CLONED PRODUCT VARIANTS:
FROM AD-HOC TO WELL-MANAGED SOFTWARE REUSE

by

Julia Rubin

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

© Copyright 2014 by Julia Rubin
Abstract

Cloned Product Variants:
From Ad-Hoc to Well-Managed Software Reuse

Julia Rubin
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2014

A large number of companies, especially in the automotive, electronics, aerospace and defense domains, develop a portfolio of closely related software products designed to satisfy similar, yet not identical, needs of their customers (a.k.a. a software product line). Even though numerous software product line engineering approaches promise to ease the product line development and maintenance effort, in practice, the adoption of such approaches is still limited. Instead, products are often established ad-hoc, e.g., by copying existing variants and modifying them to fit the requirements of a new customer or market segment.

In this thesis, we investigate the reuse culture in organizations that employ cloning to realize their product lines. By analyzing development activities in several industrial product lines, we show that an efficient management of clones relies on both (1) the unification of the cloned variants into single-copy representations promoted by product line engineering methods and (2) the construction of a management infrastructure on top of existing variants, which allows to mitigate the shortcomings of cloning while leveraging its benefits. These two directions, in fact, coexist and complement each other within a single organization during the transition from ad-hoc to managed reuse – a process that usually takes several years.

Based on empirical evidence, we build the foundations for the management of cloned product variants, contributing a framework that specifies a set of conceptual operators required for variant maintenance and evolution. We show that these operators provide the necessary building blocks that can be composed to realize complex scenarios related to the management of clones, during both the unification and the maintenance of existing variants. We discuss possible implementations of the operators, mapping them to existing work when available. We also provide implementation for some of the missing functionality, and identify opportunities for future research.

We believe that the operator-based view promoted by this work provides a systematic approach for understanding the required, existing and missing functionality. It thus transforms the effort of introducing managed reuse in organizations that employ cloning from opportunistic to predicted, controlled and reproducible. Our work supports both researchers and practitioners interested in improving reuse practices in families of related products, promoting an incremental and staged transition from ad-hoc to well-managed reuse.
Acknowledgements

Everything I have ever accomplished is because of my Mom. Even though she did not get to see me pursuing a doctoral degree, she is always with me in my mind, every single day that passes.

Down here on earth, I was extremely fortunate to work with an amazing supervisor, Professor Marsha Chechik, who is not only an outstanding professional but also an empowering, supportive and generally great individual. She taught me to look at the core of problems, gave me tools for developing scientific solutions and significantly polished my research skills. My professional identity was shaped by her deep knowledge, creativity and unwavering desire to produce foundational results. Over and above that, she was always there for me and provided a lot of support when I needed it the most. I could not wish for a better supervisor.

Many ideas presented in this thesis were influenced by discussions with Professor Krzysztof Czarnecki from the University of Waterloo, who was part of my supervisory committee and served as an additional technical mentor for this work. I am very grateful to him for his insightful input, knowledge, creativity, support and encouragement. His insistence on studying problems “in the wild” before offering solutions led to a number of empirical studies that truly helped shape this work.

I am also grateful to the internal members of my supervisory committee, Professor Steve Easterbrook and Professor Sheila McIlraith, for their patience and for giving me direction towards improving and completing the thesis. Thorough and insightful comments of my external examiner, Professor Paul Grünbacher, helped make this thesis better and are very much appreciated as well.

My work benefited from discussions with many people in the University of Toronto and elsewhere in Canada, the US, Israel and Europe. It would be impossible to list them all, but I would like to extend a special thanks to Rick Salay, Michalis Famelis, Anya Tafliovich and Andrei Kirshin for being great colleagues and also great friends and for helping me make it through. In addition, I would like to recognize the colleagues that helped me to get started and introduced me to the product line community, such as Alan Hartman, Goetz Botterweck, David Weiss and, yet again, Krzysztof Czarnecki. Also, Andrian Marcus was always there to discuss and lend a supportive ear.

Last but not least, I am thankful to my husband Yoav and to my kids Shira and Tamir for their patience and continuing support. I never would have done it without you!
# Contents

1 Introduction  
1.1 Software Product Lines in Industry  
1.2 Cloned Software Product Variants  
1.3 Contribution of This Thesis  
1.4 Organization  

2 Cloning Practices in Industry  
2.1 Research Methodology  
2.1.1 Participants  
2.1.2 Data Collection  
2.1.3 Data Analysis  
2.2 Results: Cloning Practices  
2.2.1 Efficiency  
2.2.2 Overhead  
2.2.3 Short-Term Thinking  
2.2.4 (Lack of) Governance  
2.3 Discussion  
2.3.1 The Individual Perspective  
2.3.2 The Technical Perspective  
2.3.3 The Organizational Perspective  
2.4 Threats to Validity  
2.5 Related Work  
2.6 Summary  

3 Preliminaries: Software Product Line Engineering  

4 Cloned SPL Management Framework  
4.1 Key Observations  
4.2 Operators  
4.3 Composition of the Operators  
4.3.1 Sharing Features Between Variants  
4.3.2 Merge-Refactoring Variants into an SPL Representation  
4.4 Implementing the Operators  
4.5 Related Work
5 Cloned SPL Management Framework – An Experience

5.1 Research Methodology and Case Studies

5.1.1 Case Study Selection

5.1.2 Data Collection and Analysis

5.1.3 Description of the Case Studies

5.2 Analysis of Merge-Refactoring Activities

5.2.1 Case Study #1

5.2.2 Case Study #2

5.2.3 Case Study #3

5.3 Analysis of Activities Related to Maintenance of Existing Clones

5.4 Limitations and Threats to Validity

5.5 Summary

6 Locating Features in Cloned Variants

6.1 Example

6.2 Background

6.3 Distinguishing Features and Diff Sets

6.4 Diff Set-Based Heuristics

6.4.1 Improving Precision

6.4.2 Improving Recall

6.4.3 Combining Approaches

6.5 Empirical Evaluation: Experimental Setting

6.5.1 Subject Software Systems

6.5.2 Implementation

6.5.3 Methodology and Measures

6.6 Results

6.6.1 Analyzing Individual Heuristics

6.6.2 Combining the Heuristics: the \(PB_{m}\mathcal{F}\) Algorithm

6.6.3 Per-Project Analysis

6.6.4 Threats to Validity

6.7 Discussion and Related Work

6.8 Summary

7 Merge-Refactoring of UML Models

7.1 Example

7.2 Model Representation

7.3 Model Merging

7.4 The Merge-In Algorithm

7.4.1 Merge-In Definition

7.4.2 Proof of Correctness

7.5 Quality-Based Merge-Refactoring Framework

7.5.1 Explicating The Refactoring Goal
7.5.2 Constructing the desired refactorings ..................................................... 84
7.6 Implementation ............................................................................................. 84
7.7 Experience ..................................................................................................... 86
  7.7.1 Subjects .................................................................................................. 86
  7.7.2 Results .................................................................................................... 86
  7.7.3 Threats to Validity .................................................................................... 88
7.8 Related Work ................................................................................................. 88
7.9 Summary ........................................................................................................ 90

8 N-Way Model Merging .................................................................................... 91
  8.1 Example ....................................................................................................... 92
  8.2 N-Way Model Merging ................................................................................ 94
    8.2.1 Definition of the Operators .................................................................. 94
    8.2.2 Towards Implementation of the Operators ......................................... 95
  8.3 N-Way Matching via Weighted Set Packing ............................................... 96
    8.3.1 Weighted Set Packing and Its Approximation Algorithms ............. 97
    8.3.2 Preliminary Evaluation ....................................................................... 98
  8.4 Polynomial-Time Approach to N-Way Matching ..................................... 98
    8.4.1 Subset-Based Approaches ................................................................ 99
    8.4.2 The Algorithm NwM ......................................................................... 99
  8.5 Evaluation .................................................................................................... 104
    8.5.1 Theoretical Evaluation ...................................................................... 104
    8.5.2 Empirical Evaluation ......................................................................... 105
    8.5.3 Threats to Validity ............................................................................ 108
  8.6 Related Work ............................................................................................... 108
  8.7 Summary ...................................................................................................... 109

9 Conclusions and Future Work ....................................................................... 110
  9.1 Summary .................................................................................................... 110
  9.2 Conclusions ................................................................................................ 112
  9.3 Future Research Directions ...................................................................... 113
    9.3.1 Short-Term Goals ............................................................................. 113
    9.3.2 Long-Term Goals ............................................................................. 113

Bibliography ....................................................................................................... 115

A A Survey of Feature Location Techniques ..................................................... 130
  A.1 Basic Underlying Technologies ............................................................... 131
    A.1.1 Formal Concept Analysis (FCA) ....................................................... 131
    A.1.2 Latent Semantic Indexing (LSI) ....................................................... 133
    A.1.3 Term Frequency - Inverse Document Frequency Metric ............ 134
    A.1.4 Hyper-link Induced Topic Search (HITS) ....................................... 135
  A.2 Classification and Methodology ................................................................. 135
  A.3 Feature Location Techniques ................................................................... 137
    A.3.1 Static Feature Location Techniques .............................................. 138
List of Tables

2.1 Interview participants (ordered chronologically) ........................................ 11
2.2 Results of the pre-interview attitude questionnaire, used to elicit participants’ agreement with certain statements. The table shows the number of participants checking each cell. .................................. 11
4.1 Main benefits and shortcomings of cloning (see Chapter 2 for details) ............ 28
4.2 Operators for Managing Cloned Variants .................................................... 31
5.1 Analyzed Case Studies .............................................................................. 39
5.2 Applicability of the Operators ................................................................. 52
6.1 Characteristics of projects used in evaluation .............................................. 61
6.2 Configuration Parameters ......................................................................... 64
6.3 Comparing original algorithms (Orig) to algorithms extended with a combination of Filtering (F), Pruning (P), Boosting (B), and Boosting with Modifiers (Bm) ..................... 65
7.1 Quality metrics ......................................................................................... 83
7.2 Weights \(W\) Used by Compare [85, 112, 178] .............................................. 87
7.3 Thresholds \(S\) Used by Match ................................................................. 87
7.4 Varying Class Similarity Threshold \(S_{\text{class}}\) .............................................. 88
8.1 Matching algorithms ................................................................................ 96
8.2 Execution time for Greedy on Hospital and Warehouse cases .................... 97
8.3 Weights and execution times of \(NwM\) compared to subset-based approaches. For each case, the results of \(NwM\) and the best subset-based approach are boldfaced .......................... 104
A.1 Similarity of the documents and the query in Figure A.5 as calculated by LSI. .... 134
## List of Figures

1.1 Two GPS products and their features. ............................ 2  
1.2 An overview of the approach leading to the cloned software product line management framework. 5  
3.1 SPLE: Domain and Application Engineering [119]. ............................ 23  
3.2 Feature-Driven Annotative SPL Representations. ............................ 24  
3.3 Vending Machine SPL Architecture. ............................ 25  
3.4 Vending Machine Variants Derived from the SPL Architecture in Figure 3.3. ............................ 25  
3.5 Vending Machine Feature Model. ............................ 26  
4.1 Managing Cloned Variants. ............................ 30  
4.2 Algorithm for Sharing Features. ............................ 33  
4.3 Transferring the Feature \texttt{Free} from Variant (c) to (b) in Figure 3.4. ............................ 34  
4.4 Algorithm for Merge-Refactoring Products. ............................ 34  
4.5 An Initial Sketch of the Knowledge-Based Library. ............................ 36  
5.1 Transition Activities in Case Study #1. ............................ 43  
5.2 Transition Activities in Case Study #2. ............................ 47  
5.3 Transition Activities in Case Study #3. ............................ 49  
6.1 A snippet of the Eclipse call and data flow graph, containing the \texttt{Emacs-style incremental search} feature. ............................ 56  
6.2 Cloned product variants. ............................ 61  
6.3 F-measure and recall for the analyzed projects (%). ............................ 68  
7.1 Fragments of Washing Machine models and some of their refactorings. ............................ 73  
7.2 Partial representation of the Controller Statechart in Figure 7.1(a). ............................ 75  
7.3 A sketch for the proof of Lemma 3. ............................ 80  
7.4 Merge-Refactoring Framework. ............................ 83  
8.1 Example UML Models. ............................ 92  
8.2 Optimal matches “across” models. ............................ 93  
8.3 Results of comparing \texttt{NwM} against subset-based algorithms. ............................ 103  
8.4 Relative performance of \texttt{Gr4↓} on the Random case. ............................ 107  
A.1 The automatic save file call graph snippet. ............................ 131  
A.2 The \texttt{MindMapMapModel} code snippet. ............................ 131
A.3 Formal context for the example in Figure A.1. Objects are method names, attributes are tokens of the names. ................................................................. 132
A.4 Concepts and the corresponding concept lattice generated for the formal context in Figure A.3. ................................................................. 133
A.5 Term-by-document co-occurrence matrix for the example in Figure A.1. Documents are method names, terms are tokens of the names and the query. ................................................................. 133
A.6 Vectorial representation of the documents and the query in Figure A.5. ................................................................. 134
A.7 Surveyed techniques and their categorization. ................................................................. 136
A.8 Underlying technology, program representation and input type of the surveyed techniques. ................................................................. 137
A.9 Criteria for selecting a feature location technique. ................................................................. 150

B.1 Washing Machine Model A. ................................................................. 153
B.2 Washing Machine Model B. ................................................................. 154
B.3 Washing Machine Model C. ................................................................. 155
B.4 Washing Machine Model A+B [threshold ≤ 0.70] ................................................................. 156
B.5 Washing Machine Model (A+B)+C [threshold ≤ 0.70] ................................................................. 157
B.6 Washing Machine Model A+B [threshold = 0.75] ................................................................. 158
B.7 Washing Machine Model (A+B)+C [threshold = 0.75] ................................................................. 159
B.8 Washing Machine Model A+B [threshold = 0.78] ................................................................. 160
B.9 Washing Machine Model (A+B)+C [threshold = 0.78] - Part 1 ................................................................. 161
B.10 Washing Machine Model (A+B)+C [threshold = 0.78] - Part 2 ................................................................. 162
B.11 Washing Machine Model A+B [threshold = 0.80] ................................................................. 163
B.12 Washing Machine Model (A+B)+C [threshold = 0.80] - Part 1 ................................................................. 164
B.13 Washing Machine Model (A+B)+C [threshold = 0.80] - Part 2 ................................................................. 165
B.14 Washing Machine Model A+B [threshold = 0.85] - Part 1 ................................................................. 166
B.15 Washing Machine Model A+B [threshold = 0.85] - Part 2 ................................................................. 167
B.16 Washing Machine Model (A+B)+C [threshold = 0.85] - Part 1 ................................................................. 168
B.17 Washing Machine Model (A+B)+C [threshold = 0.85] - Part 2 ................................................................. 169
B.18 Washing Machine Model A+B [threshold ≥ 0.90] - Part 1 ................................................................. 170
B.19 Washing Machine Model A+B [threshold ≥ 0.90] - Part 2 ................................................................. 171
B.20 Washing Machine Model (A+B)+C [threshold ≥ 0.90] - Part 1 ................................................................. 172
B.21 Washing Machine Model (A+B)+C [threshold ≥ 0.90] - Part 2 ................................................................. 173
B.22 Microwave Oven Model A ................................................................. 174
B.23 Microwave Oven Model B ................................................................. 175
B.24 Microwave Oven Model C ................................................................. 176
B.25 Microwave Oven Model D ................................................................. 177
B.26 Microwave Oven Model C+D [threshold ≤ 0.70] ................................................................. 178
B.27 Microwave Oven Model A+B [threshold ≤ 0.70] ................................................................. 179
B.28 Microwave Oven Model (C+D)+(A+B) [threshold ≤ 0.70] ................................................................. 180
B.29 Microwave Oven Model C+D [0.75 ≤ threshold ≤ 0.85] ................................................................. 181
B.30 Microwave Oven Model A+B [0.75 ≤ threshold ≤ 0.85] ................................................................. 182
B.31 Microwave Oven Model (C+D)+(A+B) [0.75 ≤ threshold ≤ 0.85] ................................................................. 183
B.32 Microwave Oven Model C+D [threshold ≥ 0.90] - Part 1 ................................................................. 184
B.33 Microwave Oven Model C+D [threshold ≥ 0.90] - Part 2 ................................................................. 185
B.34 Microwave Oven Model A+B [threshold ≥ 0.90] ................................................................. 186
Chapter 1

Introduction

1.1 Software Product Lines in Industry

Software surrounds us and drives our lives: most modern systems heavily rely on software. Managing the complexity of these software systems is a challenging task. It is even more challenging for Software Product Lines (SPL) – families of software product variants with similar yet not identical functionality, commonly developed and maintained by companies in the automotive, electronics, aerospace and defense domains.

To deal with the complexity of SPL development, Software Product Line Engineering (SPLE) emerged as a software engineering discipline that promotes predicted and managed software reuse. SPLE relies on identifying, tracking and manipulating common and variable artifacts – those software entities that are part of all products of the product line, and those that are specific to some, but not all, individual products, respectively \[78, 174, 30, 63, 119, 114\]. Common and variable artifacts are traced to features: high-level characteristics of product functionality. The set of all product line features, together with relationships among the features, is specified in a feature model \[78\]. A particular selection of features from a feature model defines a concrete product of an SPL.

Given a feature selection, the product is said to be derived from an SPL architecture. SPL architectures can largely be divided into two categories: compositional, which group product artifacts into distinct fragments and allow feature-driven derivation of a specific product by composing a set of fragments, and annotative, which assume that there is one “maximal” product in which annotations indicate the artifacts that realize a particular feature. A specific product is then derived by removing artifact corresponding to unselected features \[83, 82, 20\].

Numerous SPLE approaches promise to ease the product line development and maintenance effort, improve time-to-market and quality, reduce portfolio size, engineering costs and more \[30, 114\]. Yet, in practice, the adoption of such approaches is still limited and reuse of artifacts between products rather occurs ad-hoc, in an opportunistic manner. Popular forms of ad-hoc reuse include managing variations using makefiles and preprocessor directives, putting software components into a library in the hope that opportunities for reuse will arise, or just cloning an existing variant of an already released product and modifying it to fit the requirements of a new customer or market segment (the “clone-and-own” approach).

In this thesis, we focus on studying the latter approach. We aim at gaining a better understanding of current cloning practices, their benefits and shortcomings, and then at outlining support companies require for mitigating the identified shortcomings. We organize and systematize support that exists, identify remaining gaps and contribute to filling some of these gaps. Our work facilitates a shift towards structured, managed reuse in organizations that employ cloning.
1.2 Cloned Software Product Variants

Current Cloning Practices. The first step towards suggesting alternatives that can replace cloning is to gain a better understanding of the current cloning practices. Consider, for example a fictitious company GlobalCo that delivers GPS solutions (see Figure 1.1). GlobalCo develops an advanced product, GPS-Pro, that has the Trip Computer feature for monitoring the vehicle speed and the time to destination, and the Layered Map feature for overlaying graphical objects on the map. As the product has been tested and released to the market, GlobalCo’s market analysis reveals the need for a simplified and less expensive product variant, GPS-EZ. Layered Map is determined to be the only essential feature of this product, while Trip Computer should not be included.

The goal of the company now is to release the new product to the market as fast as possible. Efficient management of reuse, although important, is hardly considered a priority. The easiest way for the development team responsible for GPS-EZ to cope with this request is thus to branch the already tested code of GPS-Pro and remove the Trip Computer feature implementation from it. This is mainly due to benefits of cloning such as its simplicity, availability and independence of developers that it provides.

From this point on, however, the two product variants become independent from each other and their corresponding implementations grow apart. As the number of cloned variants and the differences between their implementations increase, it becomes difficult to keep track of changes made to each of the variants and propagate the changes between them. In our scenario, imagine that the development of GPS-Pro continues, adding the ability to show points of interests (POI) and Live Traffic Info as two additional layers on the map. The team also implements the ability to show 3D Buildings as an additional layer on the map. The team also implements an extension to the POI feature copied from GPS-Pro, but cannot immediately propagate the change back to that product because it is currently frozen towards a close release. The two products not only have a different set of features now, but two seemingly identical features have different implementations in the distinct products, challenging the portfolio management task.

Due to a lack of information about the dependencies between features, borrowing the Night Mode and Shortest Time Routing features from GPS-Pro is an additional challenge: the Night Mode feature might not work well with 3D Buildings, because it was not designed to work with that functionality as GPS-Pro does not contain it. Shortest Time Routing depends on the Live Traffic Info feature that is not available in GPS-EZ. Moreover, if GlobalCo now
decides to establish a new product variant that has the Layered Map, POI, Night Mode and 3d Buildings features, it is unclear which product should be used as a starting point, and how to remove their unnecessary features while borrowing the required features from other products.

The example demonstrates the reasons leading a company to rely on cloning to realize its SPL, together with several key challenges that the company faces, such as sharing features between product variants, reconciling changes made in distinct product variants and establishing new products. To address these challenges, the SPLE community commonly suggests to unify the cloned product variants into a single-copy SPLE representation, as discussed next.

**Approaches for Unifying Cloned Variants.** Usage of cloning is strongly discouraged in the SPLE literature [30, 63, 119, 95], and numerous authors propose solutions for re-engineering cloned products into a single-copy software product line representation. Some of these works, e.g., [14, 53, 79, 90, 68, 87, 75, 74], mainly report on experiences with manual analysis and re-engineering of individual systems. These approaches involve reviewing the design, implementation and documentation of a system in order to identify the set of product line features, as well as the set of artifacts which implement these features. Manual re-engineering is a time- and effort-consuming process, and, in many cases, it is hard to reuse the results of these works in a different context – for new organizations and development settings.

Other works attempt to automate (at least) a part of the functionality related to unification of cloned variants. For example, Koschke et al. [91] present an automated technique for comparing software variants at the architectural level and reconstructing the system’s static architectural view which describes system components, interfaces and dependencies, as well their grouping into subsystems. Ryssel et al. [148] introduce an automatic approach to re-organize Matlab model variants into single-copy representations while identifying variation points and their dependencies. Yoshimura et al. [180] detect variability in a software product line from its change history. Several works, e.g., [52, 95, 109] propose techniques based on clone detection for identifying similar code-level functions which can be moved to the product line “core”. Works on feature-oriented refactoring [103, 111] focus on identifying the code for a feature and factoring it out into a single module or aspect, with the aim to decompose the program into features. Each of the above works addresses a particular aspect of the re-engineering problem and makes several, largely different, assumptions on the structure of existing artifacts and the representation of the desired result. The applicability of these works in a context of another, different organization thus often remains unclear.

Additional approaches, developed in seemingly unrelated contexts, can also assist in the clone unification task. For example, feature location techniques (surveyed in Appendix A) allow tracing features to their corresponding implementation artifacts – a useful functionality when constructing a SPLE representation out of a legacy set of artifacts. Likewise, feature interaction techniques, studied in depth in the telecommunications domain [181], could contribute to identifying relationships between features of the legacy cloned variants. Yet, it is often unclear which of the available techniques are applicable in a specific context and what the assumptions that they make are.

**Gaps Identified in This Thesis.** Despite several isolated solutions aiming to assist the practitioners in unifying collections of related software product variants, there is no systematic approach that aims at understanding the support such practitioners require. The lack of a systematic view on this problem makes it unclear why the shift to SPLE is difficult, what the main problems are, how to identify and reuse existing work and what functionality is still missing.
The following questions then emerge:

Q1. Why is cloning still a method of choice in many industrial organizations?

Q2. What support does an organization require to improve its reuse practices and reduce maintenance costs? Can this support be provided by existing works and what are the remaining gaps?

Q3. How can (at least some of) the gaps be addressed?

By using a combination of exploratory and foundational approaches, this thesis aims at answering the above questions and also proposes techniques to fill some of the identified gaps.

1.3 Contribution of This Thesis

In this thesis, we empirically investigate benefits and shortcomings of using cloning to realize SPLs. We study development activities that a company performs when it relies on cloning and those that it performs when it re-engineers clones into a single-copy SPL representation. Based on the collected empirical evidence, we conclude that an efficient management of clones relies on both refactoring cloned variants into a single-copy product line representation and improving the development experience when maintaining existing clones.

We then develop a framework that specifies a basic set of conceptual operators which can be composed to realize the studied development activities in both scenarios. Our framework builds the foundations for the management of cloned product variants: focusing on the operators provides means to scope and structure the required automation, reuse existing work, and identify gaps and research opportunities. It also allows companies to estimate the investment they need to make in order to improve their reuse practices, as they can reason about the transition process in terms of existing and missing operator support. Further, we contribute solutions that fill some of the gaps identified in this work.

Specifically, this thesis aims at addressing the three questions identified above and makes the following main contributions:

1. A Study of the Cloning Culture in the SPL Context (Chapter 2).

Limited understanding of the nature of current cloning practices makes the attempts to transform collections of cloned products into representations promoted by contemporary SPLE approaches, as well as attempts to improve the SPLE approaches themselves, difficult and unfocused. We thus first conduct a study to gain more insights on the cloning culture in six different product lines realized via cloning.

Our study shows that companies that rely on cloning to realize their SPL often do not use advanced approaches proposed in the literature because they do not plan upfront to develop a product family. Instead, as in the GlobalCo example, the companies incrementally grow their product portfolios as they become successful and as new customers require modified variants of already existing products. Moreover, despite its shortcomings, cloning is still perceived as a favorable and natural reuse approach because of its benefits such as simplicity, availability and developer independence.

We draw two main conclusions from the study: first, any approach that aspires to be better than cloning has to address the great perceived benefits of cloning. Without such argument, many SPLE approaches fail to convince practitioners that they would yield better results. Second, the transition to a chosen SPLE approach is a time- and labor-intensive task by itself. The overhead related to this task should be comparably lower than the overall overhead related to the management of existing clones. Automation plays a crucial role here: mapping and organizing available support for automating individual transition tasks will streamline the SPLE adoption.
Software product lines are often created by \textit{cloning} and \textit{modifying} existing variants.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.2.png}
\caption{An overview of the approach leading to the cloned software product line management framework.}
\end{figure}

\section{A Cloned SPL Management Framework (Chapters 4 and 5)}

We empirically investigate development activities in three companies that rely on cloning to realize their SPLs. Our study shows that there are two main directions for improving reuse practices and reducing costs in such companies (see Figure 1.2). The first direction, promoted by the SPL\textsuperscript{E} community, relies on unifying cloned products into single-copy SPL\textsuperscript{E} representations (we call it a \textit{merge-refactoring} approach). In the GlobalCo example, this translates to merging the branches of GPS-Pro and GPS-EZ while identifying features of each product and relationships between these features, as well as unifying commonalities at the code level and tracing differences \textit{(a.k.a. variabilities)} to their corresponding features.

However, in real-life cases with numerous large and complex products, merge-refactoring is an incremental process which is often delayed. When started, the process takes several years \cite{68, 75, 74}. Moreover, it initially considers only a subset of the products and usually targets future releases, while (most of) the existing ones are still maintained as distinct clones. Hence, focusing on merge-refactorings only is insufficient for the needs of realistic companies.

We claim that it is necessary to complement merge-refactoring with an additional direction that allows building an efficient management infrastructure on top of existing products, mitigating the shortcomings of cloning while leveraging its benefits (we call it a \textit{clone-based SPL\textsuperscript{E}} approach). The management infrastructure helps detect and capture essential product line constructs such as features, dependencies between features, as well as traceability information between the features and their implementation artifacts. For the GlobalCo case, it provides a global view on the set of specific changes performed by each team, making it apparent that the POI feature works differently in GPS-Pro and GPS-EZ, and that the Shortest Time Routing feature requires the Live Traffic Info in order to operate. Such knowledge, collected incrementally, also eases a future merge-refactoring, if an organization decides to do so at a later stage.

Considering both the merge-refactoring and clone-based SPL\textsuperscript{E} approaches, we build the \textit{cloned SPL management framework} – a set of seven composable operators allowing to realize development activities related to cloned SPL\textsuperscript{E}s management (see Figure 1.2). Our goal is to provide a systematic view on the problem and to show that the proposed set of operators is reasonable to typical real-life activities that we identify in industrial settings (rather than to produce a \textit{complete} set of operators that can accommodate \textit{any} real-life scenario).

The operators identified in this work are listed below and discussed in detail in Chapter 4.

- \textit{findFeatures} identifies features implemented by an SPL product. For the GlobalCo example in Figure 1.1,
these are listed next to each product.

- **findFeatureImplementation** (a.k.a. *feature location*) allows to trace each feature to its corresponding implementation artifacts.

- **dependsOn?** determines whether one feature of a product variant requires another in order to operate, e.g., the features Shortest Time Routing and Live Traffic from GPS-Pro.

- **same?** determines whether two seemingly identical features from distinct product variants are indeed identical, e.g., the POI features in GPS-Pro and GPS-EZ.

- **interacts?** determines whether two sets of features from distinct product variants can operate together, e.g., the features Night Mode and 3D Buildings from GPS-Pro and GPS-EZ, respectively.

- **merge** puts together distinct product variants or a partial set of these variants’ features.

- **reorganize** improves an SPL architecture by a semantic-preserving refactoring that maintains the original set of products.

By breaking processes down into logical components and producing the operator-based view for organizing the cloned product variant management tasks, our framework allows to identify the functionality that is required, already exists and is still missing – an effort that is expected to assist both practitioners and solution developers. Creating a body of knowledge around the implementations of the operators, i.e., a library of possible implementations for each operator, together with the assumptions that each implementation makes, can facilitate an efficient management of cloned variants and enable reuse across organizations and domains. We see the proposed framework as a first step towards providing a “roadmap” for both practitioners that manage their cloned product variants and researchers / solution developers that aim at assisting such practitioners.

3. Steps Towards Realization of the Framework (Chapters 6, 7 and 8).

We take several concrete steps towards the realization of operators that were missing in the scenarios that we considered.

1. **Find Feature Implementation** (a.k.a. *feature location*).

First, we take a closer look at the problem of feature location in a collection of related product variants. While there are more than 20 different feature location techniques available (see Appendix A for a detailed survey), their accuracy is still limited. Most of the techniques also do not specify the kind of features they detect. In our work, we focus on locating *distinguishing* features – those that are present in one variant while absent in another. These features are of a premier importance for both the cloned-based SPL-E and merge-refactoring approaches: artifacts that correspond to distinguishing features should be identified in one variant in order to reuse it in another or treated properly when building an SPL-E representation by relating them to their corresponding features (while similar treatment of the common features can be postponed).

We leverage information available when considering multiple product variants together for extending the set of heuristics used by existing feature location techniques when locating distinguishing features, substantially improving the accuracy of these techniques. Such information is obtained by comparing the code of a variant that contains a particular feature of interest to the one that does not. The comparison provides an initial “coarse-grained partitioning” of a program into relevant and irrelevant parts and assists the feature location process: distinguishing features are implemented in the unshared parts of the program, providing a clue where to locate them.
2. merge.
As a realization of the merge operator for UML models and model fragments, we contribute an algorithm for merging multiple UML model fragments containing class and statechart diagrams – a common choice for companies that use commercial tools such as IBM Rhapsody\(^1\) to automatically generate fully-functional code from their models. Our algorithm is semantically correct: it ensures that the resulting merged model is equivalent to the set of the original inputs (it is common to define the semantics of an SPLE representation via a set of all products that it represents).

Next, we show that several semantically correct but syntactically different merge results are possible, raising the need to choose one that is more preferable than the others. We thus propose an approach that captures the merge goal explicitly, i.e., by means of a syntactic quality function. We use this explicitly captured goal to guide the merge process towards a result that best fits the user’s intention. An evaluation of the approach on a set of case studies, including a large-scale case of an industrial partner, validates the contribution of this work to the automation of the product line merge operator and reasoning about merge alternatives.

Finally, we show that considering all input models simultaneously during model matching and merging is more beneficial than processing them in any particular order – an approach used by all state-of-the-art techniques so far. We also show that simultaneous matching of three or more models can be reduced to the weighted set packing problem, which is known to be NP-hard; and that existing approximation algorithms for weighted set packing do not scale beyond very few models with a small number of elements in each. We thus devise a scalable heuristic algorithm that operates on a set of all input models simultaneously and demonstrate that it achieves substantial improvements over approaches used in practice, without any degradation in performance.

1.4 Organization

The remainder of this thesis is structured as follows.

- Chapter 2 sets the scenes for the thesis by presenting an exploratory study on cloning practices in industry. It is based on a publication in [41]. We introduce the research methodology that we used for the study, describe the case studies and then discuss the benefits and shortcomings of cloning, as perceived by industrial teams involved in development of six different software product lines. We focus on human, technical and organizational perspectives. Based on our observations, we outline issues preventing the adoption of SPLE approaches, and identify future research directions.

- Chapter 3 sets the terminology and formalizes the SPLE concepts that we use in this thesis.

- Chapter 4 introduces the cloned SPL management framework. It is based on our publications in [140, 145]. We start by listing requirements for an efficient management of clones. These are inspired by an empirical analysis of realistic scenarios observed in industrial settings and include two directions: refactoring cloned variants into a single-copy product line representation and improving development experience when maintaining existing cloned variants. We then present the seven conceptual cloned SPL management operators that constitute the framework, demonstrate their composability and their applicability to the above directions. We discuss possible implementations of the operators, including synergies with existing work developed in seemingly unrelated contexts, and describe our vision of building a shared body of knowledge around the operator implementations. The discussion aims at helping researchers and practitioners to

\(^1\)http://www-03.ibm.com/software/products/en/ratirhpfami
understand and structure existing work, identify opportunities for reuse as well as opportunities for future research. We conclude by discussing related work.

• Chapter 5 grounds the framework presented in the previous chapter in empirical evidence and exemplifies its usefulness. It is based on our publications in [144, 145]. We describe in detail three case studies of organizations with cloned product lines and analyze their development activities. We show that these activities relate to the merge-refactoring and clone-based SPLE approaches. We break the activities down into individual steps and demonstrate the applicability of the operators for describing them. We conclude by discussing limitations and threats to validity of our analysis.

