoon Lee, and Patrick Donohoe. Feature-oriented project line engineering. *IEEE Software*, 19(4):58–65, 2002.
- [KLM⁺97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

- [Kru92] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24, 1992.
- [Kru01] Charles W. Krueger. Easing the transition to software mass customization. In *Software Product-Family Engineering, 4th International Workshop, PFE 2001, Bilbao, Spain, October 3-5, 2001, Revised Papers*, volume 2290 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2001.
- [KTA08] Christian Kästner, Salvador Trujillo, and Sven Apel. Visualizing software product line variabilities in source code. In *SPLC Volume 2*, 2008.
- [LAG⁺14] Lukas Linsbauer, Florian Angerer, Paul Grünbacher, Daniela Lettner, Herbert Prähofer, Roberto E. Lopez-Herrejon, and Alexander Egyed. Recovering feature-to-code mappings in mixed-variability software systems. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 426–430. IEEE Computer Society, 2014.
- [LB01] Roberto E. Lopez-Herrejon and Don S. Batory. A standard problem for evaluating product-line methodologies. In *Generative and Component-Based Software Engineering, Third International Conference, GCSE 2001, Erfurt, Germany, September 9-13, 2001, Proceedings*, pages 10–24, 2001.
- [LBB⁺15a] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *IFIP SEC*, 2015.
- [LBB⁺15b] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *ICSE*, 2015.
- [LBKLT16] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *SANER*, 2016.
- [LBL06] Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *Proceedings of the 28th International Conference on Software Engineering, ICSE '06*, pages 112–121, New York, NY, USA, 2006. ACM.
- [LBP⁺16] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. Technical report, 2016.
- [Lev66] Vladimir Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10, 1966.
- [LFC⁺14] Roberto E. Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Evelyn Nicole Haslinger, Alexander Egyed, and Enrique Alba. Towards a benchmark and a comparison framework for combinatorial interaction testing of software product lines. *CoRR*, abs/1401.5367, 2014.
- [LHE13] Roberto E. Lopez-Herrejon and Alexander Egyed. Towards interactive visualization support for pairwise testing software product lines. In *VISSOFT*, pages 1–4. IEEE, 2013.

- [LHGB⁺12] Roberto Erick Lopez-Herrejon, José A. Galindo, David Benavides, Sergio Segura, and Alexander Egyed. Reverse engineering feature models with evolutionary algorithms: An exploratory study. In *SSBSE*, volume 7515 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2012.
- [Li12] Yang Li. Adaptive learning evaluation model for evolutionary art. In *IEEE Congress on Evolutionary Computation*, 2012.
- [LLB⁺16] Li Li, Daoyuan Li, Tegawendé F Bissyandé, David Lo, Jacques Klein, and Yves Le Traon. Ungrafting malicious code from piggybacked android apps. Technical report, 2016.
- [LLE16] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Variability extraction and modeling for product variants. *SoSyM: Open Access*, 2016.
- [LLM07] Begoña Losada, David Lopez, and Jabier Martinez. Intergram, an user-centered design process. In *9th European Conference for the Advancement of Assistive Technology, AAATE 2007, October 3-5, San Sebastian, Spain*, pages 786–790, 2007.
- [LMSM10] Alberto Lora-Michiels, Camille Salinesi, and Raúl Mazo. A method based on association rules to construct product line models. In *VaMoS*, volume 37 of *ICB-Research Report*, pages 147–150. Universität Duisburg-Essen, 2010.
- [LMTY13] Antonios Liapis, Héctor Perez Martínez, Julian Togelius, and Georgios N. Yannakakis. Adaptive game level creation through rank-based interactive evolution. In *2013 IEEE Conference on Computational Intelligence in Games (CIG), Niagara Falls, ON, Canada, August 11-13, 2013*, pages 1–8, 2013.
- [LMZ14] Amine Lajmi, Jabier Martinez, and Tewfik Ziadi. DSLFORGE: textual modeling on the web. In *Demos@MoDELS*, volume 1255 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2014.
- [LMZ⁺16] Li Li, Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Mining families of Android applications for extractive SPL adoption. In *Proceedings of the 20th International Conference on Software Product Lines, SPLC 2016, Beijing, China, September 19-23, 2016*.
- [LRC15] Yi Li, Julia Rubin, and Marsha Chechik. Semantic slicing of software version histories (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 686–696. IEEE Computer Society, 2015.
- [Mar10] Jabier Martinez. Generation language - enabling scalability for product realisation. In *Software Product Lines - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Workshop Proceedings*, pages 211–216. Lancaster University, 2010.
- [McG07] John D. McGregor. Testing a software product line. In *Testing Techniques in Software Engineering*, volume 6153 of *Lecture Notes in Computer Science*, pages 104–140. Springer, 2007.
- [McI68] Malcolm D. McIlroy. Mass-produced software components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

- [MDPG09] Jabier Martinez, Jessica Díaz, Jennifer Pérez, and Juan Garbajosa. Software product line engineering approach for enhancing agile methodologies. In *Agile Processes in Software Engineering and Extreme Programming, 10th International Conference, XP 2009, Pula, Sardinia, Italy, May 25-29, 2009. Proceedings*, volume 31 of *Lecture Notes in Business Information Processing*, pages 247–248. Springer, 2009.
- [MDSD14] Raúl Mazo, Cosmin Dumitrescu, Camille Salinesi, and Daniel Diaz. Recommendation heuristics for improving product line configuration processes. In *Recommendation Systems in Software Engineering*, pages 511–537. 2014.
- [MGC⁺16] David Méndez-Acuña, José A. Galindo, Benoît Combemale, Arnaud Blouin, Benoit Baudry, and Gurvan Le Guernic. Reverse-engineering reusable language modules from legacy domain-specific languages. In *Software Reuse: Bridging with Social-Awareness - 15th International Conference, ICSR 2016, Limassol, Cyprus, June 5-7, 2016, Proceedings*, volume 9679 of *Lecture Notes in Computer Science*, pages 368–383. Springer, 2016.
- [MLA⁺09] Jabier Martinez, Cristina Lopez, Aitor Aldazabal, Jason Mansell, and Marta del Hierro. Plum (product line unified modeller). *13th International Software Product Line Conference, SPLC, Tool demo*, 2009.
- [MLUdH09] Jabier Martinez, Cristina Lopez, Estibaliz Ulacia, and Marta del Hierro. Towards a model-driven product line for web systems. *5th Model-Driven Web Engineering Workshop, MDWE*, pages 1–15, 2009.
- [MMZF15] Jabier Martinez, Jan Malburg, Tewfik Ziadi, and Görschwin Fey. Towards analysing feature locations through testing traces with BUT4Reuse. In *Workshop on Design Automation for Understanding Hardware Designs (DUHDe)*, 2015.
- [MRSB13] Ben Meadows, Patricia Riddle, Cameron Skinner, and MichaelM. Barley. Evaluating the seeding genetic algorithm. In Stephen Cranefield and Abhaya Nayak, editors, *AI 2013: Advances in Artificial Intelligence*, volume 8272 of *Lecture Notes in Computer Science*, pages 221–227. Springer International Publishing, 2013.
- [MRZ⁺15] Jabier Martinez, Gabriele Rossi, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Estimating and predicting average likability on computer-generated artwork variants. In *Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings*, pages 1431–1432. ACM, 2015.
- [MT12] Jabier Martinez and Anil Kumar Thurimella. Collaboration and source code driven bottom-up product line engineering. In *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 2*, pages 196–200. ACM, 2012.
- [MYH14] Héctor Perez Martínez, Georgios N. Yannakakis, and John Hallam. Don’t classify ratings of affect; rank them! *T. Affective Computing*, 5(3):314–326, 2014.
- [MZB⁺15a] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Automating the extraction of model-based software product lines from model variants (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 396–406. IEEE Computer Society, 2015.