• Chapter 6 presents our contribution to the realization of the feature location operator, which is optimized to work on a set of cloned product variants. This chapter extends our publication in [139]. We first give the necessary background on feature location approaches and then specify the exact focus of this work – locating distinguishing features that are present in one variant of a program but absent in another. We propose a set of heuristics for improving the accuracy of existing feature location techniques when locating distinguishing features. Our heuristics are based on diff sets – the information obtained when comparing program variants to each other. We empirically study the nature of the proposed heuristics, comparing them to each other and evaluating their effectiveness on a large number of case studies. We conclude by discussing related work.

• Chapter 7 presents the merge operator for UML models containing class and statechart diagrams and is based on our publications in [138, 143]. We start by giving formal foundations of model merging and define our merging-based product line refactoring technique. Our technique relies on identifying commonalities and variabilities in existing products and further merging those into product line representations which reduce duplications and facilitate reuse. We prove semantic correctness of the technique and show that several semantically correct, yet syntactically different refactoring results are possible. We thus propose to automatically guide the refactoring towards the results preferred by the user and captured using a syntactic quality function. We describe our experience varying the quality function when combining product variants in three different case studies. We conclude by discussing related work.

• Chapter 8 extends the work presented in Chapter 7 by discussing an approach for simultaneous merging of multiple models, a.k.a. the n-way merge. This chapter is based on our publication in [142]. We start by defining the n-way merge problem and show that it can be reduced to the weighted set packing problem which is known to be NP-hard. We then study the applicability of existing approximate solutions, and show that these solutions do not scale for real-size software models. We thus evaluate alternative approaches of merging models that incrementally process input models in smaller subsets and propose our own algorithm that considerably improves precision over such approaches without sacrificing performance. We evaluate our algorithm on a large set of inputs, including two sets of real life model sets with 8 and 16 models in each. We conclude by discussing related work.

• Chapter 9 summarizes the research in this thesis and identifies possible directions for future work.

• Appendix A presents a detailed survey of existing feature location techniques. It is based on our publication in [141].

• Appendix B provides the full set of models used as case studies in the merge-refactoring approach discussed in Chapter 7.
Chapter 2

Cloning Practices in Industry

Software Product Line Engineering (SPLE) approaches support development of products from a common set of core assets in a prescribed way [30, 63, 119]. These approaches advocate strategic, planned reuse that yields predictable results. However, in reality, software product lines often emerge ad-hoc, when a company has to address its target market needs by releasing a new product that is similar, yet not identical, to existing ones. In many cases, artifacts of an existing product are cloned and modified to fit the new requirements – the “clone-and-own” approach [52, 109, 50].

Usage of the “clone-and-own” approach is discouraged in the SPLE literature, and numerous authors propose solutions for re-engineering cloned products into a single-copy software product line representation [52, 109, 56, 77, 148, 180]. Why is cloning then still a popular method of choice in many industrial organizations?

The first step towards suggesting alternatives that can replace cloning is to gain a better understanding of the current cloning practices. Are the organizations satisfied with these practices? What are the benefits and shortcomings of the approach? Do the organizations wish to eliminate or minimize cloning? And if so, what prevents them from doing that? Surprisingly, to this date, there is no systematic study that investigates these questions, and little empirical knowledge is available. Without such knowledge, understanding and improving the current cloning practice remains difficult and unfocused.

In this chapter, we address this gap by reporting on an exploratory study that we conducted to investigate the cloning culture in the context of software product lines. Specifically, we performed eleven semi-structured interviews with participants involved in development of six cloned product lines in three large-scale organizations providing solutions in the data storage, aerospace and defense, and automotive domains. We interviewed employees from a variety of roles, such as developers, testers and product managers. We used a structured questionnaire to complement the interview data and analyzed the collected data using the grounded theory approach [59].

Among our findings, we observed that the majority of practitioners that develop product lines using cloning techniques still perceive it as an effective reuse approach, mainly because it is a rapidly available mechanism that allows practitioners to start from an already specified and verified functionality, while having the freedom and independence to make any necessary modifications. In fact, several practitioners are satisfied with the cloning practices and believe that it is a viable reuse mechanism. While others would wish to shift to a better managed approach, we observed that existing organizational structures might impede such a shift because there is usually no role in an organization that is responsible for promoting reuse.

We use these findings to drive our research agenda and set the focus of this thesis. We also develop a set of recommendations for practice and suggestions for future research. Our analysis addresses organizational structure,
governance models, and tool support required to efficiently develop and manage families of related software product variants.

**Contributions.** This chapter makes the following contributions.

1. We outline benefits and shortcomings of cloning, as collected in an exploratory study that investigated processes and stakeholder perceptions in six industrial cloned software product lines.

2. We note that cloning, while widely discouraged in the literature, is still perceived as a favorable and natural reuse approach by the majority of practitioners in the studied companies. This is mainly due to its benefits such as simplicity, availability and independence of developers.

3. We discuss issues preventing the adoption of systematic software reuse approaches, and identify future research directions.

This chapter is based on our publication in [41]. We proceed as follows. Section 2.1 introduces our research methodology. Section 2.2 reports the main results, followed by a discussion on their implications in Section 2.3. Section 2.4 outlines limitations and threats to validity, and Section 2.5 summarizes related work. Finally, Section 2.6 concludes.

### 2.1 Research Methodology

Our study had an exploratory, theory-building nature: its objective is to understand how product cloning happens in practice and what its underlying mechanisms are, rather than confirm or refute any specific hypotheses. We thus based our study on *grounded theory* – a methodology that allows building a *theory* from data collected during exploratory qualitative analysis [59, 31]. Grounded theory recently became popular in qualitative studies related to Software Engineering [2], we thus follow this method here. Below, we describe the selection of participants for the study, as well as our data collection and analysis techniques.

#### 2.1.1 Participants

As summarized in Table 2.1, we interviewed eleven practitioners involved in development of six software product lines in three different enterprises from the aerospace and defense (A&D), data storage management (DSM), and automotive (Auto) domains. Our goal was to select companies from a wide range of domains, backgrounds and sizes. We limited our selection to those that use cloning to realize their product lines and applied convenience sampling, i.e., we approached companies that we already had access to. Confidentiality issues prevent us from sharing the names and exact details about these companies and product lines they develop.

At the time of the study (2012), the first product line (PL1) was relatively new: it was supported for around three years. The second and the third ones (PL2 and PL3) were mature product lines developed over eight to ten years each. The forth one, PL4, was initialized one and a half year before the study started as a proof of concept. By the time of the study, some of its products were already commercialized as they became successful. PL5 was developed over the course of five years and, at the time of the study, was in the process of merging with a larger product line, due to a company acquisition. Like PL2 and PL3, multiple products of PL6 existed for almost ten years. This product line was constantly growing, with new products emerging continually up to the time of the study. Almost all products of these six product lines were still sold on the market.
Table 2.1: Interview participants (ordered chronologically)

<table>
<thead>
<tr>
<th>Participant</th>
<th>Industry</th>
<th>Product Line</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>A&amp;D</td>
<td>1</td>
<td>Software leader</td>
</tr>
<tr>
<td>p2</td>
<td>A&amp;D</td>
<td>2</td>
<td>Senior software leader</td>
</tr>
<tr>
<td>p3</td>
<td>A&amp;D</td>
<td>3</td>
<td>Integrator &amp; QA engineer</td>
</tr>
<tr>
<td>p4</td>
<td>A&amp;D</td>
<td>4</td>
<td>Architect</td>
</tr>
<tr>
<td>p5</td>
<td>A&amp;D</td>
<td>2</td>
<td>Developer</td>
</tr>
<tr>
<td>p6</td>
<td>DSM</td>
<td>5</td>
<td>Architect</td>
</tr>
<tr>
<td>p7</td>
<td>DSM</td>
<td>5</td>
<td>Technical leader</td>
</tr>
<tr>
<td>p8</td>
<td>DSM</td>
<td>5</td>
<td>Senior architect</td>
</tr>
<tr>
<td>p9</td>
<td>DSM</td>
<td>5</td>
<td>Developer</td>
</tr>
<tr>
<td>p10</td>
<td>DSM</td>
<td>5</td>
<td>Technical leader</td>
</tr>
<tr>
<td>p11</td>
<td>Auto</td>
<td>6</td>
<td>Technical leader &amp; developer</td>
</tr>
</tbody>
</table>

Table 2.2: Results of the pre-interview attitude questionnaire, used to elicit participants’ agreement with certain statements. The table shows the number of participants checking each cell.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Not at all</th>
<th>To a great extent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. I am aware of a product line development strategy in my organization</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2. My team develops core assets that are later used by other teams</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>6. We regularly clone pieces of code</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>10. We measure how many times a certain core asset is used</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>17. People in my team know who should approve a change request</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>19. We have work procedures that include cloning of artifacts</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>20. I feel that our development process is well-defined</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>21. We clone to reuse artifacts between products of a product line</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>22. In practice, all project teams follow the defined process</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>23. We have many clones of code</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>27. We do not change main APIs without receiving an approval</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>31. We have relatively big clones of code</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Development and maintenance of each product line involved between 26 and 100 people. To gain a broad perspective, we selected representative roles that have decision rights and responsibilities at various development stages (see column “Role” in Table 2.1). Applying theoretical sampling [59], after the fourth interview we decided to extend our understanding by going back and interviewing another participant from PL2 (participant p5). This was done to gain an extended perspective on the product line governance issues, involving a number of members from the same team. Also, in order to gain team perspective, we interviewed five participants from the same product line (participants p6–p10).

2.1.2 Data Collection

We used a semi-structured interview and a pre-interview attitude questionnaire [118] to collect our data. Combining both research tools allowed us to enrich our understanding of the studied concepts and also triangulate data during the analysis.
**Questionnaire.** The interviewees were asked to spend between five to ten minutes filling in the questionnaire right before starting the interview. The goal of the questionnaire was to gather interviewees’ initial perception, before starting the conversation. When filling in the questionnaire, the interviewees were asked to think about the product line they work on and to indicate the extent to which each of the questionnaire statements describes it. Together with statements about the cloning practices, the questionnaire included statements that relate to the product line setting in general.

Table 2.2 presents an aggregated view of answers to some of the questionnaire’s statements. We only show statements that strongly support or contradict the derived theory (Section 2.2) and are relevant to our discussion (Section 2.3). The number in each cell indicates the number of participants who marked this cell. For example, six participants indicated that they extensively clone to reuse artifacts between products of a product line (two rightmost columns in statement #21).

**Interviews.** Following the questionnaire, we conducted semi-structured interviews that included open-ended questions and took about one hour. Each interview started with general questions regarding the product line environment and the interviewee’s role, and then focused on the way the cloning is performed. For example, we asked the interviewees to describe the product line they are involved with, its lifecycle and phases, and the tools used.

Specifically, we asked about the capabilities of the tools as well as about the roles and activities that are directly related to the product line development. We investigated the way cloning is implemented, e.g., who decides to make a clone and how the information about existing clones is maintained. We asked participants to describe which artifacts are cloned and at which phases, the size and the number of the clones, etc. We were also interested to find out whether there are procedures that regulate the cloning process and whether there are any measurements in this respect.

During the course of the study, we adjusted and evolved our research tools, as recommended in [31]. For example, a detailed questionnaire that was used in the first interview was subsequently shortened to the attitude questionnaire described above. We also used information obtained during interviews to refine questions that we then asked subsequent participants.

### 2.1.3 Data Analysis

We used grounded theory’s open coding method [31] to analyze the collected data. The transcripts of our interviews and the answers to the questionnaire were analyzed line by line to detect concepts – key ideas contained in data. When looking for concepts, we searched for the best word or two that describes conceptually what we believe is indicated by the raw data. Based on the concepts, we created more abstract categories that group lower-level concepts and represent threads of ideas that we found in the analyzed data. We then related concepts to each other, forming a hierarchy of concepts and sub-concepts. This hierarchy is presented in Section 2.2.

In grounded theory, data collection and analysis are interrelated processes: as mentioned above, we used evolving concepts and categories to refine interview questions and involved additional participants in our study when we identified gaps in our understanding of the concepts (i.e., applied theoretical sampling). This process continued until no new concept emerged; thus, conceptual saturation was reached. Since all findings are linked to specific evidence – either an interview quote or a questionnaire response – we are able to provide grounding for the results we report and attribute them to the collected data.
2.2 Results: Cloning Practices

This section presents the findings of our study. Following the analysis of the interviews and the questionnaires, we learned that cloning happens at all stages of software development, from requirements through design, implementation and testing, up to the preparation of user documentation: artifacts in each stage are cloned and adjusted to the needs of a new customer. Practitioners, however, indicated that in their projects, code is cloned more often than other artifacts.

All but three practitioners indicated that, by design, their products should be built on top of a platform that contains reusable code. However, this platform is rarely updated and cloning is frequently applied as the main reuse mechanism. A team lead of a product indicated that:

“there are [core, reusable] assets but they are just place-holders; < ... > [core asset] repository does not contain significant assets, but the intention is there.”

We also observed that clones can be of different size, from a few methods or single components up to significant sub-products. Below, we identify and summarize four major characteristics (i.e., grounded theory categories) related to the use of cloning in the software product lines context, which are further discussed in Sections 2.2.1–2.2.4. When presenting our empirical observations, we do not attempt to assess their positive and negative implications; these implications are discussed in Section 2.3.

Efficiency: Cloning is perceived to be a simple yet efficient reuse mechanism that saves time and resources. It allows participants to start the development from already implemented and verified set of artifacts. At the same time, it provides independence and freedom to change these artifacts as needed.

Overhead: Adapting cloned artifacts to the new needs sometimes involves a significant effort. Effort in maintaining the artifacts can also be increased because some tasks need to be performed on each cloned copy. Also, propagating modifications between clones is not a trivial task.

Short-Term Thinking: Lack of resources to invest in the systematically managed reuse, as well as lack of awareness of other reuse approaches, leads to choosing cloning as the favorite reuse mechanism. Organizations often focus on making sure their individual products are successful and postpone dealing with reuse issues to the future.

(Lack of) Governance: Knowledge about reuse is rarely maintained. Reuse is not measured and there are usually no roles in the organization that are responsible for reuse practices and processes.

In what follows, we discuss these observations in more detail and illustrate them with the corresponding quotes from the participants.

2.2.1 Efficiency

There are three major reasons for considering cloning as an efficient reuse mechanism in software product line engineering.

1. Cloning saves time and reduces costs: Most interviewees stated that it is faster and more efficient to start with an already developed and tested set of artifacts:

   “It is easier to start with something. Cloning gives [us] an initial basis.”

   “< ... > there is no need to reinvent the wheel.”
“The most significant thing is to take a ‘stack’ as is and reuse it. This can save thousands of hours.”

“We want to make the development faster. The work assumption is that we clone.”

Moreover, most participants highlighted the importance of reusing the already validated code:

“It saves time. These components were already used, tested, closed. A kind of an off-the-shelf software.”

“We did something. It is ‘old’ and for most cases it is stable. The amount of time to bring [new code] to the required level of quality is not easily estimated.”

2. Cloning provides independence: After cloning, developers enjoy the freedom of making any necessary changes to their clones. They do not have to worry about synchronization with other teams, neither about the form of the common artifacts nor about their lifecycle and schedule.

“It gives freedom to change, [when cloning] there is no damage to existing products.”

“The management is more convenient. We believed that this code will be significantly changed so no point in keeping it as a shared code.”

“No code sharing, [because] the two copies are not developed in the same pace.”

Also, one participant emphasized the understandability and readability of the cloned product-specific code compared to the more complex reusable code:

“[In the past,] a new variant was integrated back into the mainstream by using preprocessor switches. This has made the code very unreadable, so we wanted to go away from that and we started to branch off the files that differ among variants.”

3. Cloning is the most available mechanism: Some of our participants indicated that there are no other readily available reuse mechanisms:

“There are simply no other options.”

“There was no other choice. [But] we need to avoid doing cloning, it has a price.”

2.2.2 Overhead

As noted in the last quote, cloning practices have their price: organizations usually have to deal with the overhead involved in creating and managing the clones. We identified four major issues introduced by cloning.

1. Propagating changes between clones is difficult: Most interviewees stated that no connection between clones is maintained. Thus, it is difficult to make sure that changes and bug fixes made to one of the clones are propagated to the others.

“< ... > code that we cloned loses connection with the product which it is cloned from, and then there is no sharing of new insights and innovations.”

“Sometimes, we find the same bug again in a different variant that nobody thought about before.”
2. Integration of the cloned artifacts is difficult: In some cases, practitioners stated that adapting clones to the new needs involves larger effort than expected.

“In this process we always lose quality. Sometimes, we have no choice but to throw away the code and re-write.”

“It is usually not possible to port without making changes to the code.”

 “[They] took an existing asset and tried to reuse. They claimed that integration duration was too long.”

“Sometimes clones are too big for relatively small needs.”

“It is a copy and a lot of adaptation.”

3. Repetitive tasks are common: Some participants indicated that cloning causes a significant increase in the maintenance effort.

“If we find a bug then many times it can be here and also in other places. The new product contains code that exists also in the old product. So, if we fix the old one then we also fix the new or vice versa.”

“We need to perform many activities several times: for each variant, we have to check the code and implement the change or fix. Then, the design and documentation documents, as well as the test specification need to be adapted for each variant. Tests need to be run.”

4. It is not clear which variant to use as the source for cloning: One participant indicated that when several clones exist, it is not always clear which one is the best “starting point” for cloning.

“With each new project we ask: ‘where do you start from’?”

2.2.3 Short-Term Thinking

As stated in Section 2.2.1, sometimes practitioners perceive cloning as the only available reuse mechanism. This often occurs because development organizations invest little time and resources in supporting and managing reuse. Below are three major factors reflecting organizational thinking with respect to reuse.

1. Lack of planning: Most practitioners stated that thinking ahead and planning for reuse is rare.

“We were not aware we develop product lines.”

“When a new customer came, we needed to decide how to implement his requirements in the fastest way. We do not have time to think thoroughly about generic approaches.”

Sometimes it is unclear in the beginning of product development that reuse would be needed:

“At the beginning we did not know that we will have to support all the controllers that we support now – this emerged over time.”

“Maybe we can [think about reuse] from the beginning. Still this is easy to say now, when we know that the first product is a success. At the beginning, the other risks are more important.”
2. **Lack of resources**: Our interviewees were concerned with the lack of support for product line engineering practices.

“There is a lack in resources for an organized work and methodology with respect to the product line engineering.”

“We sometimes need to beg for reuse.”

3. **Unawareness of other approaches**: In some cases, cloning is considered to be “state-of-the-art” in reuse.

“We clone code and should do better with cloning requirements and design.”

“If something is good < ... > then it will be cloned.”

Only in one case, the participant explicitly stated that even though other approaches were considered, the company deliberately chose cloning as their reuse approach:

“We explicitly decided to use separate branches for our variants.”

### 2.2.4 (Lack of) Governance

All but one of the participants clearly stated that no governance of product line development exists in their organizations. We identified three issues related to the (lack of) product line governance.

1. **Lack of reuse tracking**: All practitioners indicated that no infrastructure that tracks and facilitates reuse opportunities exists. Information about cloned artifacts exists primarily in people’s mind and they are responsible to make sure that changes between clones are propagated correctly. Similarly, reuse opportunities are identified if somebody remembers which similar artifact can be reused. Reuse is done via personal knowledge, memory and networking.

“When requirements are given, the software leader and most of us know if there is already such a thing.”

“I am aware of things that I did and saw. If I recall something similar to what I need, I’ll find it and copy.”

“A person who was in a specific team takes the capabilities to another project.”

“Many things are in the heads of people: ‘why don’t you use what we did?’”

2. **Lack of organizational roles and processes**: In most cases, there are no organizational roles that are responsible for reuse. Many times, project leads are those who actively look for reuse opportunities, in order to reduce costs of their tasks. However, there were no participants who indicated that they are encouraged to contribute reusable artifacts.

“No one [is responsible for reuse]. One who requires an asset, takes it.”

“No one is in charge of the cloning knowledge – in practice, it is the one who implements [a functionality] and the architect who is in charge of the work item.”

“The decision to do cloning is probably done by a manager or an architect.”
“The responsibility at the end is on the software leaders.”

“In each project, there is a software leader who manages the software development activities as part of the system development. He or she is in charge of using existing assets.”

Only in two cases, the interviewees indicated that there is an organizational or technical structure that mandates the process of reuse.

“There is an architecture forum that is being led now by the person who is in charge of the assets management. There are members of all the disciplines in the department in this forum. The forum’s role is to manage the department assets and identify artifacts that can be reused.”

“We have a pool of components or files that are meant to be reused by the projects (using branching). However there is no dedicated group that maintains these assets; this is the responsibility of the projects.”

In another case, a participant stated that there is a person who can technically clone artifacts when asked to.

“Developers are not supposed to clone. It happened at the beginning, but not later. The configuration manager has a procedure how to do it.”

However, similarly to the lack of roles, there are mostly no processes that define when and how to clone artifacts.

“There is no place or procedure that asks to search for existing assets.”

“It is not perceived as a process. It is simply something that is done.”

3. Lack of measurement: Measurement of reuse was not observed in any of the cases. Organizations lack any quantitative indication of the benefits or drawbacks of cloning.

“In quarterly reviews we should report how many hours in average we saved by reusing. [But there are] no measures. Usually there are reports on few hundreds of hours.”

“There is no structured method [to evaluate reuse]. [We do] design review in which we validate that reuse was done.”

“We don’t really measure, but there are some places that know the level of reuse, e.g., that we use an asset four times. In general, no one measures.”

2.3 Discussion

In this section, we discuss the findings and the observations we made during data collection and analysis. We derive recommendations and identify directions for further research. Our discourse provides three different angles:

1. The individual (human) perspective that looks at the attitudes and motivation for cloning.

2. The technical perspective that deals with the way cloning is implemented and ways to improve this practice.

3. The organizational perspective that examines processes, management style and organizational structures leading to the selection of cloning for realizing reuse in the context of software product lines.
2.3.1 The Individual Perspective

Among our subjects, cloning is perceived as a natural technique to support the development of similar products, mainly due to its low entrance barrier. Also, in most cases, practitioners can perform cloning in an ad-hoc manner without the need to adhere to formal procedures. Most practitioners are happy with cloning and can easily explain their motivation: it accelerates development as “we use what we already have”; it saves time and, therefore, it saves money.

However, there is ambivalence in the way cloning is perceived. Delving into the details and raising questions on how cloning is performed, as well as on how clones are evolved and maintained, reveals feelings such as frustration and helplessness. Some interviewees were calling to avoid cloning or to “eliminate it”. Information about cloning is usually kept by each individual and no special tools are used to store, maintain and share it. Maintaining different products that include independently-evolved cloned artifacts is time-consuming and gives the interviewees the feeling that they are “not professional enough”.

In addition, it seems that practitioners lack the awareness and knowledge about different forms of reuse and, specifically, about methods that avoid cloning. Promoting education on this topic could contribute to improving practices of individuals and organizations. This includes, among others, studying and leveraging experiences of other organizations. For example, our interviewees indicated that reusing already-tested code is one of the main benefits of cloning. Yet, several studies show that testing in the original environment is not necessarily sufficient for the target conditions of the cloned code. A prominent example is the Ariane disaster \[115\] which occurred due to reuse of code that was tested and verified on previous missions but which did not fit the new advanced system.

2.3.2 The Technical Perspective

As mentioned above, some of our interviewees suggested to “eliminate” cloning. Yet, it is unclear how that process would be performed and what are its implication on product schedules and staffing constraints of an organization. Understanding the transition process and its individual steps can facilitate the transition to a different, better managed reuse mechanism. The ability to mine and refactor existing artifacts can further promote such transition as organizations that made a significant investment in building their systems are not ready to abandon them for “starting from scratch”. As these systems are usually very large and complex, automation becomes a necessity.

Another way to improve the organizations’ reuse practices is to manage the cloning knowledge. Since cloning has clear benefits, i.e., simplicity and availability, we suggest to study ways for resolving its shortcomings and to define conditions in which cloning can be accepted as a reasonable practice. Our interviewees suggested to strengthen the cloning practice by integrating it into the architectural decision making process. This means that cloning activities should be performed after an approval, should be consistent with architectural decisions and should be validated during architecture and design reviews. The interviewees also suggested to document cloning tasks, meaning both to identify a task as a cloning one and to document its lifecycle. Such documentation could provide the ability to trace and share the cloning knowledge and to examine the cloning patterns. Numerous existing tools allow detecting existing clones, propose techniques for their elimination, or promote actions to keep the clones synchronized (see Section 2.5 for more details). Yet, it is not clear how to do that in an effective manner in a software product line context. Future studies are needed to identify precise requirements for clone management tools in SPL and to determine suitable existing tools.

Finally, there is lack of quantitative data on the saving from and costs of cloning. It is also unclear to what size of a product line cloning is still beneficial and when it stops being effective. There is a need for additional studies and better guidance on these topics. One needs to understand when it pays off to eliminate clones and when it
may be better to keep them. Unfortunately, the SPLE community has little or no guidance on that. We suggest to compare between product line environments that use cloning and those that avoid cloning, thus improving the understanding of the alternatives. Any approach that aspires to be better than cloning has to suggest a way to address the great perceived benefits of cloning, which is simplicity and availability. Many SPLE approaches do not get sufficient traction because they fail to convince practitioners that they would yield better results. Hard data would play a role in providing correct guidance and improving adoption of such alternative techniques.

2.3.3 The Organizational Perspective

Table 2.2 shows that code is cloned fairly often (statement #6). Still, the level of reuse across a product line is not always perceived as high (statement #2) or just not measured (statement #10). Although the work procedures are defined and participants comply (statements #17, 20, 22, 27), they do not include cloning activities (statement #19), which means that cloning is performed in an ad-hoc manner. Surprisingly, eight out of eleven participants indicated that they are aware of the product line reuse strategy in their organization (statement #1). We conclude that the topic of reuse is being discussed on a general level but without any actual definition or enactment of mechanisms to implement it in an efficient manner.

Cloning is part of an organization reuse strategy – whether it is explicitly defined or implemented in an implicit manner. Yet, we found that in the studied companies, processes regulating reuse are missing and organizational roles responsible for reuse are not well defined. Measurements regarding cloning are usually not taken and there is a lack of tools to support cloning (or to facilitate its elimination). Since organizations regularly define work procedures and other governance mechanisms to steer the development process, we suggest to increase the awareness of cloning as a reuse method and to integrate it into current processes. This includes shaping the role schema, refining work procedures, adding basic measures for assessing its implementation, and educating employees on the risks and trade-offs of cloning.

There is a non-negligible investment in setting up a managed product line architecture. Organizations might face difficulties in raising and justifying funds to pay these costs. Moreover, current organizational structures require significant changes to shift from per-product to a product line development. However, there is a trade-off between short-term savings obtained through cloning and the long-term maintenance problems caused by it. This trade-off should be assessed and taken into account when deciding on a reuse strategy. In organizations with loose governance, people might be tempted to “locally optimize” their work, thus sacrificing the “globally-optimal” result.

2.4 Threats to Validity

External validity. The main threat to external validity of our work is the limited number of subjects stemming from six product lines. We attempted to mitigate this threat by approaching development organizations from different industrial sectors and by interviewing practitioners holding a spectrum of organizational roles. Furthermore, we acknowledge this threat by carefully avoiding generalizing our results. Instead, we invite other researchers to confirm or refute our findings by studying further companies.

Our exclusive data sources are interviews and questionnaire responses. In particular, we did not perform any artifact study of the actual product line projects, e.g., to analyze sizes of clones or the extent of replication. Such an analysis could complement our study and aim at cross-checking results. However, artifact studies are, in general, difficult to perform for commercial closed-source projects, which are the main interest of our work.
We limited our subjects to those who apply code cloning for product line development. Thus, estimating the frequency of this technique in practice is out of scope of our work. Based on our own experience and literature reviews, we speculate that code cloning is a common approach to product line realization; however, a broader follow-up study would be necessary to empirically confirm (or refute) this hypothesis.

**Internal validity.** As for internal validity, we might have misphrased some interview questions in a way that affects participants’ answers, especially since some interviewees were unaware of product-line-specific terminology. We mitigated this threat by doing a pre-test of our study and refining our interview guide when questions raised confusion.

### 2.5 Related Work

*Software cloning* has seen active research in the last decade, both in the context of single systems development and software product lines. We now discuss such related work.

**Clone Management Techniques.** Clone management techniques have been extensively studied; in particular, *clone detection* – identifying cloned code artifacts [15], *clone synchronization* – tracking clones during evolution and propagating changes [38, 40], *clone merging* – integrating multiple changes that happened simultaneously to the clones on both sides [113], *clone correction* – eliminating clones through refactoring [12, 130], and *clone prevention* – assisting developers while writing code [97]. Our work differs from those, as we do not suggest any technique, but rather empirically investigate cloning practices in industry. Further, we study cloning in the context of software product lines, not single system development.

**Studies on Software Cloning.** Although clones were initially considered undesirable [73], multiple authors later argued that this assumption is not necessarily true and reported corresponding empirical data. Kapser and Godfrey [81] describe several observed patterns of cloning and provide evidence that cloning is used as a principled engineering technique. Aversano et al. [9] present empirical data on how clones are maintained. The results seem to indicate that the majority of clones is maintained consistently, and if an inconsistency is introduced, then often intentionally with a reason. This finding was confirmed by Göde and Koschke [60] who study the frequency of changes to clones, concluding that only 12% of clones were ever changed and only 15% of all changes were unintentionally inconsistent. Kim et al. [88] further indicate that eliminating clones not necessarily improves the development. By analyzing clone genealogies from codebase histories, they conclude that for short-lived clones, the removal effort is often too high to pay off, whereas long-lived clones are usually too hard to refactor. Ernst et al. [50] study forking of open source projects, which can be seen as a special form of inter-organizational cloning by putting whole codebases under another leadership. Forking allows addressing new requirements, but at the risk of fragmenting developer communities. Although our work focuses on intra-organizational cloning in the context of SPLE, such studies complement our finding that cloning is considered a beneficial and sometimes necessary technique.

Cordy [32] presents an experience report showing that developers often enjoy the freedom of making arbitrary changes to their clones and believe that cloning reduces coupling, testing costs, and maintenance risks. The report focuses on a few projects from the financial domain in which the largest part of the development costs (70%) were spent on the testing phase. Producing shared code that eliminates clones in these projects would thus have raised costs significantly as changing such shared code requires re-testing of all modules using it. Cordy’s observations are consistent with our study. However, our work has a broader nature and also specifically targets the SPL perspective.
**Cloning in the Context of SPL.** Several research works [131, 52, 108, 109] indicate the existence of industrial product lines that are realized by cloning instead of using variability management techniques. Faust and Verhoef [52] further describe “Software Mitosis” – the uncontrolled adoption of systems in a global organization when successful systems are duplicated and modified by local sub-organizations.

Staples et al. [160] indicates that Software Configuration Management (SCM) systems, such as CVS, RCS or SVN, are used to realize product lines by exploiting their branching and merging capabilities. The authors report experiences of creating a product line using an SCM system and challenge some tenets from the literature, such as the need for a complete upfront scoping process and a variability-enabled architecture. Van Gurp et al. [171] confirm this perspective and argue that SCM can be sufficient for SPL. Thao et al. [164] proposes a specific product line tool based on SCM which has built-in product derivation support.

Our study was inspired by all these individual reports and we also observed the use of SCM tools. In fact, four of our participants stated that they indeed created a branch for each new product. However, the main focus of our study was a qualitative investigation of the cloning practice. Studying whether SCM systems improve the cloning practice is out of scope of our work, but is a subject for a valuable study in the future.

**Maturity of SPL.** Our results can be further compared to existing work on organizational aspects and maturity levels of SPL, such as Riva and Del Rosso [131] and Bosch [19]. These authors classify the clone-and-own approach as the lowest level of maturity in an organization. Our study challenges this simplification, at least from the perception of our participants: in several cases cloning appeared to be a legitimate practice, even in organizations with very mature development processes.

**Technical Debt.** In the SPL context, cloning can be seen as a form of technical debt [34] accumulated by a company over the years. In fact, several observations made by our study concur with those on technical debt made by Kruchten et al. [94]. Specifically, as with technical debt, we also observed that scheduling pressure, lack of education and poor processes constitute some of the reasons for companies to choose cloning. Moreover, cloning does not necessarily occur as a result of having made a wrong choice originally but rather as the result of evolving the product portfolio to the point where the original choice to clone might not be quite right in retrospect. Yet, unlike technical debt that can occur at either small or large scale and can merely be a result of carelessness and negligence, product cloning is a major, organization-wide and conscious choice. The implications of this choice are usually broad and are reflected not only in development practices but also in the organizational structure and management processes.

### 2.6 Summary

In this chapter, we reported on a study that we conducted in order to collect empirical data on current development practices in organizations that employ cloning to realize product lines. Our study involved eleven participants from six cloned product lines of three well-established large-scale organizations. We observed human, technical and organizational aspects of the cloning culture, discussed main characteristics of cloning and highlighted the perceived benefits and shortcomings of the approach.

We learned that the majority of the interviewed practitioners that are involved in developing cloned product lines perceive cloning as a favorable reuse approach which is “natural” to developers and has a low entrance barrier. Our participants stated that cloning is a rapidly available mechanism which allows them to start from an already specified and verified functionality while leaving the freedom and independence to make any necessary modifications to it. However, in the long run, cloning might result in difficulties to perform maintenance and
Our study highlighted the need to further investigate the trade-offs between savings obtained through cloning and the longer-term problems introduced by it. It also became clear that any approach that aspires to be better than cloning should show and quantify its benefits, as practitioners’ desire to abandon their current cloning practices is not obvious. Moreover, the overhead related to the transition from cloning to any structured SPLE approach should be reduced as much as possible – the main focus of industrial organizations is on products they deliver; they have little time or resources to spend on the task of managing reuse.