-
- [MZB⁺15b] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Bottom-up adoption of software product lines: a generic and extensible approach. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 101–110. ACM, 2015.
 - [MZB⁺16] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Name suggestions during feature identification: The VariClouds approach. In *Proceedings of the 20th International Conference on Software Product Lines, SPLC 2016, Beijing, China, September 19-23, 2016*.
 - [MZKT14] Jabier Martinez, Tewfik Ziadi, Jacques Klein, and Yves Le Traon. Identifying and visualising commonality and variability in model variants. In *Modelling Foundations and Applications - 10th European Conference, ECMFA 2014, Held as Part of STAF 2014, York, UK, July 21-25, 2014. Proceedings*, pages 117–131, 2014.
 - [MZM⁺14] Jabier Martinez, Tewfik Ziadi, Raúl Mazo, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Feature relations graphs: A visualisation paradigm for feature constraints in software product lines. In *Second IEEE Working Conference on Software Visualization, VISSOFT 2014, Victoria, BC, Canada, September 29-30, 2014*, pages 50–59, 2014.
 - [MZP⁺16] Jabier Martinez, Tewfik Ziadi, Mike Papadakis, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Feature location benchmark for software families using Eclipse community releases. In *ICSR*, volume 9679 of *Lecture Notes in Computer Science*, pages 267–283. Springer, 2016.
 - [NBKC14] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *ICSE*, 2014.
 - [NC⁺09] Linda M. Northrop, Paul C. Clements, et al. A Framework for Software Product Line Practice, Version 5.0. www.sei.cmu.edu/productlines/framework.html, 2009.
 - [NL93] Jakob Nielsen and Thomas K Landauer. A mathematical model of the finding of usability problems. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 206–213. ACM, 1993.
 - [Nor04] Linda Northrop. Software product line adoption roadmap. Technical Report CMU/SEI-2004-TR-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2004.
 - [NTB⁺08] Daren Nestor, Steffen Thiel, Goetz Botterweck, Ciarán Cawley, and Patrick Healy. Applying visualisation techniques in software product lines. In *Proceedings of the ACM 2008 Symposium on Software Visualization, Ammersee, Germany, September 16-17, 2008*, pages 175–184. ACM, 2008.
 - [OMG06] OMG. Meta Object Facility (MOF) Core Specification. <http://www.omg.org/spec/MOF/2.0/>, 2006.
 - [OMG13] OMG. Requirements Interchange Format (ReqIF). <http://www.omg.org/spec/ReqIF/>, 2013.
 - [OMG14] OMG. Object Constraint Language. <http://www.omg.org/spec/OCL/>, 2014.
 - [Par76] David Lorge Parnas. On the design and development of program families. *IEEE Trans. Software Eng.*, 2(1):1–9, 1976.

- [PB12] Andreas Pleuss and Goetz Botterweck. Visualization of variability and configuration options. *STTT*, 14(5):497–510, 2012.
- [PBD10] Andreas Pleuss, Goetz Botterweck, and Deepak Dhungana. Integrating automated product derivation and individual user interface design. *VaMoS*, 10:69–76, 2010.
- [PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. 2005.
- [Per85] Ken Perlin. An image synthesizer. In *SIGGRAPH*, 1985.
- [Pet01] Wiebke Petersen. A set-theoretical approach for the induction of inheritance hierarchies. *Electr. Notes Theor. Comput. Sci.*, 53:296–308, 2001.
- [Pet12] Hauke Petersen. Clone detection in matlab simulink models. In *Master thesis, Tech. Univ. Denmark*, 2012.
- [PHDB12] Andreas Pleuss, Benedikt Hauptmann, Deepak Dhungana, and Goetz Botterweck. User interface engineering for software product lines: the dilemma between automation and usability. In *EICS*, pages 25–34. ACM, 2012.
- [PKGJ08] Gilles Perrouin, Jacques Klein, Nicolas Guelfi, and Jean-Marc Jézéquel. Reconciling automation and flexibility in product derivation. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*, pages 339–348, 2008.
- [PNN⁺09] Nam H. Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Complete and accurate clone detection in graph-based models. In *ICSE 2009*, 2009.
- [Pol15] Polarsys. Capella. <https://www.polarsys.org/capella/>, 2015.
- [Por01] Martin F. Porter. Snowball: A language for stemming algorithms. <http://snowball.tartarus.org>, 2001.
- [PRB11] Andreas Pleuss, Rick Rabiser, and Goetz Botterweck. Visualization techniques for application in interactive product configuration. In *SPLC Workshops*, page 22. ACM, 2011.
- [Pre01] Christian Prehofer. Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience*, 13(6):465–501, 2001.
- [PTS⁺16] Tristan Pfofe, Thomas Thuem, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. Synchronizing Software Variants with VariantSync. In *Proceedings of the 20th International Conference on Software Product Lines, SPLC 2016, Beijing, China, September 19-23, 2016*, 2016.
- [PYW10] Jörg Pleumann, Omry Yadan, and Erik Wetterberg. Antenna. <https://antenna.sourceforge.net>, 2010.
- [RAN⁺14] Israel J. Mojica Ruiz, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E. Hassan. A large-scale empirical study on software reuse in mobile apps. *IEEE Soft.*, 31(2), 2014.
- [RC07] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. *School of computing TR 2007-541, queel’s university*, 115, 2007.

-
- [RC12a] Julia Rubin and Marsha Chechik. Combining related products into product lines. In *Fundamental Approaches to Software Engineering, FASE 2012, Tallinn, Estonia, 2012*, 2012.
- [RC12b] Julia Rubin and Marsha Chechik. Locating distinguishing features using diff sets. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pages 242–245, 2012.
- [RC13a] Julia Rubin and Marsha Chechik. N-way model merging. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, pages 301–311, New York, NY, USA, 2013. ACM.
- [RC13b] Julia Rubin and Marsha Chechik. A survey of feature location techniques. In *Domain Engineering, Product Lines, Languages, and Conceptual Models*, pages 29–58. Springer, 2013.
- [RCC13] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Managing cloned variants: A framework and experience. In *SPLC*, 2013.
- [RM02] 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, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 406–416. ACM, 2002.
- [RM08] Juan Romero and Penousal Machado, editors. *The Art of Artificial Evolution: A Handbook on Evolutionary Art and Music*. Natural Computing Series. Springer, 2008.
- [Rob05] Martin P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 11–20. ACM, 2005.
- [RPK10] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. Automatic variation-point identification in function-block-based models. In *GPCE*, pages 23–32, 2010.
- [RPK11] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. Extraction of feature models from formal contexts. In *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011. Workshop Proceedings (Volume 2)*, page 4. ACM, 2011.
- [RR00] Jason E. Robbins and David F. Redmiles. Cognitive support, UML adherence, and XMI interchange in argo/uml. *Information & Software Technology*, 42(2), 2000.
- [Rub14] Julia Rubin. *Cloned product variants: from ad-hoc to well-managed software reuse*. PhD thesis, University of Toronto, 2014.
- [SASC12] Matthew Stephan, Manar H. Alalfi, Andrew Stevenson, and James R. Cordy. Towards qualitative comparison of simulink model clone detection approaches. In *Proceeding of the 6th International Workshop on Software Clones, IWSC 2012, Zurich, Switzerland, June 4, 2012*, pages 84–85. IEEE, 2012.
- [SB97] Dominique L Scapin and JM Christian Bastien. Ergonomic criteria for evaluating the ergonomic quality of interactive systems. *Behaviour & information technology*, 16(4-5):220–231, 1997.

- [SBB⁺02] Ivan A. Sag, Timothy Baldwin, Francis Bond, Ann A. Copestake, and Dan Flickinger. Multiword expressions: A pain in the neck for NLP. In *CICLing*, 2002.
- [SBB⁺10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2010.
- [SC13] Matthew Stephan and James R. Cordy. A survey of model comparison approaches and applications. In *MODELSWARD*, pages 265–277, 2013.
- [Sch00] Klaus Schmid. *Scoping Software Product Lines*, pages 513–532. Springer US, Boston, MA, 2000.
- [Sch02] Klaus Schmid. A comprehensive product line scoping approach and its validation. In *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 593–603. ACM, 2002.
- [Sch06] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), February 2006.
- [SEH03] Susan Elliott Sim, Steve M. Easterbrook, and Richard C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, pages 74–83, 2003.
- [SFdOA13] Iuri Santos Souza, Rosemeire Fiaccone, Raphael Pereira de Oliveira, and Eduardo Santana De Almeida. On the relationship between features granularity and non-conformities in software product lines: An exploratory study. In *27th Brazilian Symposium on Software Engineering, SBES 2013, Brasilia, Brazil, October 1-4, 2013*, pages 147–156, 2013.
- [SGB⁺12] Sergio Segura, José A. Galindo, David Benavides, José Antonio Parejo, and Antonio Ruiz Cortés. Betty: benchmarking and testing on the automated analysis of feature models. In *Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings*, pages 63–71, 2012.
- [SGLS07] John T. Stasko, Carsten Görg, Zhicheng Liu, and Kanupriya Singhal. Jigsaw: Supporting investigative analysis through interactive visualization. In *IEEE VAST*, 2007.
- [SIN15] SINTEF. CVL Tool. <http://www.omgwiki.org/variability/doku.php>, 2015.
- [SL93] Stephen Schwanauer and David Levitt. Machine models of music. *The MIT press*, 1993.
- [SMA13] Abdel Salam Sayyad, Tim Menzies, and Hany Ammar. On the value of user preferences in search-based software engineering: a case study in software product lines. In *ICSE*, pages 492–501, 2013.
- [Smi07] Gene Smith. *Tagging: People-powered Metadata for the Social Web*. New Riders Publishing, 2007.