We also observed a lack of organizational roles responsible for and promoting software reuse and, as a result, ad-hoc reuse practices driven by practitioners based on their personal acquaintance with other related projects. This emphasizes the need for establishing organizational structures that support SPLE. Moreover, industrial practitioners should be educated on different reuse approaches, focusing on expected benefits from adopting those both on the personal and on the organizational level.

We use the result of this study to drive our research agenda and shape the focus of this thesis, as we believe that without understanding the nature of the existing cloning practices attempts to promote better methodologies and tools remain difficult and unfocused. We consider mainly the technical perspective, leaving other important aspects such as human and organization perspective out of the scope of this thesis. In Chapters 4 and 5, we perform an in-depth analysis of development activities in three organization that employ cloning to realize their product lines. We outline the needs of these organizations, break their activities into individual steps and discuss automation opportunities. We start, however, by fixing some terminology and defining notations that we use.
Chapter 3

Preliminaries: Software Product Line Engineering

The Software Product Line Engineering paradigm separates two processes: *domain* and *application engineering* (see Figure 3.1 taken from [119]). The first process, domain engineering, is responsible for establishing an *SPL architecture* and defining the commonalities and variabilities of a software product line. The second process, application engineering, is responsible for *deriving* individual *products* (a.k.a. *variants* or *applications*) from the SPL architecture established during domain engineering. It exploits the variability of the product line and ensures the correct binding of the variability according to the applications’ specific needs.

In this thesis, we adopt an automated *feature-driven annotative product line approach* [83, 82, 20, 35] which we describe below.
Chapter 3. Preliminaries: Software Product Line Engineering

Domain Engineering. The SPL architecture created during the domain engineering phase represents all possible products of a product lines (a.k.a. a 150% view). As schematically shown in Figure 3.2, artifacts of the architecture, i.e., requirements, tests, design and implementation elements, are annotated by features.

The exact type and granularity of artifacts depends on the development process that a specific organization employs. We thus loosely define artifacts as follows:

**Definition 1 (Artifact)** An artifact is a tangible by-product produced during the development of software. Some artifacts, such as class diagrams, requirements and design documents help describe the architecture and design of software. Other artifacts, such as code elements, executable models and tests are concerned with the implementation and validation of software.

A feature typically consists of a label and a short description that identifies its functionality [27]. For conciseness, either label or feature description can be dropped when clear from the context. While there is no universal agreement on what a feature is (and what it is not), we adopt the definition below taken from [80]:

**Definition 2 (Feature)** A feature is a distinctively identifiable functional abstraction that must be implemented, tested, delivered and maintained. A feature consists of a label and a short description that identifies its behavior.

Annotative product line approaches assume that a feature annotates all artifacts that contribute to its implementation. A feature can annotate one or more artifacts; an artifact can be annotated by one or more features. We also say that artifacts are traced to the feature they implement. A feature and its implementation constitute a functionality of an individual product or an entire product line.

**Definition 3** A functionality of a product or a product line is a feature and its corresponding feature implementation – a set of artifacts annotated by that feature.

Intuitively, an annotative SPL representation is similar to code parameterized with preprocessor directives. It thus does not require a paradigm shift in the way software is being developed, especially in the embedded domain, and is easier to adopt in practice [20]. It is up to a particular SPL architecture to decide on the granularity of annotations, e.g., whether features annotate complete functions of source code or individual statements. Likewise,
in models, features can annotate coarse- or fine-grained artifacts, e.g., complete statecharts or individual states and transitions.

Figure 3.3 shows an example of a “toy” vending machine SPL architecture taken from [28]. Artifacts of this architecture are states and transitions in a transition system. These artifacts are annotated by seven vending machine features: cancel, soda, tea, pay, free, compartment and accessible. In this thesis, annotations are represented by appending the feature label in square brackets to the elements annotated by that feature. For example, transitions #1 and #2 in Figure 3.3 are annotated by the feature pay.

The vending machine SPL architecture “encodes” numerous variants. The functionality of each variant is defined by the subset of features that comprise it. Given a feature subset, the corresponding variant is derived from the SPL architecture during the application engineering phase, as explained later in this chapter. For example, a vending machine variant in Figure 3.4(a) implements functionality defined by the pay, soda and compartment features. It accepts payment, returns change, allows to select soda and serves it. It then opens a compartment...
allowing the user to take the drink, and, when taken, closes the compartment. Another variant of the vending machine that implements functionality defined by the pay, soda, tea and compartment features is shown in Figure 3.4(b). It allows the user to choose either soda or tea and then serves the chosen drink. Yet another variant, in Figure 3.4(c), implements functionality defined by the pay, cancel, soda and compartment features. It allows the user to cancel the purchase before selecting the drink and returns the paid amount. The one in Figure 3.4(d) offers free drinks and is fully accessible, i.e., does not open or close the beverage compartment. It implements functionality defined by features free, soda and accessible.

Obviously, not all functionality combinations are reasonable or desirable. For example, having both pay and free options together, as well as having both compartment and accessible options, do not make sense. Likewise, having a vending machine variant which serves no drinks – neither soda nor tea – is unreasonable. Constraints on valid functionality combinations are expressed in a feature model \[78\]. A feature model is a rooted tree whose nodes are feature labels. Relationships between parent and child features in the tree commonly include:

- **mandatory**: a child feature must be selected when its parent is selected;
- **optional**: a child feature might be selected when its parent is selected;
- **or-group**: at least one of the sub-features must be selected when the parent is selected;
- **xor-group** (a.k.a. *alternatives*): exactly one of the sub-features must be selected when the parent is selected.

In addition to the main hierarchy, cross-tree constraints can be used to describe dependencies between arbitrary features. Commonly used cross-tree constraints are:

- **requires**: if one feature is selected the other needs to be selected as well;
- **excludes**: two features mutually exclude each other.

An example of a feature model for the vending machine product line is given in Figure 3.5. It has one optional feature: cancel; an or-group with features soda and tea; and two groups of alternative features: the first includes pay and free while the second – compartment and accessible.

Following \[157\], in this thesis, we represent a feature model as a set of features and a boolean formula that encodes the relationships between these features. For the example in Figure 3.5, the formula that represents the relationships between the features is \((\text{soda}\lor\text{tea})\land(\text{pay}\oplus\text{free})\land(\text{compartment}\oplus\text{accessible})\).

A feature model configuration is then a subset of features that “respect” the formula, i.e., for which the formula evaluates to true. Valid configurations include soda, pay and compartment (which corresponds to the product in Figure 3.4(a)), soda, pay, cancel and compartment (which corresponds to the product in Figure 3.4(c)) and more.
Definitions 4 and 5 formally describe the notion of a feature model, feature model configuration and an SPL architecture that we adopted in this thesis.

**Definition 4 (Feature Model and Configuration – simplified version of [157])** Given a universe of elements $\mathcal{F}$ that represent features, a feature model $\mathcal{FM} = (\mathcal{F}, \varphi)$ is a set of features $\mathcal{F} \subseteq 2^\mathcal{F}$ and a propositional formula $\varphi$ defined over the features from $\mathcal{F}$. A feature configuration $\hat{\mathcal{FM}}$ of $\mathcal{FM}$ is a set of selected features from $\mathcal{F}$ that respect $\varphi$ (i.e., $\varphi$ evaluates to true when each variable $f$ of $\varphi$ is substituted by true if $f \in \hat{\mathcal{FM}}$ and by false otherwise.)

**Definition 5 (SPL Architecture – simplified version of [18])** Given a universe of elements $\mathcal{A}$ that represent artifacts at a certain granularity level, an SPL architecture $\mathcal{PL} = (\mathcal{FM}, \mathcal{M}, \mathcal{R})$ is a triple, where $\mathcal{FM}$ is a feature model, $\mathcal{M} \in 2^\mathcal{A}$ is a domain model, and $\mathcal{R} \subseteq \mathcal{F} \times \mathcal{M}$ is a set of relationships that annotate elements of $\mathcal{M}$ by features of $\mathcal{F}$.

**Application Engineering.** As mentioned earlier, individual products are defined by a subset of features that correspond to the desired product functionality – a feature model configuration. Given a feature model configuration, a product is derived from the SPL architecture by removing domain model artifacts annotated by features that are excluded from the configuration.

Below, we formally describe the product derivation process. We denote by $\Delta$ the mapping between an artifact of the domain model and the corresponding artifact of the derived product. We denote by $\Delta^{-1}$ the inverse mapping. For example, let $a$ and \( \hat{a} \) refer to the transition #12 and transition #1 in Figures 3.3 and 3.4(d), respectively. Then, under the configuration that includes features soda, free, and accessible, $\Delta(a) = \hat{a}$ and $\Delta^{-1}(\hat{a}) = a$.

**Definition 6 (Product Derivation – adapted from [18])** Let $\mathcal{PL} = (\mathcal{FM}, \mathcal{M}, \mathcal{R})$ be an SPL architecture and let $\hat{\mathcal{FM}}$ be one of its feature model configurations. A set of elements $\hat{\mathcal{M}}$ is derived from the architecture $\mathcal{PL}$ under the configuration $\hat{\mathcal{FM}}$, denoted by $\hat{\mathcal{M}} = \Delta(\mathcal{PL}, \hat{\mathcal{FM}})$, if and only if the following properties hold:

(a) An element belongs to the derived product if and only if this element is annotated by a feature of the feature configuration $\hat{\mathcal{FM}}$ (under which the derivation was performed): $\forall m \in \mathcal{M}, \Delta(m) \in \hat{\mathcal{M}} \iff \exists f \in \hat{\mathcal{FM}} \cdot (f, m) \in \mathcal{R}$.

(b) Only one element can be derived from a given domain model element:

$\forall m \in \mathcal{M}, \exists ! \hat{m} \in \hat{\mathcal{M}} \cdot \hat{m} = \Delta(m)$.

(c) Only derived elements are present in the derived model: $\forall \hat{m} \in \hat{\mathcal{M}}, \exists m \in \mathcal{M} \cdot \hat{m} = \Delta(m)$.

Lemma 1 below shows that a feature model configuration uniquely identifies the derived product model.

**Lemma 1 (Uniqueness)** Let $\mathcal{PL} = (\mathcal{FM}, \mathcal{M}, \mathcal{R})$ be an SPL architecture, $\hat{\mathcal{FM}}$ be a feature configuration and $\hat{\mathcal{M}} = \Delta(\mathcal{PL}, \hat{\mathcal{FM}})$. Then, for each $\hat{\mathcal{M}}' = \Delta(\mathcal{PL}, \hat{\mathcal{FM}}')$, $\hat{\mathcal{M}}' = \hat{\mathcal{M}}$.

**Proof 1** Assume to the contrary that $\hat{\mathcal{M}}' \neq \hat{\mathcal{M}}$ and assume without loss of generality that $\exists \hat{m} \in \hat{\mathcal{M}}$ such that $\hat{m} \notin \hat{\mathcal{M}}'$. By Definition 6(c), $\hat{m} \in \hat{\mathcal{M}}$ implies that $\exists m \in \mathcal{M} \cdot \hat{m} = \Delta(m)$. By Definition 6(a), this means that $\exists f \in \hat{\mathcal{FM}} \cdot (f, m) \in \mathcal{R}$. Since $\hat{\mathcal{M}}'$ was derived from $\mathcal{PL}$ under the same configuration $\hat{\mathcal{FM}}$, $\Delta(m) \in \hat{\mathcal{M}}'$ by Definition 6(a), which implies that $\exists \hat{m}' \in \hat{\mathcal{M}}' \cdot \hat{m}' = \Delta(m)$ by Definition 6(b). Since $\hat{m} = \Delta(m) = \hat{m}'$, we conclude that $\hat{m} \in \hat{\mathcal{M}}'$ which creates a contradiction.

We rely on this quality of annotative product line representations and on notations established in this chapter in the remainder of the thesis.
Chapter 4

Cloned SPL Management Framework

In Chapter 2, we showed that despite shortcomings, cloning is often perceived as a favorable and natural reuse approach because of its benefits such as simplicity, availability and developer independence (see Table 4.1 for the summary of perceived benefits of cloning). However, when the number of cloned variants is large and the differences between their implementations are significant, it becomes difficult to keep track of changes made to each of the variants, share features between them, reconcile changes and establish new products (again, see Table 4.1 for the summary of perceived shortcomings of cloning). The prevalent view in the SPLE community suggests that cloned product variants should be unified into a single-copy SPL representation ([14, 53, 79, 90, 68, 87, 75, 74] – a direction that we refer to as merge-refactoring. Yet, despite several isolated merge-refactoring solutions designed with a specific context or application domain in mind, no systematic effort has been made to understand and organize support required by organizations that rely on cloning to realize their product lines.

Our work aims at addressing this gap. Inspired by literature reports ([68, 75, 74] and our own empirical analysis of development activities in three industrial organization (see Chapter 5), we outline key observations inducing requirements for efficient management of clones. We show that focusing on supporting merge-refactorings only is insufficient for the needs of most companies: it is necessary to complement merge-refactoring with an additional direction that allows companies to maintain and evolve their existing clones in a more efficient manner – a direction that we refer to as a clone-based SPLE approach.

We introduce a set of seven conceptual operators that can be composed to realize both the merge-refactoring and clone-based SPLE approaches. These operators constitute our cloned SPL management framework contributed in this work. We map operators to their possible implementations and identify remaining gaps. We also present our vision of building a body of knowledge around existing implementations of the operators.

Objectives of the operator-based framework. The operators presented in this work emerged from our own experience studying organizations that use cloning to realize their product lines and our analysis of reports

Table 4.1: Main benefits and shortcomings of cloning (see Chapter 2 for details).

<table>
<thead>
<tr>
<th>Benefits</th>
<th>Shortcomings</th>
</tr>
</thead>
<tbody>
<tr>
<td>– No upfront investment</td>
<td>Difficult to:</td>
</tr>
<tr>
<td>– Rapidly available</td>
<td>– Reconcile changes</td>
</tr>
<tr>
<td>– Reuse of verified code</td>
<td>– Share features</td>
</tr>
<tr>
<td>– Developer independence</td>
<td>– Establish new variants</td>
</tr>
</tbody>
</table>
published in the literature. As shown in Chapter 5, these operators are adequate for the three cases that we analyzed in detail. Yet, since our access to industrial organizations is naturally limited, we do not claim that the set of operators is necessarily complete or is captured at the optimal level of granularity: some operators might be missing and some of those we considered atomic might need to be further split up. Moreover, we do not prescribe any particular type or granularity of elements handled by the framework, and our operators are deliberately generic. Providing more formal reasoning about the operators and their compositions, e.g., in terms of an abstract algebra, could be beneficial.

Instead, the main objective of the framework is to demonstrate the usefulness of an operator-based view for capturing the support that companies require. Our goal is to allow describing the implementations specific to a particular organization or domain using a common terminology, helping both researchers and practitioners in understanding the context of the existing work, reusing it when appropriate and estimating the investment they need to make in order to implement the missing functionality. Systematizing and organizing support that companies require also allows tool builders to focus on the functionality that is missing. Indeed, we discovered that while some of the operators have received a lot of attention in the literature, others, despite being equally important, have barely been studied.

Contributions. This chapter makes the following contributions:

1. We outline key observations inducing requirements for efficient management of cloned variants in organizations that rely on cloning to realize their product lines.

2. We present the seven conceptual operators that constitute our cloned SPL management framework and demonstrate their composition for addressing realistic development activities.

3. We discuss possible implementations of the operators, including synergies with relevant solutions developed outside the SPLE context, identify remaining gaps and suggest approaches towards addressing these gaps.

The chapter is based on our publications in [140, 145]. We proceed as follows. Section 4.1 outlines the key observations we collected in this work. Section 4.2 presents the operators while Section 4.3 demonstrates their composability. We discuss existing and possible approaches for implementing the operators, as well our vision of building the body of knowledge around these implementations, in Section 4.4. Section 4.5 discusses related work. Section 4.6 summarizes and concludes the chapter.

4.1 Key Observations

As observed by multiple authors [68, 75, 74] and based on our own experience described in Chapter 5, performing merge-refactoring for large and complex real-life products is an incremental process which often takes several years. During this time, immediate customer needs still have to be addressed by cloning. Moreover, in many cases, the created SPL architecture only targets future products, while existing ones are still maintained as distinct clones.

Observation 1: Maintaining cloned product variants is a necessity even for companies that step away from cloning practices and invest in merge-refactoring their product lines.

We thus conclude that there are two main directions for improving reuse practices and reducing costs in such companies, which might coexist and complement each other even within a single organization for several years. The directions are schematically shown in Figure 4.1. The first one, traditionally promoted by the SPLE community, relies on unifying cloned products into single-copy SPLE representations (i.e., merge-refactoring). This translates
into merging variants while identifying features of each product and relationships between these features, as well as unifying commonalities at the artifact level and tracing variabilities to their corresponding features.

Merge-refactoring has to be complemented with an additional direction that allows companies to manage their existing cloned variants (i.e., clone-based SPL). Variant management includes the following activities: (1) sharing features between variants, (2) reconciling changes, (2) retiring features and, in some cases, even (4) establishing new variants via cloning. To better support such activities, it is required to build an efficient management infrastructure on top of existing products, mitigating the shortcomings of cloning while leveraging its benefits (see Table 4.1). This, again, translates into detecting and capturing essential product line constructs such as features, dependencies between features, as well as traceability information between the features and their implementation artifacts.

It is not surprising that both directions require detecting and maintaining a similar kind of information: a set of features, their dependencies and traceability to implementation-level artifacts. Such knowledge, in fact, constitutes the essence of SPL. When collected incrementally, on a need-to-have basis, it streamlines management of existing clones and provides a lightweight SPL approach that is based on clones. The adoption of this approach is incremental and gradual, and does not require a significant upfront investment. Moreover, as schematically shown in Figure 4.1, building the clone-based SPL management infrastructure on top of existing clones eases a future merge-refactoring effort, as the collected information is directly relevant to support this effort [146].

**Observation 2:** Incrementally collecting knowledge about existing cloned variants and using it to improve their maintenance introduces a lightweight SPL approach that is based on clones. It also facilitates future merge-refactoring efforts.
Table 4.2: Operators for Managing Cloned Variants.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. findFeatures</td>
<td>variant</td>
<td>set of features</td>
</tr>
<tr>
<td>2. findFeatureImplementation</td>
<td>$f \times \text{variant} \times \text{property}$</td>
<td>feature impl.</td>
</tr>
<tr>
<td>3. dependsOn?</td>
<td>$\langle f_1, \text{variant}\rangle \times \langle f_2, \text{variant}\rangle \times \text{property}$</td>
<td>set of witnesses</td>
</tr>
<tr>
<td>4. same?</td>
<td>$\langle f_1, \text{variant}_1\rangle \times \langle f_2, \text{variant}_2\rangle \times \text{property}$</td>
<td>set of witnesses</td>
</tr>
<tr>
<td>5. interact?</td>
<td>set of $\langle f_i, \text{variant}_i\rangle \times \text{property}$</td>
<td>set of witnesses</td>
</tr>
<tr>
<td>6. merge</td>
<td>system$_1 \times \ldots \times$ system$_n$ $\times$ matches $\times$ resolution</td>
<td>system</td>
</tr>
<tr>
<td>7. reorganize</td>
<td>system $\times$ property</td>
<td>system$'$</td>
</tr>
</tbody>
</table>

4.2 Operators

In this section, we present the conceptual operators of the cloned SPL management framework, which capture the technical support required for managing cloned variants – both for the merge-refactoring and the clone-based SPL directions. The operators are listed in Table 4.2. We describe them below and exemplify using the vending machine example in Figure 3.4.

In the rest of the chapter, we assume that all operators have global access to all product variants, even those that are not passed explicitly as a parameter to the operator. In addition, we use the term feature-oriented system (a.k.a. system) to refer to a feature model and a set of artifacts annotated by feature of that model. A feature-oriented system can represent an SPL architecture as well as an individual product variant. In the latter case, the feature model only includes those features that where selected when deriving the variant from an SPL architecture.

1. findFeatures returns a set of features, i.e., $\langle$feature label, description$\rangle$ pairs, realized by the given product variant. For the vending machine in Figure 3.4(b), the features include $\langle$soda, sells soda$\rangle$, $\langle$tea, sells tea$\rangle$ and $\langle$pay, allows to pay for the drink being purchased$\rangle$. For brevity, we omit either the name or the description of a given feature, if it is clear from the context.

2. findFeatureImplementation, commonly known as feature location, returns a feature implementation of the given feature $f$ – the set of artifacts that realize the input feature. We say that these artifacts are traced to the feature. The exact form of the detected feature implementation depends on the goal of feature location: e.g., “detect the artifacts that contribute only to the feature of interest” or “detect all artifacts required for the feature to be executable (including the main method of a program)”. We declaratively represent this goal using the input property that specifies inclusion and exclusion conditions for the feature location process. For example, transitions 1 and 2 of the vending machine in Figure 3.4(b) realize the pay feature w.r.t. the property which disregards transitions contributing to other features, such as soda and tea.

3. dependsOn? determines whether the functionality described by feature $f_1$ requires the functionality described by feature $f_2$ from the same product variant in order to operate. The input property captures the nature of the dependsOn dependency. Such a property can express simple dependencies such as “$f_1$ requires $f_2$ in order to compile”, or more complex behavior dependencies. The latter could be given as formal specifications or as tests.

For our example in Figure 3.4(b), the soda functionality requires the pay functionality w.r.t. the property
“soda is served only after a payment is received”. The operator returns a set of witnesses, each demonstrating the dependsOn relationship between the artifacts of $f_1$ and $f_2$ (or none if the functionalities are independent). In the example above, a witness is the flow between the artifacts implementing the pay and soda features: transitions 1 and 2, implementing the first one, precede transitions 3 and 4, implementing the second.

4. same? determines whether the functionality described by feature $f_1$ of variant $v_1$ is consistent with the functionality described by feature $f_2$ of variant $v_2$, i.e., whether there are no disagreements in both the features and the implementations of the two seemingly equivalent functionalities. For the products in Figure 3.4, the compartment feature, allowing one to take the ordered drink, is implemented similarly in Figures 3.4(a), 3.4(b) and 3.4(c), by transitions 5-7. These three product variants are in agreement on the implementation of the feature. Although feature accessible in Figure 3.4(d) implements a similar functionality, this feature is implemented only by transition 4 since the corresponding product does not need to open and close the beverage compartment. Thus, this feature implementation “disagrees” with the implementation of the compartment feature in the first three variants.

Like dependsOn?, the same? operator uses a property that specifies disagreements of interest and returns a set of witnesses exemplifying the disagreements (or none if the features agree). A simple form of disagreement is when features have different implementations, as in the above example. In that case, a witness could include artifacts that distinguish between the corresponding feature implementations. Disagreements can also be semantic, e.g., when checking for behavioral properties rather than the syntax of the implementing artifacts.

5. interact? determines whether combining functionalities described by a set of features would alter the behavior of one or more of those functionalities. The input property specifies the form of interactions to be checked and the output set of witnesses exemplifies them. For example, a composition of functionalities described by features pay and free from the transition systems in Figures 3.4(a) and 3.4(d) might result in a transition system where the transition pay follows free: one has to pay after requesting a free drink, clearly violating the main behavioral property of the free feature.

6. merge combines functionalities of the $n$ input systems, producing a single system as a result. The matches parameter specifies artifacts that are considered similar and should be unified in the combined representation. In addition, the resolution parameter declaratively specifies how to resolve disagreements and interactions between the input functionalities, e.g., by overriding one feature implementation with another, integrating the implementations together (thus producing a “merged” implementation), or keeping both as separate functionalities (with distinct feature declarations). For example, when composing the transition systems in Figures 3.4(a) and 3.4(d), one might choose to override the behavior of the accessible feature in Figure 3.4(d) with the behavior of the compartment feature in Figure 3.4(a) or keep both behaviors as alternatives. merge can be used for combining individual features (systems with a single feature each), adding a feature to an existing product (systems with a single feature combined with a system representing a well-formed product), or combining distinct products (systems each representing a well-formed product).

7. reorganize refactors a system to improve its structure, without modifying the behavior. It can be applied to either an individual product variant or a complete product line. The nature of the desired refactoring is declaratively specified by the input property. The operator returns a refined version of the system, after the reorganization has been performed. For example, the transition system in Figure 3.4(d) can be simplified by excluding the first transition w.r.t. the property that requires to eliminate elements not affecting the execution of the system.
Input: variant\textsubscript{A} and variant\textsubscript{B}, with variant\textsubscript{B} containing a feature of interest \( F \)
Output: variant\textsubscript{A} with the additional feature \( F \) from variant\textsubscript{B}
1. \( \text{functionality}_B = \langle F, \text{findFeatureImplementation}(F, \text{variant}_B, \text{property}_1) \rangle \)
2. \( \text{system}_A = \emptyset; \text{resolution} = \emptyset \)
3. \( \text{setOfFeatures}_A = \text{findFeatures}(\text{variant}_A) \)
4. \( \text{for each } F_A \text{ in setOfFeatures}_A \)  
5. \( \text{system}_A = \text{system}_A \cup \langle F_A, \text{findFeatureImplementation}(F_A, \text{variant}_A, \text{property}_2) \rangle \)
6. \( \text{end for} \)
7. \( S = \text{set of } \langle F_A, \text{variant}_A \rangle \text{ pairs } \cup \{ (F, \text{variant}_B) \} \)
8. \( \text{witnesses} = \text{interact}\? (S, \text{property}_3) \)
9. \( \text{if} \) (witnesses \( \neq \) null) provide resolution using witnesses
10. \( \text{system}_A' = \text{merge}(\text{system}_A, \text{functionality}_B, \text{matches}, \text{resolution}) \)
11. \( \text{return} \text{artifacts of } \text{system}_A' \)

**Figure 4.2: Algorithm for Sharing Features.**

### 4.3 Composition of the Operators

In this section, we describe two conceptual examples demonstrating the composability of the operators. More elaborated real-life examples are presented in Chapter 5.

#### 4.3.1 Sharing Features Between Variants

In the clone-based SPL approach, developers occasionally need to identify and share features between cloned variants. Figure 4.2 sketches the sequence of operators supporting this activity. For example, suppose the developers of the Soda and Tea variant in Figure 3.4(b) decide to adopt the cancel functionality implemented by Soda with Cancel in Figure 3.4(c). They retrieve the artifacts implementing cancel using \( \text{findFeatureImplementation} \) (line 1) with \( \text{property}_1 \) set to find the minimal set of artifacts directly contributing to this feature, as those will be transferred to the target variant. Transitions 8 and 9 in Figure 3.4(c) are returned as the result.

Next, the integration target \( \text{system}_A \) is built from the artifacts of \( \text{variant}_A \): first, all features of the Soda and Tea variant are detected using \( \text{findFeatures} \) (line 3), and then \( \text{findFeatureImplementation} \) is applied to these features (lines 4-6). Detected features and their implementations comprise a system that represents \( \text{variant}_A \).

To avoid undesired integration side-effects, \( \text{interact}\? \) is applied on the set of all features of \( \text{system}_A \) and the new cancel feature, with respect to \( \text{property}_3 \) (lines 7 and 8). This property specifies behavioral characteristics of all input features and looks for violations of these characteristics as the result of feature composition. Our example does not have violations, and no resolution is needed (line 9).

Finally, \( \text{merge} \) combines the system with the cancel functionality from the Soda and Tea variant (line 10). In our example, the \( \text{matches} \) parameter considers artifacts to be similar if they have identical names. As the result, the option to cancel the purchase is copied from the original Soda with Cancel variant to the corresponding location in the artifacts of \( \text{system}_A' \), producing the desired model in Figure 4.3.

#### 4.3.2 Merge-Refactoring Variants into an SPLE Representation

A schematic implementation of merge-refactoring is sketched in Figure 4.4: first, \( \text{system}_i \) is built for each \( \text{variant}_i \) by detecting its features and the corresponding feature implementations (lines 1-6). Unlike Figure 4.2, here \( \text{dependsOn}\? \) is applied to each pair of features from the same variant, capturing the require relationships between them (lines 7-10) – this time, such relationships are an important part of the produced representation. Next,
each pair of features from different variants is checked for potential disagreements (lines 12-15), and all possible combinations of features are checked for potential interactions (lines 16-19). Since variants derived from the generated SPL representation can contain various sets of features, we check all combinations. However, specific detection techniques can provide optimizations to avoid a large number of checks.

If disagreements or interactions are found, developers need to provide the desired resolution (lines 14 and 18). The simplest resolution strategy is to mutually exclude the disagreeing or interacting features, e.g., the accessible feature of the Free Soda variant and the compartment feature of the remaining variants in Figure 3.4: these two features do not agree on the functionality of serving the ordered drink. Likewise, feature free implemented in the Free Soda variant is mutually exclusive with feature pay implemented in the remaining variants as these two functionalities interact. Figure 3.3 shows the domain model produced by merge (line 20) when combining the variants in Figure 3.4 under this resolution strategy. Like the previous case, matches combines artifacts with the same name. More complex matching and resolution strategies allowing to override or merge features from different variants are also possible.

The generated system has one mandatory feature: soda (as it appears in all variants); two optional features:
cancel and tea (as they appear in some, but not all variants); and two groups of alternative features: compartment and accessible, and pay and free. The goal of reorganize (line 21) is then to combine soda and tea features into an or-group – a request specified by the developer based on his familiarity with the vending machine functionality. No changes to the domain model and the traceability between the feature and domain model artifacts are required in this case, producing the result that corresponds to the SPL architecture in Figures 3.3 and 3.5.

4.4 Implementing the Operators

In this section, we outline existing approaches for implementing the operators. Performing a systematic literature survey for each of the operators is out of the scope of this thesis. Instead, we aim at mapping the space of existing work and identifying obvious gaps. Indeed, we discovered that while some of the operators received a significant amount of attention in the literature, other, equivalently important, operators are poorly studied. We discuss these below and also propose possible directions for addressing the gaps identified in this work.

findFeatureImplementation (a.k.a. feature location) and interact? (a.k.a. feature interaction) are by far the most studied operators. Over 20 different feature location techniques for source code have been developed (see Appendix A for an extensive survey). Yet, it is often unclear what the exact properties of the located feature implementations are, how to compare techniques based on the features they detect, and how to extend these approaches to allow users to specify the desired properties of the location process. Also, feature location techniques for artifacts other than code, e.g., models, are poorly studied. We recently performed some initial steps towards addressing the latter problem [162] by proposing an approach for feature-based model splitting, but further investigation is still needed.

Feature interaction techniques have also received a lot of attention, especially in the telecommunications domain [181]. Most of the existing approaches, however, deal with pairwise feature interactions. They have to be extended to consider interactions between sets of features that are part of real-life products: such sets can introduce interactions that are not detectable in a pairwise manner. Also, the applicability of many techniques for analyzing feature interactions is limited because they are designed to work on special-purpose models rather than production artifacts.

Compare and merge techniques, for both code and models [54, 112, 161], as well as aspect weaving [86] and feature-oriented composition approaches [13], can be used to realize merge. Such techniques need to be extended to allow specifying the desired resolutions. Also, the techniques should be able to deal with unstructured product slices that correspond to feature implementations rather than complete, well-formed products or features declared in a specific manner. Approaches considering the “global” picture and devising strategies for combining n inputs simultaneously, rather than doing so in a pairwise manner, are also needed (see Chapter 8 for a possible solution to this problem).

Syntactic and semantic comparison techniques [70] can also be used to implement the operator same?. The implementation of the operator for textual documents can be based on analyzing lexical similarities, e.g., using the Levenshtein distance metric [99] or textual diff tools [104]. For model-level artifacts, various model comparison and matching techniques can be used [161]. Comparison of code-level artifacts can also rely on tools that attempt to detect semantic differences, e.g., [72], or on more sophisticated implementations based on code clone detection [16]. Yet, additional work is required to adapt these works for analyzing unstructured feature implementations and declaratively obtaining the desired properties of the analysis.

Code analysis techniques, e.g., program slicing [165], can be used to implement dependsOn?. However, to the best of our knowledge, there are no dedicated works focusing on detecting semantic dependencies between
Chen et al. [26] and Weston et al. [175] describe techniques for extracting a feature model from informal specifications, which can be seen as instances of `findFeatures`. However, the constructed feature model only includes features described in the documentation of the existing products. Davril et al. [36] recently proposed an automated approach for constructing feature models from publicly available product descriptions found in online product repositories and marketing web sites. There are also prior approaches designed for finding abstractions of a natural language text, e.g., [62]. These works could be extended with approaches that consider other product artifacts, such as models and code, and are applicable when no documentation is available. First steps in this direction can come from leveraging the techniques that decouple product code [67], and then concisely summarize each part individually [124].

Finally, the `reorganize` operator for a single individual product can rely on numerous code refactoring techniques [55, 110]. Borba et al. [18] formalize the problem of product line refinement and evolution. Yet, there is little work on specifying how a high-quality product representation, i.e., the target of `reorganize`, should look like (see Chapter 7 for a more detailed discussion).

### Towards Building a Body of Knowledge
The operator-based view promoted by this work provides a systematic approach for understanding the required, existing and missing functionality. However, the specific implementation of the operators can differ between organization and domains as no “generic” solution can work for all cases. Organizing existing implementations in an operator-based manner while explicating the exact assumptions made by these implementations, as sketched in Figure 4.5, can assist both researchers and practitioners interested in improving reuse practices in families of related products. Exposing organizations to existing implementation of the operators and helping understand their applicability can enable reuse of implementations across organizations with similar needs. Researchers and tool developers can use this body of knowledge to focus their effort on functionality that is missing. As such, “crowdsourcing” of existing support can eventually lead to an increased quality and a larger spectrum of available solutions. We believe that the implementations outlined in this section constitute the initial contents of the proposed body of knowledge.