-
- [SPV11] Giriprasad Sridhara, Lori L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, ON, Canada, June 22-24, 2011*, pages 71–80. IEEE Computer Society, 2011.
- [SRA⁺14] Steven She, Uwe Ryssel, Nele Andersen, Andrzej Wasowski, and Krzysztof Czarnecki. Efficient synthesis of feature models. *Information and Software Technology*, 56(9), 2014.
- [SRK⁺13] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information & Software Technology*, 55(3), 2013.
- [SRP03] Detlef Streitferdt, Matthias Riebisch, and Ilka Philippow. Formal Details of Relations in Feature Models. In *Proceedings of the 10th IEEE Symposium and Workshops on Engineering of Computer-Based Systems.*, 2003.
- [SRTC14] Daniel Struber, Julia Rubin, Gabriele Taentzer, and Marsha Chechik. Splitting models using information retrieval and model crawling techniques. In *Fundamental Approaches to Software Engineering*, volume 8411 of *Lecture Notes in Computer Science*, pages 47–62. Springer Berlin Heidelberg, 2014.
- [SS08] Julio Sincero and Wolfgang Schröder-Preikschat. The linux kernel configurator as a feature modeling tool. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*, pages 257–260. Lero Int. Science Centre, University of Limerick, Ireland, 2008.
- [SSD13] Hamzeh Eyal Salman, Abdelhak-Djamel Seriai, and Christophe Dony. Feature-to-code traceability in a collection of software variants: Combining formal concept analysis and information retrieval. In *Intern. Conf. on Inform. Reuse and Integr. IRI*, pages 209–216, 2013.
- [SSD14] Hamzeh Eyal Salman, Abdelhak Seriai, and Christophe Dony. Feature location in a collection of product variants: Combining information retrieval and hierarchical clustering. In *Intern. Conf. on Sof. Eng. and Know. Eng. SEKE*, pages 426–430, 2014.
- [SSM11] Samuel Silva, Beatriz Sousa Santos, and Joaquim Madeira. Using color in visualization: A survey. *Computers & Graphics*, 35(2):320 – 333, 2011. Virtual Reality in Brazil Visual Computing in Biology and Medicine Semantic 3D media and content Cultural Heritage.
- [SSS16] Anas Shatnawi, Abdelhak Seriai, and Houari A. Sahraoui. Recovering architectural variability of a family of product variants. *CoRR*, abs/1606.00137, 2016.
- [STE⁺10] Julio Sincero, Reinhard Tartler, Christoph Egger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Facing the linux 8000 feature nightmare. In *Proceedings of ACM European Conference on Computer Systems (EuroSys 2010), Best Posters and Demos Session*, 2010.
- [Sur05] James Surowiecki. *The Wisdom of Crowds*. Anchor, 2005.

- [SV04] Max Schlee and Jean Vanderdonckt. Generative programming of graphical user interfaces. In *Proceedings of the working conference on Advanced visual interfaces, AVI 2004, Gallipoli, Italy, May 25-28, 2004*, pages 403–406. ACM Press, 2004.
- [SVGF15] Jean-Sébastien Sottet, Alain Vagner, and Alfonso García Frey. Variability management supporting the model-driven design of user interfaces. In *Modelsward*, 2015.
- [SWY75] Gerard Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.
- [Sys89] Gilbert Syswerda. Uniform crossover in genetic algorithms. 1989.
- [Tak01] Hideyuki Takagi. Interactive evolutionary computation: Fusion of the capabilities of EC optimization and human evaluation. *Proceedings of the IEEE*, 89(9):1275–1296, 2001.
- [TBC⁺08] Pablo Trinidad, David Benavides, Antonio Ruiz Cortés, Sergio Segura, and Alberto Jimenez. FAMA framework. In *SPLC 2008, Limerick, Ireland*, 2008.
- [TCBS08] Pablo Trinidad, Antonio Ruiz Cortés, David Benavides, and Sergio Segura. Three-dimensional feature diagrams visualization. In *SPLC (2)*, pages 295–302. Lero Int. Science Centre, University of Limerick, Ireland, 2008.
- [TDAJ16] Paul Temple, Jose Angel Galindo Duarte, Mathieu Acher, and Jean-Marc Jézéquel. Using Machine Learning to Infer Constraints for Product Lines. In *Proceedings of the 20th International Conference on Software Product Lines, SPLC 2016, Beijing, China, September 19-23, 2016*.
- [TFPB14] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. A study of library migrations in java. *Journal of Software: Evolution and Process*, 26(11):1030–1052, 2014.
- [TGB06] Robert Tairas, Jeff Gray, and Ira Baxter. Visualization of clone detection results. In *OOPSLA Workshop on Eclipse Technology eXchange*, 2006.
- [Tin15] Tinkerpop. TinkerPop3: A Graph Computing Framework. <http://blueprints.tinkerpop.com>, 2015.
- [TKB⁺14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79(0), 2014.
- [vdLSR07] Frank van der Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action - the best industrial practice in product line engineering*. Springer, 2007.
- [VG07] Markus Völter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*, pages 233–242. IEEE Computer Society, 2007.
- [Völ13] Markus Völter. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. <http://dslbook.org>, 2013.
- [Voo99] Ellen M. Voorhees. The TREC-8 question answering track report. In *TREC*, 1999.