### Figure 4.5: An Initial Sketch of the Knowledge-Based Library.

<table>
<thead>
<tr>
<th>Input</th>
<th>Assumptions</th>
<th>Existing Implementations</th>
<th>Usage Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Textual documents</td>
<td>Short textual documents, e.g., requirements.</td>
<td>Levenshtein distance</td>
<td>Case study #1</td>
</tr>
<tr>
<td>UML class diagrams</td>
<td>Compared models have common ancestors. Elements are compared based on their unique ids.</td>
<td>IBM Rational Software Architect</td>
<td>...</td>
</tr>
<tr>
<td>UML class diagrams</td>
<td>Elements with similar names are likely to be similar.</td>
<td>UMLDiff</td>
<td>...</td>
</tr>
</tbody>
</table>

```markdown
model- or code-level functionalities. Such works, when developed, should clearly specify the nature of the detected dependencies and even be parameterizable and allow retrieving dependencies of a desired type.

```
4.5 Related Work

Other authors also looked at systematic classifications of programming tasks: Chen and Rajlich [27] identified six fundamental program comprehension operators that trace feature label, description and implementation to each other. We incorporated some of them into our work, as these operators are also applicable in the context of cloned product variants. However, our main focus was on cases of multiple variants rather than single-copy systems and involved manipulations on the variants rather than only comprehension activities.

Borba et al. [18] suggested a theory of product line refinement. This is a special case of variant management – the more general problem that we consider here. She et al. [156] classified several software re-engineering scenarios involving feature model synthesis. These can be seen as detailed scenarios for our findFeatures and dependsOn? operators, while our work has a broader scope. Brunet et al. [23] identified model merging operators and specified their algebraic properties. Our work is not limited to models and considers a broader set of necessary maintenance activities, focusing specifically on cloned product variants.

Several works [53, 14, 87, 79, 90] capture guidelines and techniques for manually transforming legacy product line artifacts into SPLE representations. Some also introduce automatic approaches for reorganizing product variants into annotative representations [91, 109, 149]. Works on feature-oriented refactoring [103, 111] focus on identifying the code for a feature and factoring the code out into a single module or aspect aiming at decomposing a program into features. Our work differs from those as we do not propose a specific refactoring approach or technique but rather capture and classify common tasks required during such refactoring.

4.6 Summary

In this chapter, we took a top-down approach for identifying the support required by companies that rely on cloning to realize their product lines. We observed that an efficient management of clones comes from two directions: (1) the unification of the cloned variants into single-copy representations promoted by SPLE and (2) the construction of a management infrastructure on top of existing variants, which allows to mitigate the shortcomings of cloning while leveraging its benefits. These two directions coexist and complement each other even within a single organization during the transition from ad-hoc to managed reuse – a process that usually takes several years.

We identified a set of conceptual operators for managing cloned product variants that support both of these directions and demonstrated their compositability for realizing real-life scenarios. We also discussed possible strategies for implementing the operators, including synergies with solutions outside the SPLE domain, and identified remaining gaps. By focusing on the operators, we broke processes down into well-studied logical components, thus promoting the reuse of existing component implementations when dealing with cloned product variants. We believe that the generic operator-based view leads to more efficient development and maintenance practices. Specifically, it allows organizations to locate and reuse existing work, estimate the investment in implementing the missing functionality, and share experiences with each other using a common vocabulary.

While in this chapter we showed that the proposed set of operators is reasonable to realize typical real-life activities that we identified in industrial settings, the focus of the next one is to ground this observation in empirical evidence. We perform a detailed analysis of development activities in three companies that use cloning to realize their product lines and show how these activities can be expressed in terms of the operators.
Chapter 5

Cloned SPL Management Framework –
An Experience

The goal of this chapter is to ground the generic framework proposed in Chapter 4 in empirical evidence and exemplify its usefulness. In particular, we systematically analyze three industrial case studies of companies with cloned product lines and show that our set of operators is reasonable and sufficient for expressing their development activities. We consider both the case when cloned products are refactored into a single-copy software product line representation and the case when they are maintained as distinct clones.

Contributions. This chapter makes the following contributions:

1. We present a detailed analysis of development activities in three industrial organizations that employ (or have employed) cloning to realize their SPLs – both when managing existing clones and when merge-refactoring them into a single-copy SPL representation.

2. We discuss in detail the applicability and composability of the operators proposed in Chapter 4 for realizing these development activities, demonstrating the suitability of our framework for real-life cases.

3. We outline organization-specific operator automation opportunities.

The chapter is based on our publications in [144, 145]. We proceed as follows. Section 5.1 describes our research methodology and the three case studies that we analyzed. Sections 5.2 and 5.3 discuss the activities related to merge-refactoring and to efficient maintenance of existing clones, respectively, and map them to the instances of the operators. We discuss limitations and threats to the validity of our work in Section 5.4 while Section 5.5 summarizes and concludes the chapter.

5.1 Research Methodology and Case Studies

We start by describing the criteria that we used for selecting the case studies for our analysis. We then discuss the process that we followed for collecting and analyzing the data and give the relevant background about each of the studied companies.
Table 5.1: Analyzed Case Studies.

<table>
<thead>
<tr>
<th></th>
<th>Case Study #1</th>
<th>Case Study #2</th>
<th>Case Study #3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain</strong></td>
<td>Aerospace</td>
<td>Electric motor controllers</td>
<td>Aerospace and Defense</td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td>– V model (strictly waterfall)</td>
<td>– Iterative</td>
<td>– Iterative</td>
</tr>
<tr>
<td></td>
<td>– Model-centric, with full</td>
<td>– Code-centric</td>
<td>– Code-centric</td>
</tr>
<tr>
<td></td>
<td>requirements-to-code traceability</td>
<td></td>
<td>– Requirements managed by a</td>
</tr>
<tr>
<td></td>
<td>– DO-178B certified</td>
<td></td>
<td>requirements management tool but</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>no traceability to code is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>maintained</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>– Requirements-based testing</td>
</tr>
<tr>
<td><strong>Artifacts</strong></td>
<td>– System and software requirements</td>
<td>– C/C++ code</td>
<td>– Textual requirements</td>
</tr>
<tr>
<td></td>
<td>– Executable design models (code</td>
<td>– Tests</td>
<td>– C/C++ code</td>
</tr>
<tr>
<td></td>
<td>is generated)</td>
<td></td>
<td>– Tests</td>
</tr>
<tr>
<td></td>
<td>– Tests</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Transition</strong></td>
<td>– Over 50 product variants before</td>
<td>– At least four product variants in</td>
<td>– Five commercial product</td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td>transition</td>
<td>the new series</td>
<td>variants and a couple of prototypes</td>
</tr>
<tr>
<td></td>
<td>– Complex derivation graph</td>
<td>– Two products used as input to</td>
<td>before transition</td>
</tr>
<tr>
<td></td>
<td>– Six products as initial input to</td>
<td>transition</td>
<td>– Two products as initial input to</td>
</tr>
<tr>
<td></td>
<td>transition</td>
<td></td>
<td>transition</td>
</tr>
<tr>
<td><strong>Target SPL</strong></td>
<td>Component library and a</td>
<td>#ifdef-based configurable platform</td>
<td>Component library</td>
</tr>
<tr>
<td><strong>Architecture</strong></td>
<td>configurable framework connecting</td>
<td>based (150% view)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>the components</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Status</strong></td>
<td>Transition to SPLE in progress</td>
<td>Transition to SPLE completed</td>
<td>Transition to SPLE initiated</td>
</tr>
</tbody>
</table>

5.1.1 Case Study Selection

The framework presented in Chapter 4 stems from our interactions with numerous companies that use cloning to realize product lines and from reports published in the literature. For the deeper analysis presented in this chapter, we selected three of these companies using the following criteria:

1. A company employs (or used to employ) cloning to realize its product portfolio.

2. We have access to a significant amount of company-specific data and a deep understanding of the development process.

3. The companies are diverse enough to draw insightful conclusions.

As the last criterion shows, we intentionally focused on the needs and requirements of companies at different levels of maturity with respect to their support of cloning and transition to SPLE: from those that already completed the transition to those that are only exploring its feasibility. Our first company is currently in the advanced stage of the transition from cloning to a managed SPLE approach (Section 5.1.3 – case study #1). The transition in the second case, Danfoss Drives, is completed and has been reported in [75, 74] (Section 5.1.3 – case study #2). In the third case, the company still relies on cloning to implement its commercial products, and the transition to SPLE is just being explored (Section 5.1.3 – case study #3). Studying the development practices of the third company allowed us to gather first-hand experience and needs of those practitioners that still employ cloning to establish new variants. Confidentiality issues prevent us from sharing the names of the first and the third companies; both these companies are from the aerospace and defense domain.
5.1.2 Data Collection and Analysis

In the first case study, we collaborated with a research partner who acted as a consultant for the company, assisting the company’s engineering team in creation of the product line transition strategy and the desired product line architecture. The consultant participated in the project for about two years and had frequent interactions with the team, including weekly calls and bi-monthly visits. The goal of these interactions was to assist in creation and assessment of the target product line architecture and suggest implementation techniques supporting the transition. Our interpretation of the company-specific processes and development artifacts relied on the deep understanding this consultant gained while performing action research within the studied organization – a process that balanced problem solving actions with data-driven research to understand underlying causes and enable future predictions about organizational changes [126].

In the second case study, we analyzed a series of publications by an engineer directly involved in the transition and a consultant assisting the company in this process [76, 75, 74]. The publications provided a detailed summary of the company’s activities over more than five years. We enhanced our understanding of this case study by analyzing a summary of a one-hour interview performed in January 2012 with a key architect responsible for the company’s product line efforts. The interview summary was provided to us by a research partner. This summary and subsequent discussions with the partner helped clarifying the processes described in the papers.

In the third case study, we collaborated with a consultant who had worked with the company for more than five years and had weekly interactions with the engineering and management teams. We had access to transcripts of five one-hour interviews conducted with the company’s developers and managers in early 2012. The purpose of these interviews was to study processes related to the management of clones and the transition effort. During 2012, we also conducted several in-person meetings with the employees of the company in order to get deeper insights into the development practices of this company and provide advice on the product line adoption strategy.

Our analysis of the collected data focused on each company’s development environment, including the tools it used, the development artifacts it maintained, the processes it followed, the target SPLE approach the company aimed to adopt and the challenges it faced. These are described in the remainder of this section and also summarized in Table 5.1.

5.1.3 Description of the Case Studies

Case study #1

For the first case study, we analyzed a software product line of a large aerospace company that was developed over the course of 15-20 years by approximately 300 engineers, half of them system and half software engineers. As summarized in the second column of Table 5.1, the product line of this company is realized using four types of artifacts: system requirements, software requirements, executable design models and tests. System and software requirements are maintained in an internal requirements management tool similar to IBM Rational Doors1. Design models are specified in SCADE2, with code being automatically generated from the design models using model-driven technologies [154]. Due to regulatory requirements3, detailed vertical traceability is established between the artifacts: from system to software requirements and further to the design models and code.

The development process strictly follows a waterfall model. Over 50 products of the product line are implemented by cloning (captured by a complex clone derivation graph) with the cloned variants kept as separate

---

1http://www-01.ibm.com/software/awdtools/doors/
2http://www.esterel-technologies.com/products/scade-suite/
3DO-178B
branches in a Software Configuration Management (SCM) system. The goal of the company is to establish a component library as well as a configurable framework architecture connecting the components. Those are to be used as a basis for all future products of the product line. However, the company does not intend to re-implement existing products on top of the common framework – these products are to be maintained as is.

Case study #2

The second case study, Danfoss Drives, is one of the largest producers of frequency converters – electronic power conversion devices used to control shaft speed or torque of a three-phase induction motor. According to the reports in [75, 74], the company produces three main product series – for the manufacturing industry, for HVAC (Heat, Ventilation and Air-Conditioning) systems, and for the water segment. In addition, the company provides many specialized products, e.g., for crane and specific textile industry applications. It employs 1200 individuals globally and approximately 200 in the R&D department. About 60 of them are embedded software developers, working in four development sites located in four different countries. Almost all of the functionality in a frequency converter is handled by software.

The main characteristics of the Danfoss development environment are summarized in the third column of Table 5.1. Product development is addressed by a matrix organization where projects are carried out by the integrated product development with dedicated personnel and target markets. Line organizations provide skilled developers for projects to draw upon to ensure application and domain knowledge for timely product releases. Exceptions are the testing organization which is “global” to all products, and an Embedded Software Platform team responsible for enforcing reuse.

The company follows an iterative development approach, with most artifacts being C++ code and test cases. Since late 1990s, the idea of managed reuse was promoted within the Danfoss development organization, resulting in creation of an object-oriented framework architecture written in C++. Yet, the evolution of the code base was not really controlled, and reuse between products was done in a “clone-and-own” way: code was taken from one project branch to another via merging in an SCM system.

In 2005, the company decided to migrate its recently launched new series of products to an SPLE approach, specifically, a configurable platform (sometimes referred to as “the 150% view”) – a common code base in which variability points are selectable via compiler switches. The new series of products initially contained four cloned variants, two of which were selected for a pilot study. The code of the selected variants was analyzed to identify common and variable parts, and refactored into a common platform.

In the following years, the remaining two products of the new series were integrated into the common platform, and all additional products were developed on top of it. For the platform-managed code, each product team could either use the code “as is” or create a branch from it with product-specific changes and additions. Products of other series were still maintained as distinct clones, with some being phased out.

At the time of writing, all products of the new series are derived from a shared platform in a feature-oriented manner. The company uses pure::variants⁴ to manage its platform containing over 1100 features and hundreds of configurations used by customers.

Case study #3

The third case study, just like the first one, comes from a large aerospace and defense company (see the last column of Table 5.1). We analyzed a relatively recent software-intensive product line, developed over the last five years.

⁴http://www.pure-systems.com/
It stems from two similar products which became successful and grew into a family containing five commercial products and a couple of in-development prototypes. Software is responsible for almost all of the functionality of these products.

The product line was developed by around 30 R&D professionals involved in different development stages – from requirements engineering to testing, with a centralized development team responsible for all products of the product line. Since the products are supportive rather than safety-critical, the company spends less time on regulatory issues, compared to the first case. For the same reason, design artifacts are rather informal and not always synchronized with code which is hand-written in C. The company uses IBM Rational Doors to maintain requirements. It follows rigid testing procedures at all development stages including customer sites.

The product line management team indicated that they realized that they were developing a product line only after the success of the initial products, when customers started to ask for additional variants. The company made an effort to establish a library of reusable components early in the process, yet this library was often bypassed, and developers used cloning to rapidly serve their customers’ needs. Moreover, together with the shortcoming of cloning, developers saw several benefits of the approach: it did not require any upfront investment; it was a rapidly available and easy to use practice; and it gave the developers freedom to make necessary changes in their code, without any need to synchronize their work with others. For these reasons, the team mostly continued to clone.

With an increase in the number of products, a transition to SPLE received a higher priority. The company thus aims to inspect the existing implementations and identify reusable configurable components with well-defined interfaces. These components will then form a shared component library, allowing new products / projects to pick components from it. Each component should be independently testable so that the focus of the new product testing becomes integration testing. Thus, the library of components is anticipated to speed up both the development and the testing. Since the entire product line is developed by one centralized team, the team is able to redefine the development processes such that this time the library is put into use and kept up-to-date.

At the time of writing, an initial process of identifying commonalities and variabilities in the developed variants and building a library of reusable components has started.

5.2 Analysis of Merge-Refactoring Activities

We now conduct a detailed analysis of the development activities that the companies perform as part of the merge-refactoring process. We then identify their atomic steps and map those steps to instances of the operators (see Table 5.2 for the summary). The goal of this analysis is to validate the conceptual framework proposed in Chapter 4 by using the operators as a logical description of the steps and to discuss organization-specific automation opportunities.

5.2.1 Case Study #1

The transition to SPLE proceeded in a top-down fashion, i.e., it started by first analyzing requirement documents, then creating a common architecture, and then building common assets, as schematically shown in Figure 5.1. At the time of writing, the architecture creation step has concluded and its verification is yet to commence. In terms of implementation, the created component libraries cover about 10% of the architecture. The assets were built incrementally using three major activities, described below.
Figure 5.1: Transition Activities in Case Study #1.
Activity 1: Analyze commonalities and variabilities

This activity involves a comparison between a few variants in order to assess the variability scope and build an initial feature model.

1.1 Compare requirement documents

As the first step, requirement documents are analyzed and compared to each other at the structural level. In this case study, all requirement documents follow the same structure: major capabilities are described in separate document sections which are grouped hierarchically. Each section contains a set of requirements statements. Every variant has its own requirements document. If a variant does not have a certain capability, the corresponding section in the requirements document for that variant is missing.

\textit{findFeatures}: In effect, each document outline represents a feature tree of the corresponding variant, revealed using two conceptual operations. The first, captured by the \textit{findFeatures} operator (row 1 in Table 4.2), returns the set of all features realized by a given variant. In this case study, the implementation of the operator simply collects all sections in the corresponding document and treats them as features: section titles and descriptions are treated as feature labels and descriptions, respectively. The result has to be reviewed by a domain expert: while the majority of sections represent features, a few may not. Also, some section titles are rather long, and shorter feature labels have to be created.

\textit{dependsOn}?: The second operation retrieves dependencies between features and is captured by the \textit{dependsOn}? operator (row 3 in Table 4.2). There are several possible forms of such dependencies, declaratively specified by the input \textit{property} parameter. For example, one feature can require another in order to compile. Behavior dependencies, when one feature requires the other in order to operate correctly, are more complex. In our case, the input \textit{property} of the operator is configured to determine features as dependent if their corresponding subsections have a parent / child relationship. The operator returns a \textit{set of witnesses}, each demonstrating the \textit{dependsOn} relationship between the artifacts of \(f_1\) and \(f_2\) (or \textit{none} if the features are independent). In this case, a \textit{witness} is just a pair of section numbers where one is a subsection of the other.

The result of applying these two operators is a tree containing mandatory features of each variant. Note that in this case study, the particular implementations of the two conceptual operators can easily be merged and optimized to produce a candidate feature tree in one pass, but this is not always true in general.

\textit{same}?: The requirement documents are further compared at the statement level to find pairs of features from distinct variants that are considered similar. Conceptually, this can be seen as an application of the \textit{same}? operator (row 4 in Table 4.2), which determines whether two functionalities are equivalent by considering their feature declarations and implementations. Like \textit{dependsOn}?, this operator uses a \textit{property} that specifies equivalence criteria and returns a \textit{set of witnesses} exemplifying the disagreements (or none if the features are equivalent).

In this case study, the implementation of \textit{same}? matches features based on their lexical similarity (the Levenshtein distance metric [99]) of their corresponding sections in the requirement documents as well as the individual statements of these sections. Features that correspond to sections containing a significant number of similar statements are considered similar as well. This implementation works well since in this case documents are mostly produced by cloning. Yet, the result still needs to be reviewed and refined by a domain expert.

1.2 Compare implementations at a feature level

\textit{findFeatureImplementation}: Next, the design models of several variants are compared to each other. In this case, due to the regulatory constraints, every element of design models contains traceability links to the requirements it implements. Thus, establishing traceability between a feature and its corresponding feature implementation – an instance of the \textit{findFeatureImplementation} operator (row 2 in Table 4.2) – is trivial,
and the property input parameter of findFeatureImplementation is simply configured to look for existing traceability links.

same?: Once the traceability is established, the features of distinct variants are compared again, this time considering design-level functions. This operation can be seen as a separate application of the same? operator. While conceptually both comparison operations could have been performed at once, in this case study they were executed separately, mostly because the comparison at the implementation level was done by manual inspection. We envision that this step can be automated using model matching techniques [161]. Regardless of how the implementation-level matching is done, its result still has to be reviewed by a domain expert and modified where necessary. For example, some of the differences can correspond to new, rather than already defined, features and hence should be lifted to the feature tree level.

1.3 Create an initial version of a feature model

merge: This step involves composing the feature trees of the analyzed variants, unifying those features that are deemed similar. This can be seen as an instance of the conceptual merge operator (row 6 in Table 4.2): for \( n \) input systems, a set of matches between the corresponding input elements, and a resolution parameter defining how to handle conflicts that occur when input elements disagree. The operator produces a merged system. In our case, we are only interested in feature trees which are given to the merge operator as a parameter. The set of matches contains features decided by a process similar to the operator same?, and the resolution parameter specifies which variant to prefer if the two variants disagree. Again, while it is performed manually in this case study, we can envision automation of the operator, relying on existing works on feature model composition [1].

Activity 2: Develop a common architecture

In this activity, three recent variants are analyzed in order to create a common architecture, define modules, interfaces and connections. The variants are picked so that they appear far apart in the cloning derivation graph and implement a diverse set of features, ensuring a sufficient scope coverage. The architecture is then validated using another three variants.

2.1 Extract a common architecture

reorganize: The chosen variants are first reorganized and “normalized” to create modules that cluster related artifacts together, usually by their functionality and also by applying certain architectural patterns. To support this task, we use the reorganize operator (row 7 in Table 4.2), which receives a system and a property that declaratively defines the nature of the required reorganization, as defined above. The operator returns a modified version of the system, after the reorganization has been performed.

When possible, the implementation of reorganize should ensure that the traceability between implementation artifacts and their corresponding feature declarations, established earlier using findFeatureImplementation, is preserved. Alternatively, new traceability relationships are to be created. In this case study, the reorganization was performed manually while keeping the traceability relationships. Automation relying on existing model and code refactoring techniques [110] might also be possible and should be explored further. Yet, we do not envision a fully automated process but rather a set of automated techniques that assist the domain expert as necessary.

merge: Following the reorganization, the artifacts (both feature trees and implementations) of distinct variants are combined using merge, creating a candidate architecture and a refined feature model, with traceability between them. This step was also performed manually, involving architects with experience in building some of the previous variants. The outcome was a document describing the resulting architecture.

2.2 Verify the common architecture
**reorganize**: The created candidate architecture and the feature model are further reviewed to validate their fitness and the ability to support additional three variants, distinct from those that were used as input to the previous step. This part of the review, conceptually captured by an instance of reorganize applied on the generated feature-based system, was also performed manually. This step reveals the need of applying the reorganize operator for both the cloned variants before the transition and the product line architecture after the transition, thus obtaining a *feature-oriented system* as a parameter rather than a concrete *variant*.

**Activity 3: Develop common assets**

The development of common assets is done incrementally, with each increment covering a different part of the architecture. The created assets, which include requirements templates (text documents with placeholders) and component implementations, are intended to be used as libraries in the development of new variants.

**findFeatureImplementation, same?**: The activity relies on the ability to trace features from the generated feature model to their implementations in the analyzed variants, as established by the findFeatureImplementation operator, as well as the ability to compare and identify disagreements at the implementation level, using the same? operator. Then, existing variants are manually inspected and consulted during the development of the common assets: most of the assets are built from scratch by a domain expert to fit the more general context, to distill robust abstractions and ensure a high degree of modularity.

### 5.2.2 Case Study #2

In contrast to case study #1, the transition to SPLE in the second case proceeded in a bottom-up fashion, starting from code differences and working up to a feature model. The process was carried out incrementally, refining the common platform to improve its quality and modularity by refactoring along the way. The process involved four major activities, schematically shown in Figure 5.2 and described below.

**Activity 1: Merge initial set of variants**

**merge**: As the first step, the implementations of two subsystems is compared to each other on the code level using a textual diff tool, and further unified. During the unification, conditional compilation directives with `#ifdef PRODUCT_IS XXX` commands are introduced wherever the source code files disagreed with each other. This activity is conceptually represented by an instance of the merge operator, where the matches parameter is empty (i.e., only identical elements are considered similar) and the resolution is to insert conditional compilation directives that represent original variants.

**findFeatures, findFeatureImplementation, interact?**: The implementations of findFeatures and findFeatureImplementation are trivial: a feature is created for each inserted directive and traced to its corresponding code. A *mutually-exclusive* relationship is defined between each pair of features that correspond to distinct variants. That provides a trivial resolution to the potential interaction of features that were not designed to work together. Feature interactions, ranging from purely syntactical to behavioral, are detected using the interact? operator (row 5 in Table 4.2), which obtains as input a *property* specifying the form of interactions to be checked.

As the result of applying the above operators, a *feature-oriented system* containing a common “150% view” representation of the code artifacts is created. The system contains a “primitive” feature model with mutually-exclusive features that correspond to input variants. This feature model drives the definition of two different makefiles for building the two original products from the unified code base.
Figure 5.2: Transition Activities in Case Study #2.
Activity 2: Refactor code to introduce meaningful features

reorganize: In this activity, product-specific #ifdef statements are manually inspected by the domain expert and replaced with feature-specific statements #if HAS_FEATURE.xxx == 1. This activity, seen as an instance of the conceptual reorganize operator, includes refining the set of features, their corresponding code and the traceability relationships between them.

In the same step, product-specific makefiles are also replaced by feature-specific counterparts: for each feature, one makefile holds the list of artifacts to build when the feature is enabled, whereas the other holds the list of artifacts required when the feature is disabled (in most cases – an empty list).

Activity 3: Bring additional variants onto the platform

The remaining variants are incrementally integrated into the constructed platform, one-by-one, until the full coverage is achieved. First, the code of a variant is combined with the existing platform, using product-specific #ifdef directives (via the operators merge, findFeatures, findFeatureImplementation and interact?, as described above). Later, the code is manually refactored (via reorganize) to improve structure and modularity and to include feature-specific statements.

Activity 4: Create a feature model and transformations

As the final step, compiler directives used to configure the code are extracted into a feature model that is managed by pure::variants.

4.1 Extract features

findFeatures, findFeatureImplementation: This step can be seen as an instance of the findFeatures operator whose implementation trivially extracts the existing compilation directives. In the Danfoss case, the extracted directives were listed in a text file which was further imported to pure::variants. In this case, detecting traceability between the extracted features and the code that corresponds to them (the instance of the findFeatureImplementation operator) is also trivial.

4.2 Define relationships between features

dependsOn?, same?, interact?, reorganize: Since the above process does not produce dependencies between features, the created feature model lists all features as optional. Refining it to include richer relationships such as grouping or alternatives conceptually relies on operators that can determine such relationships between features – dependsOn?, same?, interact?, as well as on the ability to reorganize the resulting feature model and improve its structure [157]. In the Danfoss case, this step was carried out manually by domain experts. Finally, transformations for creating makefiles for a specific feature configuration were developed using pure::variants, again, in a straightforward manner.

5.2.3 Case Study #3

In the third case, the transition process to SPLE has only just begun. The company started from the analysis of the product portfolio and the existing product artifacts, aiming at capturing commonalities and variabilities among the current set of variants. Later on, the development of a reusable component library is planned, as shown in Figure 5.3. At this stage, all activities in this case study are performed manually, with the goal to identify and investigate automation opportunities. Thus, we do not discuss automation of the operators yet.
Figure 5.3: Transition Activities in Case Study #3.

**Activity 1: Analyze commonalities and variabilities**

The goal of this activity is to create a feature model that scopes the product portfolio and explicates the list of supported features. Unlike in case study #1, there is no clear documentation listing the set of all existing features.
Thus, the activity is performed in two steps.

1.1 Brainstorming

**findFeatures**: During a few brainstorming sessions, the development team members holding various organizational roles (i.e., managers, architects, developers and testers) proposed an initial set of features (findFeatures), focusing mainly on capabilities that are perceived to be distinguishing between the developed product variants and those that customers use to describe the products. The set of elicited features is captured in a text document.

**dependsOn??, same??, interact??**: Identifying requires and mutually exclusive relationships between the found features is again performed manually based on the team’s familiarity with the developed products. A requires dependency is introduced every time one feature depends on the presence of the other (as represented by the dependsOn? operator), while a mutually exclusive relationship is introduced every time features are not designed to work together (interact?) or implement functionality that is perceived similar (same??).

1.2 Comparing artifacts

**findFeatures**: In this step, the initial set of features is further refined by inspecting development artifacts such as requirement documents, code and test descriptions. Artifacts are compared to each other manually and, similarly to the case study #1, disagreements “hint” at additional variation points, which become features. This is an instance of the findFeatures operator, performed manually.

**dependsOn??, same??, interact??**: Relationships between features are also detected manually. Conceptually, require dependencies between features (dependsOn??), distinct implementations of similar features (same??) and conflicting features (interact?) are considered. The set of the discovered features and relationships between them produce a version of a feature model.

**Activity 2: Develop a reusable library (proposed)**

**merge, reorganize**: The company plans to build a library of reusable components in an incremental manner, by analyzing product variants one by one. Each analyzed variant either augments the library with additional components (an instance of the merge operator) or refines parameters of the components already in the library (an instance of merge followed by reorganize).

**reorganize, findFeatureImplementation**: After each iteration, the assets might be extended, refined and refactored (reorganize) in order to improve encapsulation and ensure that components can be usable in a diverse set of variants. The feature model is to be consulted at this stage to ensure that the components are easily configurable to provide the functionality perceived as important. If possible, traceability between features and the artifacts that implement them is to be established (findFeatureImplementation).

5.3 Analysis of Activities Related to Maintenance of Existing Clones

In this section, we analyze activities related to maintenance and evolution of existing clones and show that these activities can be mapped to instances of the conceptual operators defined in Chapter 4. Our analysis demonstrates the applicability of the operators to improve the cloning experience and to support the clone-based SPL approach. Since clone maintenance activities are similar in all our case studies, we discuss them collectively (see Table 5.2 for the summary).
Activity 1: Propagate changes between variants

`findFeatures, findFeatureImplementation, same?`: Changes made in one cloned variant might be useful in another. To locate such changes, correspondences between features of a variant (detected using `findFeatures`) and the artifacts that implement them (detected using `findFeatureImplementation`) are established. Differences between distinct implementations of the same feature (detected using `same?` and represented by a set of witnesses) are inspected and propagated between variants. In the simplest form, the differences can be detected using a textual difference tool, as was done in case study #2. Detecting more sophisticated behavioral differences is also possible, e.g., using the technique in [72].

Activity 2: Share features between variants

`findFeatures, findFeatureImplementation`: Like individual changes, complete features can be shared between distinct product variants. Here, again, a list of features, together with traces to implementation-level artifacts, is identified and maintained (instances of `findFeatures` and `findFeatureImplementation`).

`same?, dependsOn?`: Different implementations of the chosen feature of interest are compared to each other (using `same?`) selecting the one found most appropriate. Further, the set of other features it requires (detected using `dependsOn?`) is inspected. If those features are not part of the target product, some of their artifacts have to be transferred to the target product together with the selected feature, to ensure its correct operation, as discussed in [146].

`interact?, merge`: Next, the `interact?` operator verifies whether the new feature interferes with the functionality of the existing ones in the target product variant. Following that, `merge` integrates the selected feature and those that it requires in the target system, resolving the conflicts identified by `interact?`.

Activity 3: Retire features

`findFeatures, findFeatureImplementation, dependsOn?`: While new features are added, some of the existing features might no longer be needed. Like in the previous activities, the set of features and their corresponding implementations is detected (using `findFeatures` and `findFeatureImplementation`) and features that depend on the one being removed are identified (using `dependsOn?`). Since the functionality of such features should not be affected by the feature retirement, artifacts that these features use are not removed.

Activity 4: Establish new variants

`findFeatures, same?, dependsOn?, interact?, merge`: Depending on the maturity of the transition process, there might still be a need to create new variants following the existing cloning practices, like in the case study #3. In such cases, the feature portfolio of all existing variants (detected using `findFeatures`) is inspected, and the variant with the most similar functionality is used as a starting point for cloning. Then, features that are not required in this variant are removed, as described in Activity 3, while additional features are either developed from scratch or “borrowed” from other variants, as described in Activity 2. Instead of removing features, one could also mark them as optional, to start introducing variability that eventually leads to a single copy SPL representation.
### Table 5.2: Applicability of the Operators.

<table>
<thead>
<tr>
<th></th>
<th>findFeatures</th>
<th>findFeatureImplementation</th>
<th>dependsOn?</th>
<th>same?</th>
<th>interact?</th>
<th>merge</th>
<th>reorganize</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cloned SPL</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merge-refactoring</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Case 1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Case 2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>Case 3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td><strong>Clone-based SPL</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Case 1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>Case 2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>Case 3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>⊥</td>
<td>⊥</td>
</tr>
</tbody>
</table>

### 5.4 Limitations and Threats to Validity

Limited access to the studied companies prevented us from using a common data collection approach in all of the case studies. It also prevented us from validating our interpretation of the companies’ development activities and the breakdown of these activities to instances of the operators with the companies’ personnel. Such lack of validation is a considerable threat to the validity of our work. Yet, we discussed and validated our own analysis with research partners who work closely with the studied companies over the course of several years and have a very detailed view of their development processes.

While we showed that the current set of operators is reasonable for the case studies that we analyzed, it is likely incomplete or can be presented at a different level of granularity. For example, we chose not to introduce an operator that allows to delete a certain product line functionality as, in our case studies, it could be implemented by a rather straightforward composition of findFeatureImplementation and dependsOn?. Extending the set of operators with a dedicated delete operation could also be a plausible choice though. Analyzing additional case studies can help further refine the set of the operators and their exact properties by identifying the missing cases, creating dedicated operators for frequently used combinations and splitting operators into finer “building blocks”, when needed. One can also come up with a different set of domain-specific operators that closer reflect the necessary development activities for each case study and exploit the increased degree of specialization. Yet, we believe that the generic operator-based view provides valuable opportunities to systemize and compare existing works, and share experiences between companies using a common vocabulary. Our goal was thus not to come up with an optimal set of operators but rather exemplify the usefulness of the operator-based view.

During our analysis, we observed that in each specific case and for each organization, the implementation of the operators varied significantly. For example, in case study #1, the implementation of findFeatureImplementation is trivial and relies on traceability information that the company has to maintain between their artifacts. No such information exists in case studies #2 and #3, making the implementation of the operator significantly more challenging. Moreover, even within a single organization, artifacts of different types require different implementations
of the operators. For example, in case study #1, the same? operator was applied both to requirement documents and model-level artifacts. In some cases, the operators cannot be automated at all and many times the results of the operators have to be revised by human experts. Yet, breaking the solutions into a combination of automated and manual “building blocks” allows organizations to locate and reuse existing work, as well as to estimate the investment in managing cloned variants in a more accurate manner. We believe that it thus leads to more efficient development and maintenance practices.

Finally, an additional weakness of this research stems from the fact that we did not attempt to refute the proposed framework, e.g., explicitly collect any evidence that contradict its usefulness. That could be a subject of follow-up research.

5.5 Summary

In this chapter, we empirically validated the applicability of the framework introduced in Chapter 4. Specifically, we showed that the proposed set of operators is sufficiently expressive to realize development activities of three industrial organizations that rely (or relied) on cloning to realize their product lines. We described in detail the development activities performed by these organizations, both when merge-refactoring cloned variants into a single-copy SPL representation and when maintaining existing cloned variants, and showed that these activities can be expressed as instances of the operators. We discussed existing and possible organization-specific implementations of the operators and outlined the limitations of our work.

As a next step, in Chapters 6–8, we address some of the gaps identified in our systematization effort. We contribute implementations of findFeatureImplementation (a.k.a. feature location) and merge operators applicable specifically to collections of cloned product variants.
Chapter 6

Locating Features in Cloned Variants

In this chapter, we take a closer look at the problem of locating features in a collection of related product variants (the `findFeatureImplementation` operator defined in Chapter 4). As discussed in Chapter 4, a feature implemented in one cloned variant might often be useful for another. Thus, its implementation should be located and copied to that different variant, promoting sharing of features between products. Moreover, identifying traceability between product features and artifacts that realize those features is a necessary step during merge-refactorings: these relationships are required when building an SPL architecture but are rarely documented.

Numerous feature location techniques for tracing features to code-level artifacts have been proposed in the literature (see Appendix A for a detailed survey). In a nutshell, these techniques are based on static program analysis [132, 173, 166], dynamic analysis [92, 47], information retrieval (IR) techniques [105], change set analysis [180], or a combination of several aforementioned techniques [153, 69, 48, 183, 152]. Obviously, each of the existing feature location techniques can be used for locating features in cloned variants when treating these variants as singular independent entities. However, the accuracy of many contemporary techniques still remains low (see [129] for a discussion); improving it for the specific case of locating features in related product variants can be beneficial.

We focus on locating distinguishing features – those that are present in one but not all variants. We propose to extend the set of heuristics used by existing feature location techniques based on additional information available when comparing the code of a variant that contains a particular feature of interest to the one that does not. The comparison provides an initial “coarse-grained partitioning” of a program into relevant and irrelevant parts and assists the feature location process: the features of interest are implemented in the `unique` parts of the program, providing a clue where to locate them. We detect and explicitly capture information about the `unique` parts of the program as a separate artifact, called a `diff set`, and use it in the proposed heuristics for improving the accuracy of feature location. Even though the implementation of distinguishing features fully resides in `diff sets`, providing a clue as to where to locate them, computing this artifact is not sufficient by itself: since a `diff set` can contain tens or even hundreds of distinguishing features, “telling them apart” is still challenging.

In this chapter, we experiment with a family of static feature location algorithms that employ an iterative program exploration approach, returning to the user a ranked list of relevant program elements. `Dora` [69], `Suade` [132] or `FRAN` [152] are some examples of this family of techniques which we describe in more detail in Section 6.2.
Contributions. This chapter makes the following contributions.

1. We specify a subarea of feature location (a.k.a. the \texttt{findFeatureImplementation} operator), focusing on identifying code that corresponds to \textit{distinguishing} features – those that are present in one variant of a program but absent in another.

2. We present a set of heuristics for improving the accuracy of existing feature location techniques when finding distinguishing features.

3. We implement these heuristics, extending two feature location techniques whose code was available to us. Our implementation provides an accurate realization of \texttt{findFeatureImplementation} for sets of cloned product variants.

4. We show that our heuristics enable substantial improvements in the accuracy of feature location (up to several hundred percent), while the cases of reduced accuracy are rare and minor. To avoid evaluator bias, our evaluation is performed on fifty-two benchmarks identified in prior works [134, 129, 43].

This chapter extends a short paper which presented an overview of our approach [139]. Here we give the complete report. We proceed as follows. Distinguishing features and \textit{diff sets} are exemplified in Section 6.1. Section 6.2 gives the necessary background on the program representation and static iterative feature location approaches, leading to a more precise definition of distinguishing features and \textit{diff sets} in Section 6.3. \textit{Diff set}-based heuristics are presented in Section 6.4. We describe our evaluation methodology in Section 6.5 and present the evaluation results in Section 6.6. Related work is discussed in Section 6.7. We conclude with a summary in Section 6.8.

6.1 Example

Consider a tiny snippet of the call- and data-flow graph of the Eclipse integrated development environment\footnote{http://www.eclipse.org/}, presented in Figure 6.1. Our goal is to locate code of a new feature which enables Emacs-style incremental search. This feature, previously studied in [129], allows the user to \texttt{repeat} a conducted search operation, providing key accelerators both in the backward and the forward direction. To support the desired functionality, method \texttt{run} of the class \texttt{IncrementalFindAction} (element \#1) sets the value of the variable \texttt{fForward} (element \#2) based on the user input and initiates a call to method \texttt{beginSession} of \texttt{IncrementalFindTarget} (element \#3). That method reads the value of \texttt{fForward} variable and, among other actions, initiates a call to method \texttt{repeatSearch} (element \#4). \texttt{repeatSearch} initiates a call to method \texttt{findAndSelect} of \texttt{TextViewer :: FindReplaceTarget} (element \#7), passing as a parameter the previous search expression stored in variable \texttt{fPrevFindString} (elements \#5). The value of this variable is set by method \texttt{leave} of \texttt{IncrementalFindTarget} (element \#6) invoked when the user ends the search session.

Elements \#1-6, shaded in the figure, contribute to the implementation of the new Emacs-style incremental search feature. The remaining elements contribute to the already existing incremental search functionality. For example, element \#7 is called by several methods, including method \texttt{replaceAll} of the class \texttt{FindReplaceDialog} (element \#11). Method \#7 itself calls method \texttt{findAndSelectInRange} of \texttt{TextViewer :: FindReplaceTarget} (element \#8) which performs the search in a given text range, as specified by the variable \texttt{fRange} (element \#9), whose value is set by method \texttt{setScope} of \texttt{TextViewer :: FindReplaceTarget} (element \#10). Even though the Emacs-style incremental search implementation requires those elements in order to operate, we do not consider
them being part of the feature, but rather contribute to a set of different features that existed before the new one was introduced. That is, our new feature depends on those existing features rather than contains them and thus elements #7-11 are considered irrelevant to the implementation of the studied feature and are not shaded in Figure 6.1.

Comparing the version of the software that includes the implementation of the Emacs-style incremental search feature to the one that does not produces a diff set containing all elements that are different between these two versions. In Figure 6.1, different elements have dotted-line frames while common elements are denoted by solid-line frames.

Since our goal is to detect distinguishing features that are present in one version of the software and absent in another, all relevant methods are located in the diff set. Yet, together with the code of the newly introduced feature of interest, diff sets often include implementation of several additional distinguishing features. That is, they can contain elements which are not specific to the studied feature, e.g., element #10. This element was also changed between the versions but does not contribute to the feature of interest. Thus, presence of an element in a diff set cannot solely determine its relevance. However, diff sets can be taken into account during the feature location process, as discussed in Section 6.4.

### 6.2 Background

We now describe the family of feature location approaches which we use in order to experiment with our diff set-based heuristics: a family of static algorithms, interactively exploring a program structure. Such algorithms usually consider program elements such as methods and fields. They explore a program starting from a given seed – a set of input program elements that are of interest to the user. The exploration follows program relationships such as method calls and field accesses, collecting elements that are considered relevant to the feature of interest. A ranked list of collected elements is returned to the user. We use the definition of [132] for a program, a program element, and a relationship between program elements:

**Definition 7** [132] A program $P = (E, R)$ consists of a set of elements $E$ and a set of relationships between these elements $R$. A program element $e \in E$ is any element that can be individually investigated by a developer. A relationship $r = (l, e_1, e_2) \in R$ is a program dependence of type $l$ between product elements $e_1$ and $e_2$. 

---

**Figure 6.1:** A snippet of the Eclipse call and data flow graph, containing the Emacs-style incremental search feature.
Chapter 6. Locating Features in Cloned Variants

Feature location algorithms of this family vary in the exact set of programming elements and relationships they consider and the scoring algorithm they employ. For example, Suade [132] uses an approach in which fields and methods are scored based on their specificity – an element is specific if it relates to few other elements, and reinforcement – an element is reinforced if it is related to other elements of interest. It considers relationships such as field access and method call hierarchies. Dora [69] traverses method call hierarchies only and scores methods based on their lexical similarity to a natural language query that is provided by the user to describe the feature of interest. FRAN [152] also traverses method call hierarchies to identify a set of relevant methods which are further scored using the HITS web mining algorithm [89].

Algorithm 1 gives a high-level description of such feature location process. The exploration step follows program relationships until a certain exploration depth is reached (line 3). Limiting the depth of exploration is often used to prevent the techniques from collecting a high number of irrelevant elements. Each analyzed element is scored by the algorithm based on its lexical and/or syntactical properties: assigned a number between 0 and 1 (line 5). Elements scored higher than a certain, empirically set, relevance threshold $t_R$ (lines 6-8) are collected into set $S_R$ (line 7), to be returned to the user (line 14). An exploration threshold $t_E$ is used to select elements whose relationships are to be traversed further (lines 9-12). All neighbors of these elements ($S_N$) are added to the exploration set $S_E$ (lines 10-11) and explored during the next iteration of the algorithm. The value of this empirically set threshold is either explicitly encoded in the algorithms (e.g., in Suade) or is implied – the user analyzes the intermediate results for their relevance and then manually initiates the next iteration (e.g., in Suade and FRAN).

Algorithm 1 Feature Location ($P=(E,R)$: Program; seed: $\{e_i \in E\}$; depth: Integer)

1. $S_E \leftarrow \text{seed}$ /* exploration set */
2. $S_R \leftarrow \emptyset$ /* results set */
3. while ($\exists e \in S_E \text{ s.t. depth}(e) \leq \text{depth}$) do
4. 
5. score $\leftarrow \text{score}(e)$ /* score $\in [0..1]$ */
6. if (score $\geq t_R$) then
7. 
8. end if
9. if (score $\geq t_E$) then
10. $S_N \leftarrow \{e' | \exists (l, e, e') \in R\}$ /* neighbors set */
11. $S_E \leftarrow S_E \cup S_N$
12. end if
13. end while
14. return $S_R$

6.3 Distinguishing Features and Diff Sets

In this section, we give our definition of distinguishing features and diff sets. As discussed in Chapter 4, it is often unclear what the exact properties of the located feature implementations are and which elements should be considered part of it, e.g., should the main method of a program be considered part of every feature or not? This work focuses on locating the minimal necessary set of elements that contribute to a feature. For distinguishing features, these are the elements that were added or modified in order to introduce the feature of interest in a certain program variant. A distinguishing feature can then depend on other features of the program, as discussed in Section 6.1. The definitions below define the exact properties of feature implementations located with our approach.
Definition 8 Let $P$ and $\bar{P}$ be programs. A feature $f$ is distinguishing for $P$ if it is implemented by a set of elements in $P$ that are either different or absent in $\bar{P}$.

Distinguishing features reside in **diff sets**:

Definition 9 Let $P$ and $\bar{P}$ be programs. A diff set of $P$ compared to $\bar{P}$ (denoted by $\Delta P[\bar{P}]$) is the set of all elements of $P$ that do not have corresponding elements in $\bar{P}$. That is, $\Delta P[\bar{P}]$ is the set of all elements of $P$ that are either different or absent from $\bar{P}$.

The granularity of elements in a *diff set* corresponds to the granularity of the elements considered for feature location. For example, a *diff set* used to enhance a technique locating Java methods and fields (e.g., *Suade*) contains elements of these two types.

Ideally, *diff sets* should be limited to high-level *semantic* program changes [58] rather than contain line-level differences or such syntactic differences as program reformatting and refactoring. However, we do not restrict Definition 9 to *semantic diff sets* only as our approach is applicable to any available *diff set*.

### 6.4 Diff Set-Based Heuristics

*Precision* and *recall* are commonly used to measure the accuracy of feature location techniques. The former computes the fraction of elements deemed relevant among those reported, while the latter computes the fraction of reported relevant elements among all those deemed relevant. In this section, we present a number of *diff set*-based heuristics, designed to improve both these metrics.

In what follows, let $f$ be a feature of interest, $P$ be a program that implements $f$, $\bar{P}$ be program that lacks $f$, and $\Delta P[\bar{P}]$ be a *diff set* obtained by comparing $P$ to $\bar{P}$.

#### 6.4.1 Improving Precision

Our first goal is to reduce the number of false positive results returned by a feature location technique.

**Filtering**

Following Definitions 8 and 9, all relevant elements reside in a corresponding *diff set*. We use this quality to improve precision of feature location by removing all those results that are not part of the corresponding *diff set*. We call this heuristic Filtering ($F$) and define it below.

**Heuristic 1** (Filtering) Let $S_R$ be a set of results produced by a feature location technique when locating an implementation of a feature $f$. Then a filtered set of results for $f$ is $S_R \cap \Delta P[\bar{P}]$.

For the example in Figure 6.1, when locating the code of the Emacs-style incremental search feature, elements #7, 8, 9 and 11 would be filtered, even if ranked high by a feature location technique, because they contribute to different search features.

The main advantage of Filtering is its ability to improve precision without negatively affecting recall, as it never removes relevant elements. In addition, Filtering operates on the result of feature location and thus is applicable to any feature location technique.
Pruning

When focusing on a specific set of techniques, i.e., the static iterative approaches described in Section 6.2, we can achieve a further improvement in precision if we use diff sets to modify the exploration process as well. We expect that elements implementing a feature are strongly connected: when starting the exploration process from an element in the diff set, unchanged elements, especially those that are scored low by the original feature location approach, are less likely to lead to additional relevant elements, e.g., element #7 in Figure 6.1. When not following such elements, the exploration process collects fewer false positive results.

We thus propose to decrease the score of elements that are not in diff sets proportionally to their original score, as calculated by the original algorithm, so that fewer irrelevant elements fall above the exploration threshold. The decrease in score is controlled by a constant $\gamma_p$, ranging between 0 and 1, whose value is set empirically (see Section 6.5.3). We call this heuristic Pruning ($P$) and define it below.

**Heuristic 2** (Pruning) Let $\gamma_p \in [0..1]$ be a constant. Then, Pruning affects the score calculation as follows:

$$
score(e) \leftarrow \begin{cases} 
    score(e) & \text{if } e \in \Delta P_\bar{P} \\
    score(e) \times (1 - \gamma_p) & \text{otherwise}
\end{cases}
$$

Neither Filtering nor Pruning help find additional relevant elements missed by the original feature location technique: both are designed just to improve its precision. However, unlike Filtering, Pruning can negatively affect recall, preventing the exploration process from reaching relevant results.

### 6.4.2 Improving Recall

Improving recall is at least as essential as improving precision: a technique that always returns an empty set of elements is very “precise”, as it does not produce any false positive results, but is rather useless. We thus propose two heuristics for improving recall: Boosting ($B$) and Boosting with Modifiers ($B_m$).

**Boosting**

We expect that those elements that are contained in a diff set have a higher probability to be relevant to the feature of interest; thus, increasing their score could be effective. However, since most of the elements in a diff set might not be relevant to a feature being identified, e.g., element #10 in the example in Figure 6.1, uniformly setting a high score to all elements in diff set is undesirable. We propose to affect the score of elements proportionally to their original score, using a constant $\gamma_b$ ranging between 0 and 1. The value of $\gamma_b$ is set empirically (see Section 6.5.3). This heuristic is called Boosting ($B$) and is defined below.

**Heuristic 3** (Boosting) Let $\gamma_b \in [0..1]$ be a constant. Then, Boosting modifies the score calculation as follows:

$$
score(e) \leftarrow \begin{cases} 
    score(e) \times (1 + \gamma_b) & \text{if } e \in \Delta P_\bar{P} \\
    score(e) & \text{otherwise}
\end{cases}
$$

If the calculated score is higher than 1, it is set to 1.

For the example in Figure 6.1, consider evaluating relevance of the element #6 w.r.t. the Emacs-style incremental search feature. While its lexical and structural relevance is relatively low, e.g., 0.4, increasing the score of this
element by 30% (corresponding to $\gamma_p=0.3$) will push it above the relevance threshold of, say, 0.5, and thus the feature location algorithm will determine it to be relevant to the feature in question.

**Boosting with Modifiers**

This heuristic attempts to distinguish between different types of features – those that introduce new, previously missing, functionality and those that modify existing features. The Emacs-style incremental search feature described in Section 6.1 is of the first type – it introduces new search capabilities not present beforehand.

The **Boosting with Modifiers ($B_m$)** heuristic is built upon an expectation that new / modified features are mostly implemented by new / modified code elements, respectively. We thus mark each feature as either new or modified using a modifier $M$. We also track the type of change of each element in the diff set – either new or modified, denoted by $mod(e)$. When both the feature and the code element are marked as new or both are marked as modified, the score is doubly boosted.

**Heuristic 4 (Boosting with Modifiers)** Let $M$ be the modifier of $f$ and $\gamma_b \in [0..1]$ be a constant. Then, the score calculation is modified as follows:

\[
\text{score}(e) \left\{ \begin{array}{ll}
\text{score}(e) \times (1+2 \ast \gamma_b) & \text{if } e \in \Delta P[\bar{P}] \land mod(e) = M \\
\text{score}(e) \times (1+\gamma_b) & \text{if } e \in \Delta P[\bar{P}] \land mod(e) \neq M \\
\text{score}(e) & \text{otherwise}
\end{array} \right.
\]

If the calculated score is higher than 1, it is set to 1.

**6.4.3 Combining Approaches**

A reasonable balance between precision and recall is the main challenge of any feature location technique. We hope to achieve improvements in both precision and recall of feature location by leveraging the advantages and mitigating the disadvantages of the proposed heuristics. We thus combine the heuristics, using $B_m$ instead of $B$, into an algorithm referred to as $\mathcal{PB}_m\mathcal{F}$ (Pruning, Boosting with Modifiers, Filtering). We use $B_m$ instead of $B$ as it was shown to be more effective in our experiments (see Section 6.6).

$\mathcal{PB}_m\mathcal{F}$ extends Algorithm 1 by modifying the score calculation (line 5) as specified below:

\[
\text{score}(e) \left\{ \begin{array}{ll}
\text{score}(e) \times (1+2 \ast \gamma_b) & \text{if } e \in \Delta P[\bar{P}] \land mod(e) = M \\
\text{score}(e) \times (1+\gamma_b) & \text{if } e \in \Delta P[\bar{P}] \land mod(e) \neq M \\
\text{score}(e) \times (1-\gamma_p) & \text{otherwise}
\end{array} \right.
\]

If the calculated score is higher than 1, it is set to 1.

$\mathcal{PB}_m\mathcal{F}$ also augments Algorithm 1 by applying Filtering (Heuristic 1) on the set of produced results (line 14).

**6.5 Empirical Evaluation: Experimental Setting**

Our goal is to study the applicability of diff sets for improving the quality of feature location techniques for finding distinguishing features. Specifically, we are interested in evaluating the effectiveness of $\mathcal{PB}_m\mathcal{F}$, as well as the individual diff set-based heuristics proposed in Section 6.4. In this section, we describe our experimental setting.
Table 6.1: Characteristics of projects used in evaluation.

<table>
<thead>
<tr>
<th>No.</th>
<th>Project</th>
<th>NCLOC</th>
<th>methods</th>
<th>fields</th>
<th>Compared variant</th>
<th>diff methods (new : mod.)</th>
<th>diff fields (new : mod.)</th>
<th>diff size</th>
<th>Features (new : mod.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Agilefant 3.0alpha10</td>
<td>23,495</td>
<td>2,595</td>
<td>1,089</td>
<td>2.0.5</td>
<td>(312 : 161)</td>
<td>(124 : 14)</td>
<td>16.59%</td>
<td>5 (1 : 4)</td>
</tr>
<tr>
<td>2</td>
<td>Eclipse 2.1.0</td>
<td>899,077</td>
<td>76,077</td>
<td>37,621</td>
<td>2.0.2</td>
<td>(22,606 : 9,423)</td>
<td>(13,292 : 1,185)</td>
<td>40.90%</td>
<td>3 (1 : 2)</td>
</tr>
<tr>
<td>3</td>
<td>Freemind 0.7.1</td>
<td>12,654</td>
<td>1,345</td>
<td>489</td>
<td>0.6.7</td>
<td>(252 : 97)</td>
<td>(84 : 4)</td>
<td>23.83%</td>
<td>1 (1 : 0)</td>
</tr>
<tr>
<td>4</td>
<td>Freemind 1.0alpha7</td>
<td>59,577</td>
<td>5,535</td>
<td>2,344</td>
<td>1.0alpha7</td>
<td>(349 : 149)</td>
<td>(142 : 5)</td>
<td>7.79%</td>
<td>5 (1 : 4)</td>
</tr>
<tr>
<td>5</td>
<td>Freemind 1.0beta2</td>
<td>62,457</td>
<td>5,820</td>
<td>2,464</td>
<td>1.0beta2</td>
<td>(98 : 87)</td>
<td>(40 : 3)</td>
<td>2.72%</td>
<td>4 (4 : 0)</td>
</tr>
<tr>
<td>6</td>
<td>Freemind 1.0beta5</td>
<td>63,196</td>
<td>5,902</td>
<td>2,495</td>
<td>1.0beta5</td>
<td>(151 : 250)</td>
<td>(66 : 47)</td>
<td>5.28%</td>
<td>1 (1 : 0)</td>
</tr>
<tr>
<td>7</td>
<td>Jbidwatch 1.0</td>
<td>22,942</td>
<td>1,812</td>
<td>1,299</td>
<td>22.942</td>
<td>(296 : 370)</td>
<td>(348 : 65)</td>
<td>34.68%</td>
<td>1 (1 : 0)</td>
</tr>
<tr>
<td>8</td>
<td>JEdit 4.3pre14</td>
<td>104,015</td>
<td>6,253</td>
<td>3,294</td>
<td>4.3pre14</td>
<td>(98 : 63)</td>
<td>(55 : 7)</td>
<td>2.34%</td>
<td>0 (0 : 1)</td>
</tr>
<tr>
<td>9</td>
<td>JEdit 4.3pre15</td>
<td>105,880</td>
<td>6,386</td>
<td>3,348</td>
<td>4.3pre14</td>
<td>(151 : 250)</td>
<td>(66 : 47)</td>
<td>5.28%</td>
<td>1 (1 : 0)</td>
</tr>
<tr>
<td>10</td>
<td>JEdit 4.3pre16</td>
<td>107,547</td>
<td>6,521</td>
<td>3,399</td>
<td>4.3pre16</td>
<td>(197 : 169)</td>
<td>(105 : 50)</td>
<td>5.25%</td>
<td>2 (0 : 2)</td>
</tr>
<tr>
<td>11</td>
<td>Rhino 1.5R6</td>
<td>45,309</td>
<td>2,502</td>
<td>1,974</td>
<td>1.5R6</td>
<td>(73 : 39)</td>
<td>(148 : 0)</td>
<td>17.04%</td>
<td>9 (0 : 9)</td>
</tr>
<tr>
<td>12</td>
<td>StreamRipStar 0.6</td>
<td>12,965</td>
<td>590</td>
<td>936</td>
<td>ini</td>
<td>(73 : 39)</td>
<td>(148 : 0)</td>
<td>17.04%</td>
<td>5 (3 : 2)</td>
</tr>
</tbody>
</table>

### 6.5.1 Subject Software Systems

#### Analyzed Systems

We selected twelve variants of seven open-source Java systems of different sizes and from different domains (see Table 6.1) and analyzed the total of fifty-two features. Some of the selected systems and their corresponding features were previously identified by existing works [134, 129, 43]. These include the Eclipse integrated development environment (line 2 in Table 6.1); the Freemind mind-mapping tool\(^2\) version 0.7.1 (line 3 in Table 6.1); the JBidWatcher tool for bidding, sniping and tracking bids on auction sites\(^3\) (line 7 in Table 6.1); the JEdit text editor\(^4\) (lines 8-10 in Table 6.1), and Rhino JavaScript/ECMAScript interpreter and compiler\(^5\) (line 12 in Table 6.1).

Even though the main focus of our work is on features in different variants of a product family realized via code cloning, releases of a program mimic the qualities of such families when locating distinguishing features. Figure 6.2 sketches the cloning process in a family of related products, in which variants are created by duplicating a specific version and continuing its development independently from the original. For example, products $P_2$ and $P_3$ are created by cloning the existing product $P_1$ at points 2 and 4, respectively. After cloning, both new and existing products continue to involve independently from each other. Furthermore, product $P_2$ itself is cloned at point 5 to create another variant – $P_4$.

![Figure 6.2: Cloned product variants.](http://freemind.sourceforge.net)

\(^2\)http://freemind.sourceforge.net
\(^3\)http://www.jbidwatcher.com
\(^4\)http://www.jedit.org/
\(^5\)http://www.mozilla.org/rhino/
Thus, for our analysis, we focus on versions of a program and use an earlier release to represent a variant that does not contain the feature of interest.

All but the Rhino cases were chosen because they focused on analyzing the new functionality described in the documentation or in online tracking systems as program patches or bug reports, i.e., distinguishing features. We adopted only those cases where we were able to clearly identify a version in which the studied feature was introduced, and had access to an earlier version of software that did not have that feature. The analyzed and the earlier version for each project, as well as the number of features we studied, are specified in the second, sixth and tenth column in Table 6.1, respectively. We also manually classified the studied features into new and modified, based on their description in the corresponding documentation (see the last column of Table 6.1). For example, the Eclipse project has one new and two modified features.

The authors of [43] analyzed Rhino features that correspond to requirements from the ECMA Script specification ECMA-262 edition 3, implemented by version 1.5R6 of the Rhino software. For our study, we adopted those cases that were marked as uncompliant by a subsequent version of the specification, ECMA-262 Edition 5.1. That specification was implemented in version 1.7R3 of Rhino, which we used to produce the diff sets. We assumed that the implementation of uncompliant features underwent major modifications and thus can be treated as distinguishing for Rhino 1.5R6. We classified all Rhino cases as modified (see Table 6.1).

In addition to the previously published case studies, we analyzed the Agilefant web-based backlog management tool6 (line 1 in Table 6.1), the StreamRipStar GUI interface for recording MP3 streaming to a hard drive7 (line 12 in Table 6.1), as well as additional releases of the Freemind software (lines 4-6 in Table 6.1). The classification of the features for these projects is discussed in Section 6.5.1.

For each of the analyzed systems, Table 6.1 lists an approximate size of the program in Non-Comment Lines Of Code (NCLOC), as measured by the open-source tool Cloc8, and the number of its methods and fields, measured using the Metrics plug-in for Eclipse9 (see columns 3, 4 and 5, respectively). The table also shows the size of the corresponding diff sets in terms of the number of new / modified methods and fields (columns 7 and 8). For example, the diff set for the Agilefant project, created by comparing versions 3.0alpha10 and 2.0.5, contains 312 new and 161 modified methods, as well as 124 new and 14 modified fields. The project itself is implemented by 23,495 lines of code and contains 2,595 methods and 1,089 fields. We use these metrics to compute a relative size of a diff set, shown in the second to last column of Table 6.1. The size is calculated as a fraction of diff set elements (methods and fields) out of the total number of program elements and gives an indication of the degree of difference between the analyzed and the previous version. Diff sets are relatively large (32-40%) for Eclipse, JBidwatch and Rhino and relatively small (less than 24%) for the rest of the projects.

**Expected Results**

In order to conduct experiments evaluating the effectiveness of feature location, we have to define the set of expected results. For Eclipse and JEdit projects, we followed the study in [129], where the authors analyzed feature location techniques for detecting Eclipse bug fixes and JEdit patches, as recorded by online tracking systems. The study focused on top ten results reported by ten different techniques. The results were further reviewed by the authors and categorized as “relevant”, “somewhat relevant” and “not relevant”. Following that, the authors used volunteers to evaluate some of the results and showed that in 90% of cases their judgment agreed with that of the volunteers’. Thus, we assume that all results determined as relevant for Eclipse and JEdit by the authors of

---

6 http://www.agilefant.org/
7 http://sourceforge.net/projects/stripster/
8 http://cloc.sourceforge.net/
9 http://metrics.sourceforge.net/
[129] are, in fact, relevant. However, relevant results may not be limited to those reported in the top ten lists. To find those for the Eclipse and JEdit projects, we asked two Java software developers employed by a commercial company to evaluate each feature and refine the list of relevant methods and fields for it. These developers were given the source code, the description of the feature and the id of the corresponding bug fix / patch. They were asked to retrieve the set of changes that correspond to the studied bug fix / patch from the online tracking system, extending the set of relevant results from those reported in [129].

Freemind version 0.7.1 was analyzed by [134], where three independent developers were used to produce a set of expected results. We used the approach in [69] where an element is considered to be relevant if at least two developers said so.

The authors of [43] manually associated methods, fields and types in the Rhino source code with each of the studied feature. We directly reused those in our work.

The Agilefant, Freemind and StreamRipStar projects are managed in git10. We carefully analyzed change histories of each and picked those commit operations which describe additions of exactly one new feature. For example, a commit operation for Freemind with the description “NEW: Purging of old tile images in cache added” corresponds to a feature. We then automatically extracted all elements that were modified by a selected commit operation and considered them to be the expected results. The commit descriptions also helped us categorize features into new or modified: e.g., the above feature is clearly new. In each of these cases, a diff set contains the set of expected results but not only them: we produced diff set by comparing stable release versions of the project to each other and thus they include elements that correspond to several commit operations, as indicated in Table 6.1.

Natural Language Queries

IR-based feature location techniques, e.g., Dora, require a natural language query to determine lexical similarity of the explored elements to the feature of interest. For Eclipse and JEdit projects, those are taken directly from [127]. Queries for Freemind 0.7.1 project are taken from [69]. Natural language queries for Rhino were provided to us by the first author of [43]. Queries describing the features of the Agilefant, Freemind and StreamRipStar projects are taken directly from the commit descriptions while dropping words that are not specific to the feature, like “new” or “added”. For example, the query that corresponds to the commit discussed above is “Purging of old tile images in cache”.

The list of all features that we analyzed, together with their corresponding queries and the expected results is available online [136].

6.5.2 Implementation

For our prototype, we extended two existing feature location techniques, Dora [69] and Suade [132], whose source code was made available to us by their authors. We had no access to the source code of additional feature location techniques and thus did not experiment with them. The choice of tools implies that we consider Java programs whose elements are methods and fields, and whose relationships are method calls and field accesses.

For both tools, we augmented the basic algorithm (Algorithm 1) with multiple combinations of Filtering, Pruning, Boosting and Boosting with Modifiers heuristics presented in Section 6.4, producing the following five versions:

1. $\mathcal{F}$, which uses Filtering only.

10http://git-scm.com/
2. \( \mathcal{PF} \), which extends Filtering with Pruning.

3. \( \mathcal{BF} \), which extends Filtering with Boosting.

4. \( \mathcal{BmF} \), which extends Filtering with Boosting with Modifiers.

5. \( \mathcal{PBmF} \), which extends Filtering with Pruning and Boosting with Modifiers.

For \( \text{Suade} \), we automatically initiated the next iteration of the algorithms with those results of the previous iteration that were ranked above the exploration threshold, instead of doing that manually.

In order to produce diff sets, we extended \text{CHANGEDISTILLER} [54] – a tool which extracts semantic fine-grained source code changes between subsequent revisions of Java classes, based on calculating differences of their abstract syntax trees. The tool detects four types of changes: \textit{Insert}, \textit{Move}, \textit{Update} and \textit{Delete}, and operates on a Java statement level. Since \textit{Dora} operates on a method level and \textit{Suade} – on a method and field level, we aggregated changes inside a method and considered a method as changed (and thus in the diff set) if at least one of its statements was changed. We also explicitly distinguished between newly added and modified elements, as shown in Table 6.1, and ignored changes that occurred in comments or in Java annotations.

### 6.5.3 Methodology and Measures

Following the experimental setting in [69], for each analyzed feature, we initiated \textit{Dora} and \textit{Suade} by choosing one element from the expected results as a seed, and averaged the calculated metrics for all seeds of a feature. We also set each technique to traverse the neighborhood of the seed element three edges away. That is, all elements related to the seed are explored, and those scored above the exploration threshold are further processed to access the elements that are related to them. After three iterations, the exploration process stops.

We used three projects (Freemind v1.0beta2, Freemind v1.0beta5 and StreamRipStar) as a training set for adjusting values of exploration and relevance thresholds for both \textit{Dora} and \textit{Suade}. That is, we ran each of the algorithms while varying the thresholds from 0 to 1, incrementing their values by 0.1. We then picked those values that resulted in the best performance of the algorithms and reused them for all of the remaining projects. Similarly, we trained the algorithms on the same set of projects to fix the values of \( \gamma_p \) and \( \gamma_b \) used by our heuristics to control the necessary degree of Pruning and Boosting. The exact values of the parameters we used are given in Table 6.2.

<table>
<thead>
<tr>
<th>Table 6.2: Configuration Parameters.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_E )</td>
</tr>
<tr>
<td>Dora</td>
</tr>
<tr>
<td>Suade</td>
</tr>
</tbody>
</table>

The remaining projects were used as a control set. We believe that our approach of combining training and control sets works closely to what the user is able to do in a real-life setting, as the set of expected results is usually unknown upfront, and it is not possible to train each technique for each project.