-
- [WCKK06] David M Weiss, Paul C Clements, Kyo Kang, and Charles Krueger. Software product line hall of fame. In *Software Product Line Conference, 2006 10th International*, pages 237–237. IEEE, 2006.
 - [WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.
 - [WP94] Zhibiao Wu and Martha Stone Palmer. Verb semantics and lexical selection. In *32nd Annual Meeting of the Association for Computational Linguistics, 27-30 June 1994, New Mexico State University, Las Cruces, New Mexico, USA, Proceedings.*, pages 133–138. Morgan Kaufmann Publishers / ACL, 1994.
 - [WWL⁺10] Yingcai Wu, Furu Wei, Shixia Liu, Norman Au, Weiwei Cui, Hong Zhou, and Huamin Qu. Opinionseer: Interactive visualization of hotel customer feedback. *IEEE Trans. Vis. Comput. Graph.*, 16(6):1109–1118, 2010.
 - [XS07] Zhenchang Xing and Eleni Stroulia. Differencing logical UML models. *Autom. Softw. Eng.*, 14(2):215–259, 2007.
 - [XXJ12] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. Feature location in a collection of product variants. In *Proc. of Working Conf. on Rev. Eng., WCRE 2012*, pages 145–154, 2012.
 - [XXJ13] Zhenchang Xing, Yinxing Xue, and Stan Jarzabek. A large scale linux-kernel based benchmark for feature location research. In *Proced. of Intern. Conf. on Soft. Eng., ICSE*, pages 1311–1314, 2013.
 - [YdPLLM08] Yijun Yu, Julio Cesar Sampaio do Prado Leite, Alexei Lapouchnian, and John Mylopoulos. Configuring features with stakeholder goals. In *SAC*, pages 645–649. ACM, 2008.
 - [YM05] Trevor Young and Gail Murphy. Using AspectJ to build a product line for mobile devices. In *AOSD*, 2005.
 - [YPZ09] Yiming Yang, Xin Peng, and Wenyun Zhao. Domain feature model recovery from multiple applications using data access semantics and formal concept analysis. In *WCRE*, 2009.
 - [Zav03] Pamela Zave. An experiment in feature engineering. In *Programming methodology*, pages 353–377. Springer New York, 2003.
 - [ZFdSZ12] Tewfik Ziadi, Luz Frias, Marcos Aurélio Almeida da Silva, and Mikal Ziane. Feature identification from the source code of product variants. In *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, pages 417–422. IEEE Computer Society, 2012.
 - [Zha14] Xiaorui Zhang. *Developing Model-Driven Software Product Lines*. PhD thesis, University of Oslo, Norway, 2014.
 - [ZHM11] Xiaorui Zhang, Øystein Haugen, and Birger Møller-Pedersen. Model comparison to synthesize a model-driven software product line. In *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*, pages 90–99. IEEE, 2011.

- [ZHP⁺14] Tewfik Ziadi, Christopher Henard, Mike Papadakis, Mikal Ziane, and Yves Le Traon. Towards a language-independent approach for reverse-engineering of software product lines. In *Symposium on Applied Computing, SAC 2014, 2014*, pages 1064–1071, 2014.
- [ZJ06] Tewfik Ziadi and Jean-Marc Jézéquel. Software product line engineering with the UML: deriving products. In *Software Product Lines - Research Issues in Engineering and Management*, pages 557–588. Springer, 2006.