To evaluate the techniques, we used the following metrics:

1. \textit{Expected}: the size of the predetermined expected result.

2. \textit{Reported}: the total number of returned elements.

3. \textit{Correct}: the total number of relevant returned elements.
Table 6.3: Comparing original algorithms (Orig) to algorithms extended with a combination of Filtering (F), Pruning (P), Boosting (B), and Boosting with Modifiers (B_m).

<table>
<thead>
<tr>
<th>No.</th>
<th>Comparison</th>
<th>Cases</th>
<th>Min.(%)</th>
<th>Max.(%)</th>
<th>Avg.(%)</th>
<th>Cases</th>
<th>Min.(%)</th>
<th>Max.(%)</th>
<th>Avg.(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F &gt; Orig (F-measure)</td>
<td>50</td>
<td>1.06</td>
<td>304.75</td>
<td>58.31</td>
<td>51</td>
<td>0.25</td>
<td>320.00</td>
<td>30.58</td>
</tr>
<tr>
<td>2</td>
<td>PF &gt; F (F-measure)</td>
<td>24</td>
<td>0.20</td>
<td>321.57</td>
<td>45.94</td>
<td>13</td>
<td>0.15</td>
<td>26.32</td>
<td>8.47</td>
</tr>
<tr>
<td>3</td>
<td>B &gt; Orig (Recall)</td>
<td>18</td>
<td>1.75</td>
<td>100.00</td>
<td>29.91</td>
<td>25</td>
<td>6.67</td>
<td>372.73</td>
<td>66.39</td>
</tr>
<tr>
<td>4</td>
<td>B_m &gt; Orig (Recall)</td>
<td>20</td>
<td>3.13</td>
<td>109.30</td>
<td>51.73</td>
<td>24</td>
<td>12.50</td>
<td>518.18</td>
<td>103.78</td>
</tr>
<tr>
<td>5</td>
<td>BF &gt; Orig (Recall)</td>
<td>21</td>
<td>1.75</td>
<td>100.00</td>
<td>31.35</td>
<td>33</td>
<td>6.67</td>
<td>372.73</td>
<td>54.68</td>
</tr>
<tr>
<td>6</td>
<td>B_m,F &gt; Orig (Recall)</td>
<td>26</td>
<td>3.13</td>
<td>109.30</td>
<td>44.54</td>
<td>34</td>
<td>12.50</td>
<td>518.18</td>
<td>80.51</td>
</tr>
<tr>
<td>7</td>
<td>PB_m,F &gt; Orig (F-measure)</td>
<td>50</td>
<td>0.01</td>
<td>617.12</td>
<td>99.35</td>
<td>43</td>
<td>7.75</td>
<td>320.00</td>
<td>62.81</td>
</tr>
<tr>
<td>8</td>
<td>PB_m,F &gt; Orig/F (Recall)</td>
<td>24</td>
<td>5.26</td>
<td>100.00</td>
<td>39.62</td>
<td>30</td>
<td>12.50</td>
<td>509.09</td>
<td>79.36</td>
</tr>
<tr>
<td>9</td>
<td>PB_m,F &gt; F (F-measure)</td>
<td>31</td>
<td>0.12</td>
<td>253.24</td>
<td>41.15</td>
<td>30</td>
<td>0.53</td>
<td>209.40</td>
<td>41.45</td>
</tr>
<tr>
<td>10</td>
<td>PB_m,F &lt; Orig (F-measure)</td>
<td>2</td>
<td>5.00</td>
<td>60.26</td>
<td>32.63</td>
<td>9</td>
<td>0.86</td>
<td>18.43</td>
<td>10.15</td>
</tr>
<tr>
<td>11</td>
<td>PB_m,F &lt; Orig (Recall)</td>
<td>9</td>
<td>6.28</td>
<td>33.33</td>
<td>18.47</td>
<td>5</td>
<td>12.50</td>
<td>24.98</td>
<td>17.08</td>
</tr>
</tbody>
</table>

4. **Precision**: the fraction of relevant elements among those reported, calculated as $\text{Correct}_{\text{Reported}}$.  

5. **Recall**: the fraction of all relevant elements reported, calculated as $\text{Correct}_{\text{Expected}}$.  

6. **F-measure**: a harmonized measure combining precision and recall, whose value is high if both precision and recall are high, calculated as $\frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$. This measure is usually used to evaluate the accuracy of a technique as it does not allow trading-off precision for recall and vice versa.

### 6.6 Results

To investigate the impact of diff sets on improving the accuracy of the existing techniques, we ran five versions of the algorithms described in Section 6.5.2 for both Dora and Suade. We analyzed the results of each algorithm separately, distilling the effect of each heuristic on the obtained improvements in F-measure. We also analyzed the improvements in recall achieved by Boosting-based heuristics designed for that purpose: as discussed in Section 6.4, it is more challenging, and thus more “valuable”, to improve F-measure by increasing recall (the fraction of relevant elements found by a technique among those deemed relevant) rather than precision (the fraction of relevant elements among those reported).

Table 6.3 shows the representative comparisons between the different runs, aggregated for all 52 analyzed features across the twelve projects from Table 6.1. For each comparison, we report on the total number of cases in which one technique performed better (worse) than another, as well as the percentage of minimal, maximal and average improvement (decrease). The per project analysis is shown in Figure 6.3. Our results demonstrate that the combination of the proposed heuristics – the $PB_m,F$ variant of the algorithms – achieves the best improvement w.r.t. both the F-measure and recall, when compared to the original approaches. Specifically, these metrics improve by up to 617% (F-measure) and 100% (recall) for Dora, as well as 320% (F-measure) and 509% (recall) for Suade (boldface in Table 6.3). The exact details of each individual run are reported online [136]. In what follows, we present an in-depth analysis of the most interesting cases.
6.6.1 Analyzing Individual Heuristics

Improving Precision

As expected, Filtering improves the F-measure in the majority of cases: 50 for Dora and 51 for Suade (see row 1 of Table 6.3). The improvement is attributed to removing those results that are clearly irrelevant when locating distinguishing features, i.e., those that are not in the diff set. For Dora, the effect of Filtering increases the F-measure between 1% and 304%, with an average improvement of 58%. For Suade, the improvement ranges between 0.25% and 320%, with an average of 30%. Suade’s maximal improvement of 320% is achieved for the Bracket matching enhancements feature of JEdit version 4.3pre6, where the original technique returns 13 results where only one is relevant. Filtering removed the 12 irrelevant elements.

Conclusion 1 Filtering can substantially improve precision of feature location without decreasing recall (as it only operates on the set of returned results). Thus, it is a useful heuristic always applicable to locating distinguishing features.

Pruning further improves Filtering w.r.t. the F-measure only in 24 cases for Dora and 13 cases for Suade, with the average improvement of 45% and 8%, respectively (see row 2 of Table 6.3). Moreover, in some cases, the improvement of the F-measure results from a significant increase in precision that comes together with a smaller decrease in recall. For example, the maximum improvement of 321% in the F-measure for Dora due to Pruning is achieved for the UnifiedTree should ensure file or folder exists feature of Eclipse. For this example, the original algorithm returns 320 elements, after Filtering, of which only 7 are relevant. With Pruning, it returns only 56 elements of which 5.29 are relevant\(^\text{11}\). We thus believe that Pruning is useful when precision of feature location requires significant improvement. As it might negatively affect recall, it should be applied for cases in which precision is the main negative factor affecting the accuracy.

Conclusion 2 Pruning (controlled by an empirically set value of \(\gamma_p\)) is useful for cases where precision of feature location is substantially lower than recall.

Improving Recall

Boosting, with and without Modifiers, was designed to improve recall of feature location techniques. Below, we evaluate the ability of these heuristics to find additional relevant elements compared to the original technique, without decreasing the F-measure.

With Boosting, recall increases in 18 cases for Dora and 25 for Suade, with an average improvement of 29% and 66%, respectively (see row 3 of Table 6.3). Boosting with Modifiers performs even better: recall increases in 20 (for Dora) and 24 (for Suade) cases. Also, with that heuristic, the size of the increase is more substantial, with the average of 51% for Dora and 103% for Suade (see row 4 of Table 6.3). The maximum improvement of recall of 518% is achieved for Suade for the Automatic save file feature of Freemind version 0.7.1, where the extended algorithm finds 11.33 relevant elements out of 24.25 reported, while the original technique finds only 1.83 out of 5. Due to the increase in recall, the F-measure for that case also increases by 214% compared to the original technique.

Furthermore, combining Boosting heuristics with Filtering – a heuristic that does not affect recall but rather efficiently removes false positive results – leads to an increase in the number of cases where recall improves without a decrease in the F-measure. The combined algorithms, \(B \mathcal{F}\) and \(B_{m} \mathcal{F}\), improve recall in 21 and 26 cases for Dora

\(^{11}\)As we average runs from multiple seeds, the number of found elements might contain fractions.
and in 33 and 34 cases for Suade, respectively (see rows 5 and 6 of Table 6.3). Again, $B_mF$ performs better than $BF$ and achieves the maximal improvement in recall of 518% for Suade’s Automatic save file feature. In this case, Filtering helps remove 4.42 false positive results, increasing the overall F-measure by 17% compared to Boosting with Modifiers applied without Filtering.

In the remaining 26 cases for Dora and 18 for Suade, Boosting is either unable to improve recall any further or the improvement comes with a larger reduction of precision, such that the overall F-measure drops. We analyze one of such cases in detail in Section 6.6.2.

**Conclusion 3** Both Boosting heuristics (controlled by an empirically set value of $\gamma_b$) are useful for cases where recall of feature location is substantially lower than precision, with Boosting with Modifiers being more effective than Boosting.

### 6.6.2 Combining the Heuristics: the $\mathcal{PB}_mF$ Algorithm

In practice, we never know whether a feature location technique would result in low precision or low recall, i.e., whether we should apply Pruning or Boosting with Modifiers. Our experiments show that combining these techniques, in addition to Filtering, is effective in the majority of cases: the former positively effects low precision cases, while the latter – those with low recall.

The combined algorithm, $\mathcal{PB}_mF$, improves the F-measure compared to the original techniques in 50 cases for Dora and 43 cases for Suade (see row 7 of Table 6.3). The F-measure increased between 0.01% and 617%, with a 99% average for Dora, and between 7.75% and 320%, with a 62% for Suade. The maximal improvement of 617% for Dora is achieved for the Agilefant project for the query Hide users that are not in the team, where $\mathcal{PB}_mF$ improves precision by finding only 10.5 false positives compared to 116.5 found by the original algorithm. The improvement in precision is achieved mostly due to Pruning; using Filtering alone results in 33.75 false positives.

For this case, $\mathcal{PB}_mF$ does not affect recall – both the improved and the original algorithms are able to find all 4 expected results, achieving 100% recall.

In more than half of the cases, $\mathcal{PB}_mF$ results in a substantial improvement in recall without any decrease in the overall F-measure: 24 cases for Dora, where the improvement ranges between 5% and 100%, with 39% on average, and 30 cases for Suade, where the improvement ranges between 12% and 509%, with 79% on average (see row 8 of Table 6.3). The maximal improvement of 509% w.r.t. recall for Suade is again achieved for the Automatic save file feature of Freemind version 0.7.1, as described in Section 6.6.1. In this example, $\mathcal{PB}_mF$ also outperforms Filtering by itself by 209% and Filtering with Pruning by 232%, due to its ability to find additional relevant elements.

$\mathcal{PB}_mF$ also outperforms the Filtering-only algorithm w.r.t. the F-measure in 31 and 30 cases for Dora and Suade, respectively. While F-measure increases by 41% on average in both cases (see row 9 of Table 6.3), it is notable that $\mathcal{PB}_mF$ increases both precision and recall whereas Filtering is effective in improving precision only. $\mathcal{PB}_mF$ results in improved recall compared to the Filtering-only approach in all case reported in row 8 of Table 6.3.

**Conclusion 4** $\mathcal{PB}_mF$ (controlled by empirically set values of $\gamma_p$ and $\gamma_b$) substantially improves the F-measure in the majority of cases. In more than half of the cases, the improvement is due to the increased recall while in others it is due to the increased precision. No individual heuristic is able to achieve such results and their combination is more beneficial than using each individual heuristic separately.
There are only a few cases where $\mathcal{PB}_mF$ results in a decrease in the F-measure (see row 10 of Table 6.3). For Dora, there are 2 such cases, with a decrease of 5% and 60%, while for Suade there are 9 cases with a decrease ranging between 0.86% and 18%, with the average of 10%. Detailed analysis of the most substantial case – the Remove old threads from mplayer if dead feature of StreamRipStar, in which the F-measure drops from 66% to 26% for Dora, reveals that the original technique was able to find both of the two expected methods, achieving 100% recall, with only two false positive results. $\mathcal{PB}_mF$, however, finds additional 11.5 false positives due to Boosting, significantly dropping precision. We believe that this is caused by the presence of several “generic” terms in the query, corresponding to elements prevalent in the StreamRipStar implementation. These elements are scored close to the relevance threshold, falling below it only because this project was in the training set used to decide on the optimal thresholds for Dora. Running Dora with a “tighter” query would solve the problem; however, query definition is beyond the scope of our work.

Decrease in recall occurs in 9 cases for Dora and 5 for Suade (see row 11 of Table 6.3). The most substantial decrease by 33% is in the case of Dora, for the play streams with mplayer and show the messages in tray feature of StreamRipStar. As discussed earlier, decrease in recall is attributed to the Pruning component of $\mathcal{PB}_mF$: instead of six relevant elements found by the original algorithm, $\mathcal{PB}_mF$ finds four. Yet, in this case, precision grows by 325% (only 11.67 elements are retrieved by $\mathcal{PB}_mF$ vs. 72.33 by the original algorithm). Thus, Pruning is “cost-effective” and the $\mathcal{PB}_mF$ algorithm results in the overall improvement in F-measure by 173% due to combining Pruning and Boosting.

**Conclusion 5** $\mathcal{PB}_mF$ decreases accuracy only in a small number of cases, and the decrease itself is minor.

### 6.6.3 Per-Project Analysis

Figure 6.3 shows absolute improvements in the F-measure and recall, for each of the analyzed projects, aggregated for all of their features. The results are presented separately for Dora and Suade. Values for the original algorithms are shown as filled bars; results achieved by $\mathcal{PB}_mF$ are shown as overlaid frames. For example, the average F-measure for all features of the Agilefant project achieved by Suade is 34% (see the third bar from the left). $\mathcal{PB}_mF$ is able to improve this metric and achieve the F-measure of 58%, as indicated by a solid frame on top of this bar. Recall achieved using the original technique is 74%, which drops to 72% with $\mathcal{PB}_mF$ (the fourth bar).

The figure reaffirms the above conclusions, showing that $\mathcal{PB}_mF$ achieves improvements w.r.t. the F-measure in all of the analyzed projects. Recall is improved in five projects for Dora and nine for Suade. Since we trained our technique on only three of the twelve analyzed projects, these results further validate that the combination of **diff set**-based heuristics employed by $\mathcal{PB}_mF$ is helpful for locating distinguishing features in practice.


6.6.4 Threats to Validity

Threats to external validity are most significant for our work. These arise when the observed results cannot generalize to other case studies. Because we used a limited number of subjects, our results and the calibration parameters might not generalize without an appropriate tuning. However, we attempted to mitigate this threat by using real-life case studies of considerable size from various application domains and by tuning our algorithms on a small training set, using the rest of our cases studies as a control set.

We have prototyped and validated our approach on only two feature detection techniques. Even though they use different scoring schemas, we were able to demonstrate improvement in the accuracy of both. However, the results might not generalize to other tools and approaches.

6.7 Discussion and Related Work

An extensive overview of existing feature location techniques is available in Appendix A. Here, we focus only on those approaches that allow detecting distinguishing features, i.e., consider multiple program variants together.

The idea of leveraging information obtained by comparing variants to each other was recently explored by several research groups [184, 179, 101, 169] – most, following our publication in [139]. Both Ziadi et al. [184] and Linsbauer et al. [101] compare variants to each other, in order to partition the code into segments. Each resulting segment corresponds to a distinct feature present in all products that contribute to the segment and absent in the remaining ones. Xue et al. [179] and Tsuchiya et al. [169] also use the idea of partitioning for reducing the search space for a subsequent application of an IR-based feature location technique within each individual partition. Yoshimura et al. [180] detect variability in a software product line from its change history. The work assumes that each product consists of individual components; components that are frequently modified together when creating one product variant from another are deemed to represent a product line variability point (a.k.a. a feature). Our work differs from those as it does not rely on statistical techniques that require analyzing a sufficient number of variants nor extensive historical data but rather suggest a simple heuristic for using diff sets obtained from comparing two versions. Moreover, we do not introduce a new feature location technique and do not limit ourselves to IR-based approaches, but rather evaluate possible heuristics for improving a variety of existing techniques.

Project documentation together with configuration management systems might help determine code that corresponds to distinguishing features. While our evaluation included several examples where associations between features and code were available, such cases are rare in practice. Several works attempted to analyze commit comments with the goal of discovering features. CVSSearch [25] annotates lines of code with CVS commit messages that correspond to changes involving these lines. Then, given a user query, it returns all lines of code that are annotated by comments containing at least one of the query words. Ratanotayanon et al. [125] propose to extend stored commit data with additional information associating changes with descriptions provided in a commit transaction. Robillard and Dagenais [133] analyze change histories to find elements that frequently change together (a.k.a. change clusters) as part of the same high-level developer’s transaction – several commit operations “linked” to each other by their description. In contrast, our approach does not use historical data nor rely on the availability of meaningful comments and is applicable even when no change tracking is present.
6.8 Summary

As mentioned in Chapter 4, feature location is one of the most common activities undertaken by developers when managing collections of related product variants realized via cloning – in both merge-refactoring and maintaining existing clones scenarios. However, in practice, the accuracy of contemporary feature location techniques still requires improvement and there is no “silver bullet” in feature location – one technique that is clearly superior to others in all possible settings.

In this chapter, we focused on locating distinguishing features – those present in one program variant but absent in another, since locating such features is of high importance in SPL-related scenarios. Our goal was to improve the accuracy of existing feature location techniques, thus leaving the developers the freedom to choose a technique that works best for the specific domain, language and implementation style of their products (see Section A.4 of Appendix A for a discussion on how to choose a technique).

We proposed to use a novel artifact, a diff set, that allowed us to extend heuristics employed by the existing techniques when detecting distinguishing features. We empirically studied the effectiveness of our diff set-based heuristics and showed that in most cases the combination of the heuristics (implemented by the \( PBm,F \) algorithm) enables substantial improvements in the accuracy of feature location, while the cases of reduced accuracy are rare and minor.
Chapter 7

Merge-Refactoring of UML Models

As discussed in Chapter 2, automating approaches for refactoring existing legacy systems into an SPL representation can promote adoption of SPL by industrial organizations. In this chapter, we focus on automating a merge-refactoring process that transforms a set of cloned product variants specified in UML into an annotative SPL representation. We contribute a particular implementation of the merge operator defined in Chapter 4 for UML models with class and statechart diagrams – a common choice for companies that use commercial tools such as IBM Rhapsody\(^1\) to automatically generate fully-functional code from their models.

Our algorithm, merge-in, identifies similar and different elements of the input models using parameterizable compare and match operators, and then constructs a refactored model using a compose operator. We formally specify the properties of these operators and prove that, as long as these properties are satisfied, our refactoring approach is “safe” and semantically correct, i.e., the refactored model is equivalent to the set of original product variants, regardless of a specific implementation of the operators used and the choice of their parameters.

We also argue that varying the merge-in parameters allows producing different syntactic representations of the resulting SPL architecture due to different possible ways to match input model elements. All these representations are semantically equivalent and derive the same set of products. However, not all possible syntactic representations are desirable: depending on the goal of the refactoring, one representation might be preferable to another. For example, a goal of the refactoring can be to highlight the differences between the products at a high-level. Another can be to maximize the comprehensibility of the resulting model by minimizing variability annotations for elements of a certain type. Yet another can be to reduce the size of the resulting refactoring – this might happen if the models are used for execution or code generation rather than human inspection. These different goals induce different SPL representations, and we propose an approach for selecting one that best fits the declaratively specified user intention.

Contributions. This chapter makes the following contributions.

1. We formally specify a merge-refactoring algorithm called merge-in that implements the merge operator defined in Chapter 4. Our algorithm constructs an annotative SPL representation from a set of individual product variants defined with UML class and statechart diagrams.

2. We state sufficient conditions of model compare, match and compose operators that allow application of merge-in and prove that any merge-refactoring produced using merge-in is semantically correct, i.e., derives the set of original input product variants.

\(^1\)http://www-03.ibm.com/software/products/en/ratirhapfami
3. We argue that there can be multiple semantically correct but syntactically different refactoring results that represent the same set of products. Not all such results are desired and explicating the goal of the refactoring can help produce those that better fit the user intention.

4. We propose to capture the goal of the refactoring using a quantitative quality function, comprised of a set of measurable syntactic metrics. This function is used to evaluate the produced refactoring results and to guide the merge-refactoring process towards a desired result.

5. We present an approach for exploring the set of different refactoring results with the goal of identifying a desired one – the one that maximizes the value of a given quality function.

6. We implement a quality-based merge-refactoring framework for UML models containing class and statechart diagrams which relies on the merge-in algorithm. We demonstrate its effectiveness using several example product lines specified in UML, including one contributed by an industrial partner.

This chapter is based on our publications in [138, 143]. We proceed as follows. Section 7.1 describes the running example and illustrates several semantically correct but syntactically different refactoring results. We introduce the model representation that we use in Section 7.2 and give formal foundations of model merging in Section 7.3. Section 7.4 defines the merge-in algorithm and proves its correctness. Our quality-based merge-refactoring framework is described in Section 7.5. In Section 7.6, we present an implementation of the framework and describe our experience applying it to three case studies in Section 7.7. A description of related work is given in Section 7.8. Section 7.9 concludes the chapter.

7.1 Example

Consider the UML model fragments in Figure 7.1(a,b) depicting two representative parts of real-life products developed by an industrial partner (since partner-specific details are confidential, we move the problem into a familiar domain of washing machines). Figure 7.1(a) shows the Controller, Washer and Dryer classes of a washing machine, together with snippets of Controller’s and Dryer’s behaviors specified by UML statechart models. The wtrLevel attribute of Controller is used to specify the desired water level. When the water is filled to that level and heated to 30°C, the washing machine controller notifies Washer that it can start operating and transitions from the Locking to the Washing state. After finishing washing, the controller initiates Dryer and transitions to the Drying state. Dryer operates for 45 minutes and returns the control to the Controller’s statechart (by sending an appropriate signal which is omitted from the picture to avoid clutter). Then, the washing machine is unlocked, and the wash cycle stops. Figure 7.1(b) shows a similar washing machine model which lacks the dryer but has a beeping function indicating the end of the wash cycle by signaling for 1 minute. In addition, in this model, the temp and waterLevel attributes control the desired water temperature and water level, respectively.

These two products have a large degree of similarity and can be refactored into an annotative SPL representation, where duplications are eliminated and variabilities are explicating. We consider only those refactorings that preserve the set of existing products rather than allowing novel feature combinations (e.g., a product with both the dryer and the beeper). Even with this simplification, several choices emerge. For example, the two input models in Figure 7.1(a,b) can be combined as shown in Figure 7.1(c) where the Controller classes of both input models are matched and merged together, while the Dryer and the Beeper classes are unmatched and thus both copied to the produced domain model “as is”, together with their corresponding statecharts. Another choice is shown
in Figure 7.1(d) where these two classes are matched and merged together, producing either a representation in Figure 7.1(g) or in (h). Combining statecharts of Controller classes can also result in two possible representations, as shown in Figure 7.1(e) and (f). That is, there are six possible refactoring options: Figure 7.1(c,e), (c,f), (d,e,g),
(d,e,h), (d,f,g) and (d,f,h).

In each case, the created domain model is annotated by features from the feature model depicted in the top left part of each figure. Since the refactored SPL architecture in our example encapsulates only the original input products, we have just two alternative features representing these products – \( f_A \) and \( f_B \). That is, the propositional formula \( \varphi \) which specifies the relationship between features is \( (f_A \oplus f_B) \). As such, the only two valid feature configurations are \( \{ f_A \} \) and \( \{ f_B \} \). The models in Figure 7.1(a) and Figure 7.1(b) can be derived from any of the refactorings under the configuration \( \{ f_A \} \) and \( \{ f_B \} \), respectively.

Yet, these refactorings are different and without specifying the goal of the refactoring, it is often unclear which one is preferable. For a refactoring that aims at maximizing the comprehensibility of the resulting model, the best representation is the one shown in Figure 7.1(c,e) since it has the least number of classes and states with variable names and the least number of variable statecharts. However, for a refactoring that aims at reducing the size of the result, the best representation is the one in Figure 7.1(d,f,h), as it contains only three classes and six states, compared to the refactoring in Figure 7.1(c,e) which has four classes and nine states.

### 7.2 Model Representation

In this section, we describe our representation of models and model elements and extend the notion of product derivation from Chapter 3 to consider elements of our model representation.

Following XMI principles [116], we define models to be trees of typed elements. Each element has a unique id which identifies it within the model and a role which defines the relationship between the element and its parent. For example, in UML, an element of type Behavior can either play the role of an Entry action or a Do in a state. In addition, a single element can fulfill several roles in a model: a Behavior can be a Do activity of a state and an Effect of a transition at the same time. To allow reusing elements for different roles, we employ a cross-referencing mechanism where an element of type Ref represents the referenced element by carrying its id. Cross-referencing, combined with roles, allows representing labeled graphs using trees: an element can be linked to multiple different elements, each time in a distinct role.

Element types, denoted by \( T \), and roles, denoted by \( R \), are defined by the domain model. For UML, types include Class, State, OpaqueBehavior, etc. Roles include PackagedElement, Subvertex, Effect, etc. If the types Ref and String are not defined by the domain model, we add them to \( T \) as well.

We differ from [116] by representing all element attributes as first-class model elements. That is, an element’s name is represented by a separate model element of role Name and type String. The implication of our representation is that elements’ attributes now have their own ids and thus, an element can have multiple attributes in the same role, e.g., multiple names or Effects for a transition. These qualities are required for defining the product line merge-in operator in Section 7.4. A formal representation of our notations is given by Definition 10 below.

**Definition 10 (Model Element)** A model element \( m \) is a tuple \( \langle m|_{id}, m|_{t}, m|_{r}, m|_{v}, m|_{s} \rangle \), where \( m|_{id} \) is a numeric identifier of the element, \( m|_{t} \in T \) is the element’s type, \( m|_{r} \in R \) is the element’s role, \( m|_{v} \) is the element’s value – either a String or an id of another element (representing a reference), and \( m|_{s} \) is a (nested) list of sub-elements.

Figure 7.2 shows a partial representation of the Controller statechart in Figure 7.1(a), where the state Unlocking, together with its incoming and outgoing transitions, are omitted to avoid clutter. In this figure, sub-elements are represented as an element’s children in the tree.

We refer to types that have no owned properties, such as String or Ref, as atomic. Other types, such as Class, State or Transition, are compound. Elements of atomic and compound types are referred to as atomic
and compound elements, respectively. While atomic elements have values, values of compound elements are determined from the values of their sub-elements. Thus, two compound elements may be equal (i.e., have the same type and role, like elements with ids 3 and 5 in Figure 7.2) but not equivalent, as they might have different sub-elements.

Definition 11 (Equivalence) Given a universe of model elements \( M \), let \( M_1, M_2 \in 2^M \) be distinct sets of elements. \( m_1 \in M_1, m_2 \in M_2 \) are equal, denoted by \( m_1 \cong m_2 \), iff \( m_1|_t = m_2|_t \), \( m_1|_r = m_2|_r \) and \( m_1|_v = m_2|_v \). Equal atomic elements are equivalent. Compound elements are equivalent, denoted by \( m_1 = m_2 \), iff \( m_1 \cong m_2 \), and their corresponding trees of sub-elements are isomorphic wrt. equality.

Definition 12 (Model and Model Equivalence) A set of elements \( M \in 2^M \) is a model iff all elements in \( M \) are connected in a tree structure by the sub-elements relationship, and each \( m \in M \) has a unique id. Models \( M_1 \) and \( M_2 \) are equivalent, denoted by \( M_1 = M_2 \), iff their corresponding root elements are equivalent.

We use the above modeling notations to represent both individual product variants and SPL domain models. Representing domain models is possible due to the ability of our data model to store attributes of a model element as separate entities. Domain model elements can thus have several attributes of the same type fulfilling the same role, which is not allowed by UML, e.g., for names of elements, effects on a transition or do activities of states. We use that property to capture alternative model elements in the generated annotative SPL representations, e.g., the alternative names of a class in Figure 7.1(d): Dryer and Beeper.

We now refine the product derivation concept specified in Definition 6 of Chapter 3 to consider our model representation. For the ease of reading, we give the complete extended definition here, even though only items (d) and (e) are new and items (a)-(c) are copied from Definition 6 as is.

Definition 13 (extension of Definition 6) Let \( \mathcal{PL} = (\mathcal{F}, \mathcal{M}, \mathcal{R}) \) be an SPL architecture and let \( \hat{\mathcal{FM}} \) be one of its feature model configurations. A set of model elements \( \hat{M} \) is derived from the SPL architecture \( \mathcal{PL} \) under the configuration \( \hat{\mathcal{FM}} \), denoted by \( \hat{M} = \Delta(\mathcal{PL}, \hat{\mathcal{FM}}) \), if and only if the following properties hold:
An element belongs to the derived model if and only if this element is annotated by a feature of the feature configuration \( \hat{FM} \) (under which the derivation was performed): \( \forall m \in M, \Delta(m) \in \hat{M} \Leftrightarrow \exists f \in \hat{FM} \cdot (f, m) \in \mathcal{R} \). 

(b) Only one element can be derived from a given domain model element: 
\[ \forall m \in M, \exists! \hat{m} \in \hat{M} \cdot \hat{m} = \Delta(m). \]

(c) Only derived elements are present in the derived model: 
\[ \forall \hat{m} \in \hat{M}, \exists m \in M \cdot \hat{m} = \Delta(m). \]

(d) Each element of the derived model preserves the type/role/value of its corresponding domain model element: 
\[ \forall m \in M, \hat{m} = \Delta(m) \Rightarrow \hat{m} \cong m. \]

(e) Each element of the derived model preserves those sub-elements of its corresponding domain model element that were annotated by the features from \( \hat{FM} \): 
\[ \forall \hat{m} \in \hat{M}, \hat{m} = \Delta(m) \Rightarrow \hat{m} \cong m. \]

### 7.3 Model Merging

We now formalize the properties of model merging [150, 112] – an operation that consists of three steps: compare, match and compose. We specify the minimal set of properties allowing us to use model merging for combining individual product variants while ensuring the semantic correctness of the result.

**Compare** is a heuristic function that calculates the similarity degree, a number between 0 and 1, for each pair of input model elements. It receives models \( M_1, M_2 \) and a set of empirically assigned weights \( \mathcal{W} = \{ w_R \mid R \in \mathbb{R} \} \) which represent the contribution of model sub-elements in role \( R \) to the overall similarity of their owning elements. For example, a similarity degree between two classes is calculated as a weighted sum of the similarity degrees of their sub-elements: names, attributes, operations, etc. Comparing classes in Figure 7.1(a) to each other yields 1, as these classes are identical in the presented model fragment. Comparing Controller classes yields a lower number, e.g., 0.8, as the classes have different owned properties and behaviors.

**Definition 14** (Compare) Let \( M_1, M_2 \in 2^M \) be models. Compare\((M_1, M_2, \mathcal{W})\) is a total function that produces a set of triples \( C \subseteq (M_1 \times M_2 \times [0..1]) \) that satisfy the following properties:

(a) The similarity degree of equal elements is 1: \( (m_1 = m_2) \Rightarrow (m_1, m_2, 1) \in C. \)

(b) The similarity degree of elements having different types or roles is 0:
\[ (m_1 |_t \neq m_2 |_t) \vee (m_1 |_r \neq m_2 |_r) \Rightarrow (m_1, m_2, 0) \in C. \]

(c) While comparing, references are substituted by the elements they refer to:
\[ m_1 |_t = m_2 |_t = \text{Ref} \Rightarrow ((m_1, m_2, x) \in C \Leftrightarrow (M_1[m_1 |_t], M_2[m_2 |_t], x) \in C); \]
\[ m_1 |_t = \text{Ref} \wedge m_2 |_t \neq \text{Ref} \Rightarrow ((m_1, m_2, x) \in C \Leftrightarrow (M_1[m_1 |_t], m_2, x) \in C); \]
\[ m_1 |_t \neq \text{Ref} \wedge m_2 |_t = \text{Ref} \Rightarrow ((m_1, m_2, x) \in C \Leftrightarrow (m_1, M_2[m_2 |_t], x) \in C). \]

(d) \( \text{compare}_{T,R} \) are domain-specific functions, used to calculate the similarity degree between atomic elements of type \( T \) in role \( R \) (e.g., elements’ names): \( m_1 |_t = m_2 |_t = T, m_1 |_r = m_2 |_r = R, T \) is atomic \( \Rightarrow ((m_1, m_2, x) \in C \Leftrightarrow x = \text{compare}_{T,R}(m_1, m_2)). \)
The similarity degree of compound elements is calculated as a weighted sum of their sub-elements’ similarity:

\[ m_1|_t = m_2|_t = T, T \text{ is compound } \Rightarrow ((m_1, m_2, x) \in C \iff x = \sum_{\{R\}} w_R \ast s_R) \]

where \( \{R\} \) is a set of possible roles for sub-elements of \( T \), \( w_R \) is the contribution of sub-elements in role \( R \) to the overall similarity of \( T \), \( \sum_{\{R\}} w_R = 1 \), and \( s_R \) is the calculated similarity between sub-elements of \( m_1 \) and \( m_2 \) in role \( R \).

**Match** is a heuristic function that receives pairs of model elements together with their similarity degree and returns those pairs of elements that are considered similar. **Match** uses empirically assigned similarity thresholds \( S = \{ S_T \mid T \in T \} \) to decide such similarity. For the example in Figure 7.1(a,b), if Dryer and Beeper classes had a calculated similarity degree of 0.75, setting the class similarity threshold to 0.7 would result in matching the classes, while setting it to 0.8 would result in these two classes being unmatched.

**Definition 15** (**Match**) Let \( M_1, M_2 \in 2^\mathcal{M} \) be models and let \( C \) be a set of triples produced by \( \text{compare}(M_1, M_2, \mathcal{W}) \). Then, \( \text{match}(M_1, M_2, C, S) \) is a function that produces a set of pairs \( S \subseteq (M_1 \times M_2) \) that satisfy the following properties:

1. **Each element from \( M_1 \) can be matched with only one element of \( M_2 \), and vice versa:** \( (m_1, m_2) \in S \Rightarrow \forall (m_1', m_2') \in S (m_1'|_{id} = m_1|_{id} \iff m_2'|_{id} = m_2|_{id}) \).
2. **Only identical atomic elements are matched:** \( m_1|_t = m_2|_t = T, T \text{ is atomic } \Rightarrow (m_1, m_2) \in S \Leftrightarrow (m_1, m_2, 1) \in C \).
3. **Compound elements are matched only if their similarity degree exceeds the threshold that is set for their type:** \( m_1|_t = m_2|_t = T, T \text{ is compound } \Rightarrow (m_1, m_2) \in S \Leftrightarrow (m_1, m_2, x) \in C \land x \geq S_T \).
4. **If two elements are matched, their parent elements are matched as well (e.g., it is not possible to match transition guards without matching the owning transitions):** \( (m_1, m_2) \in S \Rightarrow (\exists m_1^p \in M_1, m_2^p \in M_2 \cdot m_1 \in m_1^p|_s \land m_2 \in m_2^p|_s \Rightarrow (m_1^p, m_2^p) \in S) \).
5. **Either root elements of \( M_1 \) and \( M_2 \) are matched with each other, or one of them has no match at all:** \( \neg \exists m_1^p \in M_1 \cdot m_1 \in m_1^p|_s \land \neg \exists m_2^p \in M_2 \cdot m_2 \in m_2^p|_s \Rightarrow ((m_1, m_2) \in S \lor \neg \exists m_1' \in M_1 \cdot (m_1', m_2) \in S) \lor \neg \exists m_2' \in M_2 \cdot (m_1, m_2') \in S) \).

**Compose** is a function that receives two models together with pairs of their matched elements and returns a combined model that contains all elements of the input, such that matched elements are unified and appear in the resulting model only once. While the \( \text{compare} \) and \( \text{match} \) functions rely on heuristically set weights \( \mathcal{W} \) and similarity degrees \( S \), **compose** is not heuristic: its output is uniquely defined by the input set of matched elements. For example, if Dryer and Beeper classes were matched, they would be combined as in Figure 7.1(d). Their corresponding statecharts would be combined either as in Figure 7.1(g) or as in Figure 7.1(h), depending on the matching decision for the internal states. Not matching the Dryer and Beeper classes at all produces the result in Figure 7.1(c).

In what follows, we denote by \( \sigma \) the mapping from an element of an input model to its corresponding element in the merged result, and say that \( \sigma \) transforms an input model element to its corresponding element in the result. We denote by \( \sigma_1^{-1} \) and \( \sigma_2^{-1} \) the reverse mappings from an element in the merged result to its origin in the first and second models, respectively (or \( \emptyset \) if such an element does not exist in one of them). For example, let \( m_1 \),
We now define the merge-in where features represent the original input products and are defined as alternatives to each other.

**Definition 16** (Compose) Let $M_1, M_2 \in 2^M$ be models, $C$ be a set of triples produced by compare($M_1, M_2, W$) and $S$ be a set of pairs produced by match($M_1, M_2, C, S$). Then, compose($M_1, M_2, S$) is a function that produces the merged model $\hat{M}$ and satisfies the following properties:

(a) Matched elements are transformed to the same element in the output model $\hat{M}$:

$$\forall m_1 \in M_1, \exists \hat{m} \in \hat{M} : \hat{m} = \sigma(m_1)$$

(b) Each input model element is transformed to exactly one element of $\hat{M}$:

$$\forall m_1 \in M_1, \exists \hat{m} \in \hat{M} : \hat{m} = \sigma(m_1)$$

(c) Each element of $\hat{M}$ is created from an element of $M_1$ and/or an element of $M_2$. Moreover, no two distinct elements of an input model can be transformed to the same element in the result:

$$\forall \hat{m} \in \hat{M} \cdot (\exists m_1 \in M_1, m_2 \in M_2 : m_1 = \sigma^{-1}(\hat{m})) \lor (\exists m_1 \in M_1, m_2 \in M_2 : m_2 = \sigma^{-1}(\hat{m}))$$

(d) Each element of $\hat{M}$ preserves the type, role and value of its corresponding original elements. (By Definition 14(b) and 15(b), only elements with the same type, role and value can be matched: atomic elements are matched only if identical, while compound elements do not have values.)

$$\forall \hat{m} \in \hat{M} \cdot (\exists m_1 \in M_1, m_2 \in M_2 : \hat{m} = \sigma(m_1)) \Rightarrow \hat{m} \cong m$$

(e) Each element of $\hat{M}$ preserves sub-elements of its corresponding original elements:

$$\forall \hat{m} \in \hat{M}, m^c_1 \in \hat{m} \cdot \hat{m} = \sigma^{-1}(\hat{m}) \Rightarrow \hat{m} \cong m$$

### 7.4 The Merge-In Algorithm

We now define the merge-in refactoring algorithm that puts together distinct product variants and produces an annotative SPL representation. We also prove the correctness of the algorithm, i.e., that the input model variants are the only ones which can be derived from the refactored SPL architecture (Theorem 1). Our merge-in algorithm thus produces minimal behavior-preserving product line refinements, according to the definition in [18].

#### 7.4.1 Merge-In Definition

Constructing an annotative SPL model consists of three steps: creating a domain model, creating a feature model, and specifying annotation relationships between the features and the domain model elements. For creation of a domain model, merge-in relies on model merging, described above. Feature models are created using an approach where features represent the original input products and are defined as alternatives to each other, so only the original products can be derived from the constructed SPL model. Domain model elements are annotated by these features according to the product(s) that contributed them. For the example in Figure 7.1(e), state Drying is annotated by feature $f_A$ while state Beeping is annotated by $f_B$. State Washing is common (it exists in both input models) and thus is annotated by both features.

In our representation, common elements of the constructed domain model, i.e., elements present in all products derived from the SPL representation, are annotated by all features. Variable elements are annotated by some, but not all, features. Annotations of common elements are not shown in Figures 7.1 (c-h) to avoid clutter.
Chapter 7. Merge-Refactoring of UML Models

Definition 17 (Merge-in Construction) $PL' = \langle FM', M', R' \rangle$ is an SPL architecture constructed by merging-in a product $M$ into another SPL architecture $PL$ (denoted by $PL' = PL \oplus_{\mathbb{W}, S} M$), using the rules below:

(a) A new feature $f_M$, representing the merged-in product $M$, is added as an alternative to all existing features: if $FM = \langle F, \varphi \rangle$ then $FM' = \langle F', \varphi' \rangle$, $F' = F \cup \{ f_M | f_M \in F, f_M \notin F \}$, and $\varphi' = (\varphi \lor f_M) \land \bigwedge_{f \in F} \neg(f_M \land f)$.

(b) The domain model is generated by merging the existing domain model with the newly added model $M$: if $C = \text{compare}(M, M, \mathbb{W})$ and $S = \text{match}(M, M, C, S)$ then $M' = \text{compose}(M, M, S)$.

(c) The set of annotation relationships is enhanced by the relationships that annotate elements that originated in $M$ by $f_M$: $R' = \{ (f, \sigma(m)) | f \in F, m \in M, (f, m) \in R \} \cup \{ (f_M, \sigma(m)) | m \in M \}$.

We refer to $PL$ as the original SPL architecture and to $PL'$ as the constructed SPL architecture.

Any input product $M$ can be seen as a “primitive” SPL architecture with only one feature $f_M$, one feature configuration $\{ f_M \}$, and a set of annotations that relate all model elements to that feature. This representation can derive exactly one product — $M$. During the construction, the first product is then implicitly converted into the “primitive” SPL architecture and used as an input to the merge-in algorithm. For example, when combining the two products in Figure 7.1(a,b), we implicitly convert the first one into an SPL architecture and then merge-in it with the second one.

7.4.2 Proof of Correctness

For the correctness proof, we fix $\mathbb{W}$ to be a set of weights used by the compare function and $S$ to be a set of similarity thresholds used by the match functions. Let $PL = \langle FM, M, R \rangle$ be the original SPL architecture and $PL’ = \langle FM’, M’, R’ \rangle$ be an SPL architecture constructed by merging-in the model $M$.

Lemma 2 below shows that any feature configuration that contains only features from the original SPL architecture $PL$ is also a valid feature configuration for the constructed SPL architecture $PL'$, i.e., it complies to the constraints $\varphi'$ defined on the features of $PL'$. For the example in Figure 7.1, this means that the feature configuration $\{ f_A \}$ of the “primitive” product line constructed from the variant in Figure 7.1(a) is also a valid feature configuration for the “extended” SPL architecture, e.g., the one in Figures 7.1(c) and (e).

Lemma 2 Let $FM$ be a subset of $FM$. Then, $FM$ is a feature configuration of $FM$ if and only if it is a feature configuration of $FM'$.

Proof 2 By construction of $\varphi'$ (Definition 17(a)), $\varphi' = (\varphi \lor f_M) \land \bigwedge_{f \in F} \neg(f_M \land f)$. Since $f_M \notin FM$, $\neg(f_M \land f)$ evaluates to true for every $f$, and $\varphi' = (\varphi \lor \text{false}) = \varphi$. Thus, $FM$ respects $\varphi$ if and only if it respects $\varphi'$.

Lemma 3 shows that, under configurations used in Lemma 2, a model derived from $PL$ is equal to the one derived from $PL'$. That is, under the configuration $\{ f_A \}$, the model in Figure 7.1(a) is derived from the SPL architecture in Figures 7.1(c) and (e).

Lemma 3 Let $FM$ be a subset of $FM$. If $FM$ is a feature configuration for $FM$, $M = \Delta(PL, FM)$ and $M' = \Delta(PL', FM)$, then $M = M'$. That is, given a feature configuration that contains only features from $PL$, a set of elements that is generated from $PL$ is equivalent to that generated from $PL'$, under the same configuration.
Proof 3 To prove the lemma, we show that \( f = \Delta(\sigma(\Delta^{-1}(.))) \) is an isomorphism between the elements of \( \hat{M} \) and the elements of \( \hat{M}' \) that respects \( \cong \). That is, we prove the following four statements, showing that \( f \) is an edge-preserving bijection. The construction of the corresponding elements in \( \hat{M} \) and \( \hat{M}' \) is schematically sketched in Figure 7.3.

1. Any element of \( \hat{M} \) has the corresponding equal element in \( \hat{M}' \): \( \forall \hat{m} \in \hat{M}, \exists! \hat{m}' \in \hat{M}' : \hat{m} = f(\hat{m}) \land \hat{m}' \cong \hat{m} \).
   Let \( \hat{m} \in \hat{M} \). By Definition 13(a), this means that there exists an element \( m \in M \), and a feature \( f \in FM \), such that \( (f, m) \in R \) and \( \hat{m} = \Delta(m) \). By Definition 16(b), \( m \) is transformed by compose to an element \( \hat{m}' \in M' \), such that \( \hat{m}' = \sigma(m) \). By Definition 17(c), this element is annotated by the same feature as \( m \): \( (f, \sigma(m)) \in R' \). Thus, \( \Delta(\sigma(m)) \in \hat{M} \) by Definition 13(a). Since \( \hat{m} \) is derived from \( m \), \( m = \Delta^{-1}(\hat{m}) \). It follows that \( \Delta(\sigma(\Delta^{-1}(\hat{m}))) \in \hat{M}' \). Let’s denote that element by \( \hat{m}' \). There exists only one such element by Definition 13(b,c) and 16(b), \( \hat{m}' \equiv \hat{m} \) by Definition 13(d) and 16(d).

2. Any element of \( \hat{M}' \) has the corresponding equal element in \( \hat{M} \): \( \forall \hat{m}' \in \hat{M}', \exists! \hat{m} \in \hat{M} : \hat{m}' = f(\hat{m}) \land \hat{m} \cong \hat{m}' \).
   Let \( \hat{m}' \in \hat{M}' \). By Definition 13(a), this means that there exists an element \( \hat{m} \in \hat{M} \), and a feature \( f \in FM \), such that \( (f, \hat{m}') \in R' \) and \( \hat{m}' = \Delta(\hat{m}) \). By Definition 16(c), there are three possible cases: (1) \( \sigma_1^{-1}(\hat{m}') \in M \), \( \sigma_1^{-1}(\hat{m}') = 0 \); (2) \( \sigma_2^{-1}(\hat{m}') \in M \); (3) \( \sigma_2^{-1}(\hat{m}') \in M \).

   For cases (1) and (3), \( (f, \hat{m}') \in R' \) implies that \( (f, \sigma_1^{-1}(\hat{m}')) \in R \) by Definition 17(c), and thus, \( \Delta(\sigma_1^{-1}(\hat{m}')) \in \hat{M} \) by Definition 13(a). Let’s denote this element by \( \hat{m} \). It is easy to see that \( f(\hat{m}) = \hat{m}' \) (that is \( \Delta(\sigma(\Delta^{-1}(\hat{m}))) = \hat{m}' \)). There exists only one such element \( \hat{m} \) by Definition 13(b,c) and 16(c). \( \hat{m} \equiv \hat{m}' \) by Definition 13(d) and 16(d).

   For case (2), \( \sigma_1^{-1}(\hat{m}') = 0 \) implies by Definition 17(c), that \( \hat{m}' \) is annotated by \( j_M \), and, since \( j_M \notin FM \), \( \Delta(\hat{m}') \notin \hat{M}' \), which, together with \( \hat{m}' = \Delta(\hat{m}') \), creates a contradiction to \( \hat{m}' \in \hat{M}' \).

3. Any sub-element of \( \hat{m} \) has the corresponding sub-element in \( f(\hat{m}) \): \( \forall \hat{m} \in \hat{M} : (\forall \hat{m}' \in \hat{M} \setminus M \land \hat{m}' \equiv \hat{m} \Rightarrow f(\hat{m}') \in f(\hat{m}) \setminus M) \).
   Since \( \hat{m}' \in \hat{m}' \), by Definition 13(a,e), there exist elements \( m, m' \in M \), and features \( f, f' \in FM \), such that \( (f, m) \in R \), \( (f', m') \in R \), \( \hat{m} = \Delta(m) \), \( \hat{m}' = \Delta(m') \) and \( m' \in M \) (it is also possible that \( f = f' \)). By Definition 16(b,e), \( \sigma(m') \in \sigma(m) \). By Definition 17(c), \( (f, \sigma(m)) \in R' \) and \( (f', \sigma(m')) \in R' \), which, by Definition 13(a,e), implies that \( \Delta(\sigma(m')) \in \Delta(\sigma(m)) \). Since \( \Delta(\sigma(m')) = \Delta^{-1}(\hat{m}') \) and \( m = \Delta^{-1}(\hat{m}) \), \( f(\hat{m}') \in f(\hat{m}) \setminus M \), as desired.

4. Any sub-element of \( \hat{m}' \) has the corresponding sub-element in \( \hat{m} \): \( \forall \hat{m}' \in \hat{M}' : \forall \hat{m} \in \hat{M} : (\forall \hat{m} \in \hat{M} \setminus M \land \hat{m}' \equiv \hat{m} \Rightarrow \exists! \hat{m} \land \hat{m}' \in \hat{M} \) \).
   Let \( \hat{m}' \in \hat{M}' \) be elements such that \( \hat{m}' \in \hat{m}' \). By Definition 13(a,e), there exist elements \( \hat{m}' \in \hat{M}' \), and features \( f, f' \in FM \), such that \( (f, \hat{m}') \in R' \), \( (f', \hat{m}') \in R' \), \( \hat{m}' = \Delta(m) \), \( \hat{m}' = \Delta(m') \) and \( m' \in M \).
it is also possible that \(f = f^c\). Similarly to case 2, \(\sigma^{-1}_1(\tilde{m}') \neq \emptyset\) and \(\sigma^{-1}_1(\tilde{m}^e) \neq \emptyset\). By Definition 16(e), either \(\sigma^{-1}_1(\tilde{m}^e) \in \sigma^{-1}_1(\tilde{m}')|_s\) or there exist \(m_1, m_2 \in M\), such that \(\sigma^{-1}_1(\tilde{m}^e)\) is matched with \(m_1\), \(\sigma^{-1}_1(\tilde{m}')\) is matched with \(m_2\), and \(m_1 \in m_2|_s\). The latter case is impossible by Definition 15(a,d,e): if \(\sigma^{-1}_1(\tilde{m}^e)\) has a parent in \(M\), it should be matched with \(m_2\) and, thus, should be \(\sigma^{-1}_1(\tilde{m}')\), implying, again, that \(\sigma^{-1}_1(\tilde{m}^e) \in \sigma^{-1}_1(\tilde{m}')|_s\).

Otherwise, \(\sigma^{-1}_1(\tilde{m}^e)\) has no parent in \(M\), which means that \(\sigma^{-1}_1(\tilde{m}')\) is a root element in \(M\) and \(\sigma^{-1}_1(\tilde{m}')\) belongs to its sub-tree. By Definition 15(e), \(m_2\) cannot be a root element, as the root of \(M\) is not matched, while \(m_2\) is matched with \(\sigma^{-1}_1(\tilde{m}')\). By recursively applying Definition 15(d), this means that some parent of \(m_2\) is matched with \(\sigma^{-1}_1(\tilde{m}')\) which, by Definition 15(a), creates a contradiction to the fact that \(\sigma^{-1}_1(\tilde{m}')\) is matched with \(m_1\) that is a sub-element of \(m_2\). We conclude that \(\sigma^{-1}_1(\tilde{m}^e) \in \sigma^{-1}_1(\tilde{m}')|_s\).

For the former case, since \((f, \tilde{m}') \in \mathcal{R}', (f^c, \tilde{m}^e) \in \mathcal{R}'\), by Definition 17(c), \((f, \sigma^{-1}_1(\tilde{m}')) \in \mathcal{R}, (f^c, \sigma^{-1}_1(\tilde{m}^e)) \in \mathcal{R}\) and thus, by Definition 13(a,e), \(\Delta(\sigma^{-1}_1(\tilde{m}^e))) \in \Delta(\sigma^{-1}_1(\tilde{m}'))|_s\). Let’s denote these elements by \(\tilde{m}^e\) and \(\tilde{m}\), respectively. \(f(\tilde{m}^e) = \Delta(\tilde{m}^e) = \tilde{m}^e\) and \(f(\tilde{m}) = \Delta(\tilde{m}) = \tilde{m}\), implies \(\tilde{m}^e \in \tilde{m}|_s\), as desired.

The above lemma implies that our construction preserves the set of products of the original SPL architecture: the set of models derived from \(\mathcal{P}L\) can still be derived from \(\mathcal{P}L'\), as shown by the following corollary.

**Corollary 1 (Monotonicity)** Let \(\{\mathcal{P}L\}\) denote the set of all models derived from an SPL architecture \(\mathcal{P}L\). That is, \(\{\mathcal{P}L\} = \{\Delta(\mathcal{P}L, \widehat{\mathcal{F}M}) \mid \widehat{\mathcal{F}M} is a feature configuration of \(\mathcal{F}M\)\}. Then, the set of models derived from \(\mathcal{P}L\) can be derived from \(\mathcal{P}L'\) as well: \(\{\mathcal{P}L\} \subseteq \{\mathcal{P}L'\}\).

**Proof 4** For each \(\tilde{M} \in \{\mathcal{P}L\}\), there exists a configuration \(\widehat{\mathcal{F}M}\), such that \(\tilde{M} = \Delta(\mathcal{P}L, \widehat{\mathcal{F}M})\). By Lemmas 2 and 3, \(\tilde{M} = \Delta(\mathcal{P}L, \widehat{\mathcal{F}M}) = \mathcal{P}L'\). Thus, \(\tilde{M} \in \{\mathcal{P}L'\}\).

We now show that model \(M\) which we merge-in into the original SPL architecture \(\mathcal{P}L\) can be derived from the constructed SPL architecture \(\mathcal{P}L'\). That is, when we merge-in the model in Figure 7.1(b) into the “primitive” SPL architecture constructed from the model in Figure 7.1(a), we can derive the merged-in model back from the created SPL architecture, e.g., the one in Figures 7.1(c) and (e).

Since \(f_M\) is the feature that annotates elements of the merged-in model \(f_B\) in our example), we first show that \(\{f_M\}\) is a valid feature configuration (Lemma 4). Then, Lemma 5 shows that the original model \(M\) is derived from the constructed SPL architecture \(\mathcal{P}L'\) under that configuration.

**Lemma 4** \(\{f_M\}\) is a feature configuration for \(\mathcal{P}L'\).

**Proof 5** By construction of \(\mathcal{F}M'\) (Definition 17(a)), \(f_M \in \mathcal{F}'\). We now show that \(\{f_M\}\) respects \(\varphi' = (\varphi \lor f_M) \land \bigwedge_{f \in \mathcal{F}} \neg(f_M \land f)\). Since \(\varphi \not\in \{f_M\}\) for any \(f \in \mathcal{F}\), \(\neg(f_M \land f)\) evaluates to true for every \(f \in \mathcal{F}\). Since \(f_M = \text{true}\), \(\varphi \lor f_M\) also evaluated to true. It follows that \(\{f_M\}\) respects \(\varphi'\) and is a feature configuration for \(\mathcal{P}L'\).

**Lemma 5** Let \(\{f_M\}\) be a feature configuration. Then, a model that is derived from \(\mathcal{P}L'\) under that configuration is equivalent to \(M\). That is, \(M = \Delta(\mathcal{P}L', \{f_M\})\). The proof of this lemma, similarly to the proof of Lemma 3, shows that \(f = \sigma(\Delta(.))\) is an isomorphism between the elements of \(M\) and the elements of \(\tilde{M}\), and is omitted.

Finally, Theorem 1 shows that our merge-in operator is semantically correct: the set of product models that are derived from the constructed SPL architecture \(\mathcal{P}L'\) is equal to the set of models that are derived from the original SPL architecture \(\mathcal{P}L\), in addition to the merged-in model \(M\).

**Theorem 1 (Correctness)** \(\{\mathcal{P}L'\} = \{\mathcal{P}L\} \cup \{M\}\).
Chapter 7. Merge-Refactoring of UML Models

Proof 6 We first prove that \( \lfloor PL' \rfloor \subseteq \lfloor PL \rfloor \cup \{ M \} \). Let \( \hat{M} \in \lfloor PL' \rfloor \) be a model derived from \( PL' \). Then there exists a feature configuration \( \hat{F}_M \), such that \( \hat{M} = \Delta(PL', \hat{F}_M) \). Let \( f_M \) be in \( FM' \backslash FM \).

1. If \( f_M \not\in \hat{F}_M' \), then \( \hat{F}_M' \subseteq FM' \). Thus, by Lemma 3, \( \hat{M} = \Delta(PL', \hat{F}_M') \), which implies that \( \hat{M} \in \lfloor PL \rfloor \).

2. If \( f_M \in \hat{F}_M' \), then, by construction of \( FM' \) (Definition 17(a)), \( \hat{F}_M' = \{ f_M \} \). By Lemma 5, \( M = \Delta(PL', \{ f_M \}) \). Thus, by Lemma 1 in Chapter 3, \( \hat{M} \in \lfloor PL \rfloor \).

We now show that \( \lfloor PL \rfloor \cup \{ M \} \subseteq \lfloor PL' \rfloor \) by Corollary 1. By the construction of \( FM' \) (Definition 17(a)), \( f_M \in FM' \). Thus, by Lemmas 4 and 5, \( \{ f_M \} \) is a valid feature configuration for \( PL' \) and \( M = \Delta(PL', \{ f_M \}) \), which implies that \( M \in \lfloor PL' \rfloor \).

For the example in Figure 7.1 this means that the models in Figures 7.1(a) and (b), and only they, can be derived from any of the constructed SPL representations.

7.5 Quality-Based Merge-Refactoring Framework

Even though all refactorings produced by the merge-in algorithm are semantically equivalent and correct, as they all derive the exact set of original input models, varying the compare weights \( W \) and match similarity thresholds \( S \) can produce substantially different syntactic results. For example, the two product variants in Figures 7.1(a,b) can be combined as shown in Figure 7.1(c,e) or as in Figure 7.1(d,f) where a lower class similarity threshold results in \( \text{Dryer} \) and \( \text{Beeper} \) classes being matched and combined. Likewise, decreasing the weight of the name similarity between states while increasing the weight of the similarity of their corresponding incoming and outgoing transitions could, for example, result in a higher similarity degree between the \( \text{Spinning} \) and \( \text{Beeping} \) states in Figures 7.1(a,b), as their incoming and outgoing transitions are quite similar. This can subsequently lead to matching and combining these states, as in Figure 7.1(h), rather than representing them separately, as in Figure 7.1(g).

Depending on the goal of the refactoring, one representation can be preferred to another. The main objectives of our quality-based product line merge-refactoring framework are thus to (1) allow the user to explicate the goal of the refactoring process and (2) drive the refactoring process towards the result that best fits the user intention. We depict our approach in Figure 7.4 and describe it below.

7.5.1 Explicating The Refactoring Goal

We explicitly capture the goal of the refactoring using a quantitative quality function. Since in our work we focus on syntactic properties of merge-refactorings, the quality function is built from a set of metrics – syntactically measurable indicators representing specific refactoring objectives (see Figure 7.4). Typically, such metrics can assess the size of the resulting model, determine the degree of object coupling, cohesion in methods, weighted number of methods per class and more [107]. The metrics can be reused across different organizations and domains. Each concrete quality function \( Q \) assigns weights \( q_1 \ldots q_m \) to different metrics \( v_1 \ldots v_m \), indicating their “importance” in the context of a specific application domain and allows the user to specify the desired outcome of the refactoring in an explicit manner.

More formally, given an SPL architecture \( PL \), the quality function returns a number between 0 and 1, representing \( PL \)'s “quality”:

\[
\text{quality}(PL, V, Q) = \sum_{i=1}^{n} q_i \ast v_i(PL),
\]
where $\mathcal{V} = v_1, \ldots, v_n$ is a set of measurable metrics that, given $\mathcal{P} \mathcal{L}$, produce a number between 0 and 1, and $\mathcal{Q} = q_1, \ldots, q_n$ is a set of metrics’ weights.

**Examples of Possible Quality Functions.** We discuss two specific quality functions. The goal of the first one, $Q_1$, is to minimize the size of the resulting model. Since we assume that there is a large degree of similarity between input models that represent related products of a product line, aiming to reduce the total number of elements in the result leads to a reduction of duplications which, in turn, helps avoid repeating modifications for all variants.

We define our notion of model size using the number of classes, attributes, states and transitions. Specifically, the metrics $v_1$-$v_4$ listed in the first four rows of Table 7.1 measure the overall reduction of the size of the produced model when compared to the inputs. To construct $Q_1$, we assign equal weights to these metrics, considering them equally important, as specified in the second to last column of Table 7.1. $Q_1$ prefers models that are as compact as possible, e.g., the refactoring in Figure 7.1(d, f, h).

Our second goal is to produce refactorings that are the easiest for the user to comprehend. The work of [51, 33] makes an argument that an increase in the size of UML models (specifically, the number of classes, aggregations, states and transitions) leads to an increase of cognitive complexity. The authors validate this claim using controlled experiments involving human subjects. However, neither these experiments nor our early work [137] considered
variability aspects of the annotative SPL architecture representations. For example, while they minimize the size of the model, both possible merges of the Dryer and the Beep classes in Figures 7.1(g) and (h) contain 66% and 50% variable states (i.e., states annotated by features), respectively. The merge of the Controller classes in Figure 7.1(f) contains 50% variable states as well.

We believe that the higher the number of common elements in a merged model is, the easier it is to understand. We thus define a second quality function, $Q_2$, to combine size minimization with encouraging those refactorings which result in models with a high degree of commonalities: classes with a significant number of common attributes and statecharts with a significant number of common states and transitions. The metrics $v_5$-$v_7$ of Table 7.1 are designed for that purpose. They are calculated by counting the percentage of common sub-elements for a certain element in the model, i.e., those sub-elements that are annotated by all product line features. To achieve a reasonable degree of merging while discouraging too much variability, $Q_2$ gives the combination of four size-based metrics $v_1$-$v_4$ and the combination of three variability-based metrics $v_5$-$v_7$ equal importance (see the last column of Table 7.1). This quality function prefers the refactoring in Figure 7.1(c,e).

We use both $Q_1$ and $Q_2$ to evaluate refactorings of our case-study models in Section 7.7.

### 7.5.2 Constructing the desired refactorings

Since a quality function captures the criteria that are to be used when performing the merge-refactoring process, it could also be used to guide the process towards the desired result. As stated earlier, refactorings produced by the merge-in algorithm differ by the way input model elements are matched and composed, which is controlled by the merge-in configuration parameters – compare weights $W$ and match similarity thresholds $S$. Modifying these parameters, e.g., increasing weight of state name similarities during compare, can result in the refactoring shown in Figure 7.1(e). Instead, if we give more weight to the structural similarity of states, i.e., their distance to the initial and the final states and, recursively, the similarity of their neighbors [112], we get the result in Figure 7.1(f).

Obviously, merge-in parameters cannot be decided universally because their values depend on the nature of the refactored product line and the objective of the quality function. It is also unreasonable to assume that the user can set and adjust these parameters manually. Moreover, generating all possible refactorings and evaluating them based on the given quality function is not a feasible approach as it does not scale well.

We thus need a systematic way for identifying those values of merge-in parameters that result in an optimal refactoring w.r.t. the given quality function $Q$. In our work, we propose doing so by treating parameter selection as a classical optimization problem [117], using the chosen quality function as an objective function for an optimization technique. The process (1) uses an optimization heuristic to set values of merge-in parameters, (2) produces the corresponding refactoring, subject to these parameters, (3) evaluates it using $Q$, and repeats until a result of the desired quality is reached (or a certain fixed number of iterations is performed). That is, different refactorings are generated by the merge-in algorithm based on the values of compare weights and similarity thresholds that are set using an optimization algorithm aimed to maximize the value of $Q$. Only the resulting “optimal” refactoring is returned to the user. The overall efficiency of the approach is as good as the chosen optimization algorithm because the latter selects the values of parameters for the next iteration.

### 7.6 Implementation

In this section, we describe our implementation of the merge-in algorithm, used as a foundation of the merge-refactoring framework, as well as our approach for setting the merge-in parameters. We focus our work on systems
CHAPTER 7. \textsc{Merge-Refactoring} of UML Models

represented and implemented with UML models containing class and statechart diagrams – a common choice in automotive, aerospace & defense, and consumer electronics domains, where such models are often used for full behavioral code generation (e.g., using IBM Rhapsody\textsuperscript{2}).

The core part of the \textit{merge-in} algorithm is the \textit{compare} function which receives two UML elements of the same type and returns their similarity degree – a number between 0 and 1. To implement \textit{compare}, we started with existing comparison algorithms for UML classes \cite{178, 85} and statecharts \cite{112}. These algorithms calculate the similarity degree recursively, using formulas that assign empirically defined weights to similarity degrees of appropriately chosen sub-elements.

None of the existing algorithms combined information obtained by analyzing both structural and behavior models together: comparing classes did not take into account information about similarity of their corresponding statecharts. We thus extended class comparison by considering behavior information, obtained by comparing statecharts to each other, and combining it with structural information by giving them equal weights. We also extended the statechart comparison algorithm proposed in \cite{112} to consider state \textit{entry} and \textit{exit} actions, state \textit{do} activities and actions on transitions, as those were used in the real-life model provided by the industrial partner.

Based on element similarity degrees generated by \textit{compare}, our implementation of \textit{match} “greedily” selects similar elements that are above a given threshold. \textit{Compose} further puts together elements deemed similar while explicating variabilities. We use the \textit{union-merge} \cite{150} approach to implement the \textit{compose} function. It unifies matched elements and copies unmatched elements “as is” to the result. Our \textit{compose} implementation is an adaptation of the static union merge of TRe\textit{Mer}+\textsuperscript{3}, extended to deal with annotations of domain model elements by features. We use IBM Rational Software Architect\textsuperscript{4} (RSA) as our modeling environment, allowing us to reuse existing Java-based algorithms. Rhapsody models supplied by our industrial partner were first exported into UML 2.0 XMI format and then imported into RSA.

For adjusting \textit{merge-in} parameters, we have implemented a version of the local search optimization technique \cite{117} where the space of possible refactorings is explored by changing one parameter value at a time (hill-climbing). After evaluating the resulting refactoring and adjusting this value, we move to the next one, until all values are set. While this algorithm can miss the best result (global maximum) because it does not revisit the decisions that were already made, it is shown to be quite effective in practice for finding a “good” result (local maximum) in an efficient manner. We demonstrate the effect of adjusting the class \textit{similarity threshold} in Section 7.7.

We \textit{merge-in} the most similar models first: similarity degrees of all inputs – models of individual products or already constructed intermediate product lines – are evaluated, and those with the highest similarity degrees are combined first. Intuitively, merging more similar models first helps decrease the size and the number of variable elements in the result.

During our experiments, we noted that different values of \textit{compare weights} and \textit{similarity thresholds} can produce the same refactoring and thus the same \textit{quality} measurements. Since our goal is to maximize a given \textit{quality} function, any of the assignments that produce the desired result is appropriate and either combination is suitable.

\footnotesize
\textsuperscript{2}http://www-01.ibm.com/software/awdtools/rhapsody/
\textsuperscript{3}http://se.cs.toronto.edu/index.php/TRe\textit{Mer}+
\textsuperscript{4}http://www-01.ibm.com/software/awdtools/swarchitect/
7.7 Experience

In this section, we report on our experience applying the quality-based merge-refactorings. Our goal is to validate the feasibility of the approach for UML models in the embedded domain. In particular, we are interested in demonstrating the applicability and effectiveness of the proposed methodology for adjusting merge-in parameters for realistic models containing UML class and statechart diagrams, based on a given quality function. In what follows, we describe our subject product lines and present our results.

7.7.1 Subjects

We applied our refactoring approach to three sets of related products. The first is the Washing Machine example, built by us to mimic a partner’s model and to highlight its characteristics. A snippet of this example is presented in Figure 7.1 and the full version is available in Appendix B.1. The Washing Machine product line contains three different products, with a high degree of overlap in the set of classes comprising them. Specifically, each product has six classes, out of which three are identical across products (Motor, Faucet and Detergent Supplier), two are similar to each other in all three products (Controller and Washer), and one class in each product carries a unique product-specific functionality (either Dryer, Beeper or Timer). Also, statecharts of similar classes have similar structures.

The second example, Microwave Oven, has been introduced by Gomaa in [63] and is available in Appendix B.2. It includes four different, although very similar, variants of the timer control class and their corresponding statecharts.

The final example comes from the Consumer Electronics (CE) space, contributed by an industrial partner. Here, we focus on seven behavior-intensive product components which together contain 45 classes, 104 attributes, 448 states and 944 transitions. The number of classes implementing each component ranges between 2 and 14. The number of statecharts in each component ranges between 1 and 3, with the number of states and transitions for a statechart ranging between 20 and 66 states, and 31 and 81 transitions, respectively. Of the seven components, three have a similar structure and a similar set of elements; another pair of components also contains elements that are similar to each other (but less similar to the components of the first cluster), and the remaining two components are not similar to the rest.

Due to verbosity of UML models, we limit the presentation here to the statistical data about the case studies. The complete models for the first two examples are available in Appendix B. Since we cannot share details of the CE model, we built our first example, the Washing Machine, to be similar.

7.7.2 Results

To evaluate the effectiveness of our quality-based merge-refactoring approach, we analyzed different refactorings produced by varying compare weights and similarity thresholds, and evaluated them using quality functions $Q_1$ and $Q_2$ introduced in Section 7.5. As a starting point, we used empirically determined weights specified in [178, 85, 112]. We updated the weights to combine structural and behavior information when comparing classes and to take into account additional statechart elements, as described in Section 7.6. For the similarity thresholds, we started with the assumption that elements with a similarity degree lower than 0.5 are significantly different and should not be combined. For statecharts, we refined these estimates using the thresholds empirically determined in [112]. The values of weights and thresholds that we used are summarized in Tables 7.2 and 7.3, respectively. These can be used as a starting point for tuning the values in any future applications of our approach, unless additional
domain specific information that can help set up the values is provided.

Table 7.2: Weights $W$ Used by Compare [85, 112, 178].

<table>
<thead>
<tr>
<th>Element (Class)</th>
<th>Weight</th>
<th>Element (Statechart)</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_\text{Name}$</td>
<td>0.2</td>
<td>$w_\text{StateName}$</td>
<td>0.2</td>
</tr>
<tr>
<td>$w_\text{Attribute}$</td>
<td>0.1</td>
<td>$w_\text{StateType}$</td>
<td>0.05</td>
</tr>
<tr>
<td>$w_\text{Operation}$</td>
<td>0.1</td>
<td>$w_\text{StateDepth}$</td>
<td>0.1</td>
</tr>
<tr>
<td>$w_\text{GenTarget}$</td>
<td>0.1</td>
<td>$w_\text{StateActions}$</td>
<td>0.3</td>
</tr>
<tr>
<td>$w_\text{Behavior}$</td>
<td>0.5</td>
<td>$w_\text{Transition}$</td>
<td>0.35</td>
</tr>
</tbody>
</table>

Table 7.3: Thresholds $S$ Used by Match.

<table>
<thead>
<tr>
<th>Element</th>
<th>Threshold</th>
<th>Element</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_\text{Class}$</td>
<td>0.6</td>
<td>$S_\text{State}$</td>
<td>0.8</td>
</tr>
<tr>
<td>$S_\text{Attribute}$</td>
<td>0.5</td>
<td>$S_\text{Transition}$</td>
<td>0.5</td>
</tr>
<tr>
<td>$S_\text{Operation}$</td>
<td>0.5</td>
<td>$S_\text{OpaqueBehavior}$</td>
<td>1</td>
</tr>
<tr>
<td>$S_\text{Statechart}$</td>
<td>0.8</td>
<td>$S_\text{String}$</td>
<td>1</td>
</tr>
</tbody>
</table>

For illustration purposes, in this section we vary the class similarity threshold between 0.4 and 1, iteratively incrementing its value by 0.01, and evaluate the produced results using our quality functions. Table 7.4 presents the total number of elements as well as the number of variable elements of each type in the resulting refactoring. To save space, we show only distinct refactorings, omitting those that are equivalent to the presented ones. For example, in the Washing Machine case, all refactorings produced with class similarity thresholds between 0.4 and 0.7 are identical, and we only show the latest. In addition, the orig column reports the total number of input elements for each of the case studies. It is used to compare the result of the refactoring to the original models and to normalize the collected metrics during quality computation. A full description of the refactored SPL models that were produced for each step of the first two case-studies is available in Appendix B.

The results demonstrate that increasing the value of the class similarity threshold results in decreasing the value of $Q_1$ in all case studies because this function prefers refactorings that are as compact as possible: as the class similarity threshold increases, fewer classes are matched and composed, and the number of elements in the result grows. $Q_2$, however, does not exemplify such linear behavior because it balances the reduction in size with the goal of composing only those elements that are indeed similar. For example, when refactoring the Washing Machine, the result preferred by $Q_1$ is obtained by setting the class similarity threshold to 0.7 or lower, which causes merging of as many classes as possible, including those that are dissimilar (e.g., the one in Figure 7.1(d)). This produces state machines with a large percentage of variable states and transitions. $Q_2$ prefers the solution produced when the similarity threshold is set to 0.78, which combines only elements with a high degree of commonality (e.g., see Figure 7.1(c)). When the class similarity threshold is high (0.9), only identical classes got combined. A large number of classes, states and transition in the resulting model is captured by a low calculated value for both $Q_1$ and $Q_2$, since both of them are designed to minimize the size of the result.

For the Microwave Oven example, both $Q_1$ and $Q_2$ prefer the solution found when the class similarity threshold is set to 0.7 or lower (see Table 7.4). Since all four variants of the timer control class in this example are very similar, these classes are all merged together in the resulting refactoring. The percentage of variable states and
transitions in this solution remains small, and the overall reduction in their total number is significant.

Recall that our third example had two clusters of similar components (and two other components, different from the rest). The refactoring that identifies and combines components in these clusters is produced when the class similarity threshold is set to 0.7. This refactoring also maximizes the value of $Q_2$. Similarly to the Washing Machine case, lower threshold values produce more merges resulting in a high number of variable attributes, states and transitions (and thus, lower values of $Q_2$), while higher thresholds result in a large number of elements in the resulting model (and thus, lower values of both $Q_1$ and $Q_2$).

In summary, we found that in all of the above cases, quality functions were able to distinguish between different refactorings as desired and thus were appropriate to help “drive” the refactoring process towards the preferable result. Our third case study also showed that differences in the computed quality values became more pronounced as models got bigger. Furthermore, the refactorings that were produced in our examples under the strategy that maximizes the value of $Q_2$ were identical to those constructed manually by a domain expert. This encouraging result makes us believe that our quality-based merge-refactorings approach is effective for the creation of annotative SPL representations from a set of existing systems.

### 7.7.3 Threats to Validity

Threats to external validity are most significant for our work. These arise when the observed results cannot be generalized to other case studies. Because we used a limited number of subjects and quality functions, our results might not generalize without an appropriate tuning. However, we attempted to mitigate this threat by using a real-life case study of considerable size as one of our examples. Thus, even though preliminary, our results show that the approach, perhaps with some additional tuning, is effective for finding good refactorings of large-scale systems.

In addition, we limit the scope of our work to physical systems in the embedded domain, where the number of product variants usually does not exceed tens. The approach might not scale well to other domains, where hundreds of product variants are possible. However, we believe that the scalability issue mostly relates to the annotative SPL representation itself, rather than to our attempt to distinguish between different representations.

### 7.8 Related Work

#### Product Line Refactoring Approaches

A general theory of product line refinement was introduced in [18] where the authors established product line properties supporting stepwise and compositional product line development and evolution. Our approach instantiates this theory by providing a concrete refactoring technique for combining products into product lines. We prove that our refactoring is the minimal behavior-preserving product line refinement, according to the definition in [18].

---

**Table 7.4: Varying Class Similarity Threshold $S_{\text{Class}}$.**

<table>
<thead>
<tr>
<th>metrics</th>
<th>Washing Machine</th>
<th>Microwave Oven</th>
<th>CE Product Line</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>orig. 0.7 0.75 0.8 0.85 0.9</td>
<td>orig. 0.7 0.75 0.85 0.9</td>
<td>orig. 0.6 0.65 0.7 0.75 0.8</td>
</tr>
<tr>
<td>$v_1$ #classes</td>
<td>18 6 7 8 9 11 12</td>
<td>8 2 3 5</td>
<td>45 14 15 17 20 27</td>
</tr>
<tr>
<td>$v_2$ #attributes</td>
<td>25 10 12 14 20 25 25</td>
<td>4 1 2 4 104</td>
<td>56 56 75 84 80</td>
</tr>
<tr>
<td>$v_3$ #states</td>
<td>13 4 5 6 4 0</td>
<td>0 0 2 4</td>
<td>43 43 26 32 8</td>
</tr>
<tr>
<td>$v_4$ #transitions</td>
<td>56 28 31 34 40 51 56</td>
<td>44 16 24 44</td>
<td>944 245 260 402 374 412</td>
</tr>
<tr>
<td>$v_5$ #var.attributes</td>
<td>3 4 5 6 4 0</td>
<td>- 1 0 0</td>
<td>- 56 64 31 13 4</td>
</tr>
<tr>
<td>$v_6$ #var.states</td>
<td>- 6 6 5 4 0</td>
<td>44 16 24 44</td>
<td>944 245 260 402 374 412</td>
</tr>
<tr>
<td></td>
<td>18 20 22 28 28 38 43</td>
<td>18 7 9 18 448</td>
<td>177 151 211 374 412</td>
</tr>
<tr>
<td></td>
<td>43 18 20 22 28 28 38 43</td>
<td>18 7 9 18 448</td>
<td>177 151 211 374 412</td>
</tr>
<tr>
<td></td>
<td>0.587 0.528 0.469 0.333 0.148 0.083</td>
<td>0.565 0.560 0.572 0.533 0.523 0.541</td>
<td>0.797 0.797 0.797 0.797 0.797 0.797</td>
</tr>
<tr>
<td></td>
<td>0.686 0.505 0.065</td>
<td>0.646 0.650 0.668 0.601 0.623</td>
<td>0.646 0.650 0.668 0.601 0.623</td>
</tr>
<tr>
<td></td>
<td>0.646 0.650 0.668 0.601 0.623</td>
<td>- 0.646 0.650 0.668 0.601 0.623</td>
<td>- 0.646 0.650 0.668 0.601 0.623</td>
</tr>
</tbody>
</table>

Several works (e.g., [87, 90]) capture guidelines and techniques for manually transforming legacy product line artifacts into SPL representations. Instead, our goal is to introduce automation into the refactoring process by comparing, matching and composing artifacts. While no automated approach can replace a human product line designer and produce a solution which is as good as a hand-crafted one, automation can assist the designer and speed-up the refactoring process.

Koschke et al. [91] present an automated technique for comparing software variants at the architectural level and reconstructing the system’s static architectural view which describes system components, interfaces and dependencies, as well their grouping into subsystems. Ryssel et al. [148] introduce an automatic approach to re-organize Matlab model variants into annotative representations while identifying variation points and their dependencies. Yoshimura et al. [180] detect variability in a software product line from its change history. None of the above approaches, however, takes into account quality attributes of the constructed results nor attempts to distinguish between the different refactorings based on the refactoring goal.

Feature-oriented refactoring [103, 111] focuses on identifying the code for a feature and factoring the code out into a single module or aspect aiming at decomposing a program into features. Since our aim is consolidation of variants into single-copy SPL representations, these are out of the scope for our work. Similarly, UML model refactoring (e.g., [71, 163]) and code refactoring techniques (e.g., [110]), while closely related to our work, usually focus on improving the internal structure and design of a software system rather than on identifying and restructuring the system’s common and variable parts.

Visualization techniques for presenting annotative SPL models are out of scope of this thesis. However, such techniques (e.g., [84]) can clearly enhance our work.

**Product Line Quality.** Oliveira et al. [37] propose a metric suite to support evaluation of SPL architectures based on McCabe’s cyclomatic complexity of their core components, which is computed using the control flow graph of the program and measures the number of linearly independent paths through a program’s source code. Her et al. [66] suggest a metric to measure reusability of SPL core assets based on their ability to provide functionality to many products of the same SPL, the number of SPL variation points that are realized by an asset, the number of replaceable components in a core asset and more. Hoek et al. [170] describe metrics for measuring service utilization of SPL components based on the percentage of provided and required services of a component. While these works allow measuring reusability, extensibility and implementation-level complexity of SPL core assets, they do not discuss the structural complexity of annotative SPL models nor allow comparing different annotative SPL models and distinguishing between them based on their representation properties. Trendowicz and Punter [167] investigate to which extent existing quality modeling approaches facilitate high quality product lines and define requirements for an appropriate quality model. They propose a goal-oriented method for modeling quality during the SPL development lifecycle, but do not propose any concrete metrics.

Numerous works, e.g., [51, 33], propose software metrics for evaluating the quality of UML models. While we base our approach on some of these works, they are not designed for UML models that represent software product lines and do not take variability aspects into account.

Finally, some works discuss characteristics of feature implementations in code, such as feature cohesion and coupling [5] or granularity, frequency and structure of preprocessor annotations [83, 100]. However, these works are not easily generalizable to address the issue of structural complexity of models.
7.9 Summary

In this chapter, we contributed an implementation of the merge operator defined in Chapter 4 for UML models containing class and statechart diagrams. Our algorithm, called merge-in, relies on model compare, match and compose. We formally specified necessary and sufficient conditions allowing us to use these steps to construct provably correct merging – the one that produces representations that encode precisely the set of initial products.

We also argued that multiple syntactically different yet semantically equivalent representations of the same SPL model are possible, and the goal of the refactoring determines which one is preferable. We suggested an approach for guiding the refactoring process towards a result that best fits user intention, as captured by a syntactic quality function. We used the merge-in algorithm as the foundation of our quality-based merge-refactoring approach. We evaluated the proposed approach on a set of case-studies, including a large-scale example contributed by an industrial partner. We believe that our work promotes automation of product line refactoring and reasoning about refactoring alternatives.

In its current form, our merge-in algorithm combines input models one by one, in a preset order. The next chapter extends this work by exploring whether some order of combinations would be more beneficial than others and suggesting approaches for simultaneously considering multiple models together, without combining them in any particular order.
Chapter 8

N-Way Model Merging

In this chapter, we take a closer look at the problem of merging multiple input models into one. Most existing approaches, including the one discussed in Chapter 7, perform such merging in a pairwise manner, i.e., merge $n$ models using $n - 1$ pairwise operations. A subset-based approach further generalizes on that, allowing to fix a certain number of input models that are processed in each iteration. However, the quality of the result produced using subset-based approaches, as well as the effect of the chosen subset size or the order in which the input models are picked, remains unclear.

We thus redefine the model merging problem to consider $n$ inputs at once – an approach we refer to as $n$-way merge. We refine existing compare, match and compose definitions to consider tuples rather than pairs of elements from multiple distinct input models. We focus on the matching step – the most challenging among the three – and show that it can be reduced to the weighted set packing problem which is known to be NP-hard [7]. We study and compare the state-of-the-art approximations to that problem, both theoretically and empirically, exploring their applicability to merging real-life models of a reasonable size. We discover that the scalability of these approximation algorithms is limited to a small number of small models and thus that they cannot be directly applied to real-life cases of model merging.

Further, we investigate the quality of the subset-based incremental approaches and propose our own polynomial-time heuristic $n$-way matching algorithm, $NwM$, that considers all input models simultaneously rather than processing them in any specific order. We empirically evaluate the quality and the performance of $NwM$ by comparing it to the subset-based approaches, using as subjects two real-life and 300 randomly generated models. We show that $NwM$ outperforms the alternative approaches in the majority of the cases, without any significant performance penalty.

Contributions. This chapter makes the following contributions.

1. We formally define the $n$-way model merging problem.
2. We show that its matching stage is equivalent to the weighted set packing problem, which is known to be NP-hard.
3. We evaluate the state-of-the-art approximation algorithms for the weighted set packing problem both theoretically and empirically, and show that they do not scale well for real-life problems of realistic size.
4. We describe a number of polynomial-time subset-based approaches for $n$-way matching.
5. We contribute a novel heuristic algorithm, $NwM$, that simultaneously considers all $n$ input models together.
6. We evaluate the proposed \textit{NwM} algorithm both theoretically and empirically, and show that it substantially outperforms the subset-based approaches w.r.t. the quality of the produced results without any real degradation in performance.

This chapter is based on our publication in [142]. We proceed as follows. Section 8.1 describes the running example and illustrates n-way merging. We define n-way merge in Section 8.2 and study the applicability of existing solutions to this problem, via reduction to the NP-hard weighted set problem, in Section 8.3. We describe polynomial-time n-way matching approaches, including the novel \textit{NwM} algorithm, in Section 8.4. We evaluate these approaches theoretically and empirically in Section 8.5. We discuss related work in Section 8.6 and conclude in Section 8.7.

\section{Example}

We illustrate the n-way model merging problem using small UML model fragments inspired by a health care case study (see Figure 8.1(a)). The first model fragment, $M_1$, contains a single UML class \texttt{CareTaker} (element #1) which has two attributes: id that uniquely identifies the caretaker person and office that specifies where the person can be found. The second fragment, $M_2$, also contains two classes: \texttt{Physician} (element #2) and \texttt{Nurse} (element #3). Both classes have two attributes that help to identify and locate the corresponding persons: name and office for \texttt{Physician} and id and ward for \texttt{Nurse}.

When merging these two fragments, element #1 from $M_1$ can be combined either with element #2 or #3 from $M_2$. In fact, we cannot tell at this point which combination is better – in both cases, the classes have one shared
The third model fragment, $M_3$, contains a single class Nurse (element #4) that has two attributes: name and ward. As it was already decided to combine elements #1 and #3 in the previous step, element #4 can be combined either with the element that corresponds to their union or with element #2. None of these results are desired though: when considering all three fragments together, it becomes apparent that nurses belong to wards while physicians have offices. Thus, it is more appropriate to combine element #1 with #2 (rather than with #3, as was decided earlier) and then combine element #3 with #4 – see bold lines connecting the corresponding elements in Figure 8.1(a). Figure 8.1(b) shows the result of the corresponding merge, where elements are annotated by models from which they originate: elements #5 and #6 represent the merge of element #1 with #2 and #3 with #4, respectively.

This example illustrates the need and the value of considering multiple models simultaneously, as a matching decision made in a certain iteration of a pairwise merging approach (e.g., combining elements #1 and #3 in Figure 8.1(a)) can impede reaching the desired result in later iterations. That is, one needs to take into account the “global picture” as the information available for a particular subset of models is simply insufficient for reaching the “right” decision. The example also illustrates the sensitivity of the pairwise approach to the order in which the input models are picked: in this case, considering inputs in a different order, e.g., merging $M_1$ and $M_3$ first, would produce the desired result.

Fixing a particular order cannot guarantee the desired result in the general case though, as will be shown by our larger case studies. Intuitively, optimal matches are spread “across” models, as schematically shown by four model fragments in Figure 8.2. Picking any two models without considering the global picture might result in “binding” elements of the two models with each other, instead of “keeping” some elements available for later, more desired, combinations.

Figure 8.1(c) shows yet another set of model fragments inspired by the health care case study. Models $M_4$, $M_5$ and $M_6$ differ from the previous ones because the CareTaker class of $M_4$ (element #7) now has attributes name and ward, making it more similar to the Nurse class of $M_5$ (element #9) than to Physician (element #8). Also, the Nurse class of $M_5$ now has the attribute name instead of id. As the result, elements #7, #9 and #10 are combined together, while element #8 is just copied over to the merged result, as shown in Figure 8.1(d).

This example illustrates the case where the desired result is obtained by combining elements from three distinct models instead of producing pairs, like in Figure 8.1(b). Throughout the rest of the chapter, we use the above examples to showcase the merging approaches being discussed.
8.2 N-Way Model Merging

In this section, we generalize the definition of merging given in Chapter 7 to $n$ models. We formally specify the n-way counterparts of the compare, match and compose operators and discuss issues related to their implementation.

We assume $n$ input models $M_i$, $i \in [0..n]$, of size $k_i$ each. We use $k$ to denote the size of the largest input model. Each model $M_i$ contains uniquely identifiable elements $e_1 \ldots e_{k_i}$. Elements of distinct models form $n$-tuples. For an $n$-tuple $t$, we denote by $\mu(t)$ the ordered list of input models from which the elements of $t$ originate.

**Definition 18** An $n$-tuple (a.k.a. a tuple) $t$ is a set of elements satisfying the following properties:

(a) It contains at least one element: $|t| \geq 1$.

(b) No two elements belong to the same input model: $|t| = |\mu(t)|$.

For the example in Figure 8.1(a), a possible tuple can consist of elements #1, #2 and #4 from models $M_1$, $M_2$ and $M_3$, respectively, and is denoted by <1,2,4>. $\mu(1,2,4) = \{M_1, M_2, M_3\}$. Other possible tuples are <1,2> and <1,3,4> but not <1,2,3> since it contains two elements, #2 and #3, from model $M_2$.

In what follows, let $T$ denote the set of all valid tuples for the input models $M_i$, $i \in [0..n]$. The size of $T$ is $(\prod_{i=1}^{n}(k_i + 1)) - 1$, accounting for choosing an element from each model, including none, but disallowing an empty tuple.

### 8.2.1 Definition of the Operators

**Compare** assigns a similarity measure to a given tuple $t \in T$. We refer to the result of this function as a tuple’s weight and denote it by $w(t)$. Thus, compare is a function that receives a tuple $t \in T$ and returns the similarity measure $w(t) \in [0..1]$ for its elements. The larger the value of $w$, the more similar to each other the elements of $t$ are considered to be.

**Match** considers the compared tuples and selects those that are deemed similar. A validity function $V$ decides whether a tuple is eligible to be selected. It extends a simple threshold-based comparison described in Chapter 7 to include more sophisticated validity criteria.

The weight of the set of tuples $\hat{T} \subseteq T$ produced by match is defined as a sum of weight of all tuples in $\hat{T}$ and is denoted by $w(\hat{T})$: $w(\hat{T}) = \sum_{t \in \hat{T}} w(t)$. The larger the value of $w(\hat{T})$, the better is the produced match. Further, match should produce a disjoint set of tuples – in this work, we assume that an element can only be matched with a single element of any given input model, leading to the following definition:

**Definition 19** Let $V$ be a boolean validity function. Then, match is a function that returns a set of matches $\hat{T} \subseteq T$ that satisfy the following properties:

(a) All tuples are valid: $\forall \hat{t} \in \hat{T} : V(\hat{t}) = true$.

(b) All tuples are disjoint: $\forall \hat{t}, \hat{t}' \in \hat{T} : \hat{t} \cap \hat{t}' = \emptyset$.

(c) The set $\hat{T}$ is maximal: no additional tuples can be added to $\hat{T}$ without violating constraints (a) and (b).

For the example in Figure 8.1(a), match can output the tuples <1,2> and <3,4>. It can also output either <1,2,4> or <1,3,4>, but not both, since otherwise elements #1 and #4 would be matched to both elements #2 and #3 of $M_2$. Tuple <1,3> only is not a valid result as it can be augmented with the tuple <2,4>. 
Compose combines elements of each matched tuples in a particular way. Figure 8.1(b) shows a possible result of merging the input models in Figure 8.1(a), when merge assumes annotative software product line union-merge semantics and match produces tuples (1,2) and ⟨3,4⟩.

8.2.2 Towards Implementation of the Operators

Compare and compose usually encapsulate domain-specific information, i.e., which elements of a specific domain are considered similar and how such similar elements should be combined. Numerous works, e.g., [178, 85, 112, 6], proposed implementations of these operators for both models and code, taking into account syntactical properties of the considered artifacts. On the other hand, match relies on the result of compose rather than domain-specific knowledge and is the least explored operator, especially for a collection of \( n \) inputs. Thus, the main focus of this chapter is on the implementation of the n-way match step.

Yet, while a discussion of different ways for performing compare or compose is out of scope of this chapter, we need to pick an operational definition of these operators which we describe below. We assume that each model element contains a set of typed properties, and we compare elements based on these. If model elements are UML classes, as in the example in Figure 8.1(a), possible properties can be class names, attributes, methods, associations to other classes, generalization relationships, etc. For example, element \#1 in Figure 8.1(a) is defined by properties id and office of type UML class attribute and CareTaker of type UML class name. For simplicity, in our presentation we do not consider elements of types other than UML classes, although it is easy to extend the element / property approach to other types as well.

For an element \( e \), we denote by \( \pi(e) \) the set of all of its properties. Similarly, for a tuple \( t = (e_1, \ldots, e_m) \), we denote by \( \pi(t) \) the set of distinct properties of all elements of \( t \): \( \pi(t) = \bigcup_{i \in t} \pi(e_i) \).

We assume, without loss of generality, that the goal of compare is to assign high weight to tuples whose elements share similar properties. For each tuple \( t \) and a property \( p \), compare considers the number of elements in \( t \) that have that property. Then, for each tuple \( t \), it calculates the distribution of properties: the number of properties that appear in \( j \) elements of \( t \). We denote this number by \( n_j^p \). For the example in Figure 8.1(a), the property name appeared twice in tuple \( \langle 1,2,4 \rangle \) (in classes Physician and Nurse), as well as the property office (in classes CareTaker and Physician). The remaining properties, namely, attributes id and ward, as well as class names CareTaker and Physician and Nurse are unique to a single element in the tuple. Such properties “differentiate” tuple elements from each other and compare should “penalize” for that.

We thus define the compare function to assign a high weight to tuples with a large number of properties shared by a large number of elements: \( w(t) = \frac{\sum_{j=1}^{m} j^2 \cdot n_j^p}{n^2} \).

The result is normalized by the number of input models \( n^2 \) (rather than the size of the tuple \( n^2 \)) so that tuples with elements from a small subset of input models receive lower scores.

For example, the result of applying compare on the tuple \( \langle 1,2 \rangle \) in Figure 8.1(a) is \( \frac{2^2 + 1}{4 \cdot 2} = \frac{4}{8} \); only the office attribute appears in both elements of the tuple, while there are five distinct properties in total – the class names CareTaker and Physician, and the class attributes id, office and name. Applying compare on tuples \( \langle 1,3 \rangle \) and \( \langle 2,4 \rangle \) results in \( \frac{2^2 + 1}{4 \cdot 2} = \frac{4}{8} \) as well. Compare on tuple \( \langle 3,4 \rangle \) yields \( \frac{2^2 + 2}{4 \cdot 3} = \frac{2}{9} \); applying it to \( \langle 1,3,4 \rangle \) also results in the same value, i.e., \( \frac{2^2 + 2}{4 \cdot 3} = \frac{2}{9} \). The weight of a tuple containing a single element, e.g., \( \langle 1 \rangle \), as well as the weight of a tuple whose elements are completely disjoint, e.g., \( \langle 1,4 \rangle \), is defined to be zero.

Our validity criteria require that each element of a tuple share at least one property with other elements. For example, the tuple \( \langle 1,4 \rangle \) in Figure 8.1(a) is invalid, as classes CareTaker and Nurse have no attributes in common.
### Table 8.1: Matching algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Approximation factor</th>
<th>Time complexity</th>
<th>Memory efficient</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Greedy (Gr)</strong></td>
<td>( n )</td>
<td>( O(k^{2+n}) )</td>
<td>( O(n^3 \cdot k^{2+n+1}) )</td>
</tr>
<tr>
<td>Local Search Pick First</td>
<td>( n - 1 + 1/s )</td>
<td>( O(k^{n+2+2n+1} \cdot s \cdot n^3) )</td>
<td>( O(k^{n+2+2n+1} \cdot s \cdot n^3) )</td>
</tr>
<tr>
<td>Local Search Pick Best</td>
<td>( 2 \cdot (n+1)/3 )</td>
<td>( O(k^{(n+1)^2} \cdot n^3) )</td>
<td>( O(k^{(n+1)^2} \cdot n^6) )</td>
</tr>
<tr>
<td>Local Search Squared Pick Best</td>
<td>( (n+1)/2 )</td>
<td>( O(k^{(n+1)^2+n} \cdot n^4) )</td>
<td>( O(k^{(n+1)^2+n} \cdot n^6) )</td>
</tr>
<tr>
<td>Pairwise (PW)</td>
<td>–</td>
<td>( O(n \cdot k^3) )</td>
<td></td>
</tr>
<tr>
<td>Greedy 3 (Gr3)</td>
<td>–</td>
<td>( O(n \cdot n^3 \cdot k^7) )</td>
<td></td>
</tr>
<tr>
<td>Greedy 4 (Gr4)</td>
<td>–</td>
<td>( O(n \cdot n^3 \cdot k^9) )</td>
<td></td>
</tr>
<tr>
<td><strong>NwM</strong></td>
<td>–</td>
<td>( O(n^3 \cdot k^4) )</td>
<td></td>
</tr>
</tbody>
</table>

Indeed, such classes should never be matched\(^1\). All of the remaining tuples in this example are valid.

Given the compare function and the validity criteria, the match with the maximal weight is produced by a combination of two tuples: \( \langle 1,2 \rangle \) and \( \langle 3,4 \rangle \). The weight of this solution is \( \frac{4}{15} + \frac{2}{5} = \frac{14}{15} \), which is higher than picking the tuples \( \langle 1,3 \rangle \) and \( \langle 2,4 \rangle \) instead or a single tuple that combines three elements from distinct models: either \( \langle 1,2,4 \rangle \) or \( \langle 1,2,3 \rangle \). For the example in Figure 8.1(c), the best match corresponds to the solution that has a single tuple \( \langle 7,9,10 \rangle \) with the weight \( \frac{3^2+2^2+1}{3+9+10} = \frac{11}{18} \). In this case, the value is higher than for any other combination of tuples.

For compose, we assume the annotative SPL union-merge semantics. Its implementation is a straightforward extension to the compose operator discussed in Chapter 7: elements of each tuple in the solution found by match are combined by unifying their properties, and are annotated by the corresponding source model for traceability purposes. Elements of the input models that are not part of any tuple produced by match are copied to the result as is. For the example in Figure 8.1(a), where match produced two tuples \( \langle 1,2 \rangle \) and \( \langle 3,4 \rangle \), the two elements in the merged model in Figure 8.1(b), i.e., elements #5 and #6, correspond to the merge of elements in the first and the second tuple, respectively. The merge of the models in Figure 8.1(c) is shown in Figure 8.1(d). Here, element #11 corresponds to the single tuple \( \langle 7,9,10 \rangle \) returned by match. Element #12 corresponds to element #8 of the input model \( M_5 \) which is not part of any tuple and thus is copied to the result as is.

### 8.3 N-Way Matching via Weighted Set Packing

In this section, we show that the n-way matching problem is reducible to the well-known NP-hard problem of weighted set packing \(^7\). We then consider the applicability of the existing approximations of this problem to merging software models.

\(^1\)For tuples of size two, the similarity measure of invalid tuples is 0, but that does not hold for larger tuples.
Table 8.2: Execution time for Greedy on Hospital and Warehouse cases.

<table>
<thead>
<tr>
<th></th>
<th>Hospital</th>
<th></th>
<th>Warehouse</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_1=26$, $k_2=28$, $k_3=38$, $k_4=25$, $k_5=22$, $k_6=18$</td>
<td></td>
<td>$k_1=18$, $k_2=31$, $k_3=27$, $k_4=24$, $k_5=25$, $k_6=19$</td>
<td></td>
</tr>
<tr>
<td>$n=3$</td>
<td>$n=4$</td>
<td>$n=5$</td>
<td>$n=6$</td>
</tr>
<tr>
<td>&lt; 5s</td>
<td>1.86m</td>
<td>38.44m</td>
<td>10.9h</td>
</tr>
<tr>
<td></td>
<td>&lt; 2s</td>
<td>46s</td>
<td>20m</td>
</tr>
</tbody>
</table>

8.3.1 Weighted Set Packing and Its Approximation Algorithms

The reduction to the weighted set packing problem easily follows from its definition: given a collection of weighted sets of cardinality at most $n$ (in our case, approximately $k^n$ tuples consisting of elements from $n$ input models together with their weights as calculated by `compare`), the weighted set packing produces a collection of disjoint sets with the maximum total weight (in our case, the match). The problem is NP-hard [7], and thus no solution that is polynomial in the size of the input (in our case, the set of tuples), exists.

There are a number of theoretically bounded approximations to that problem, polynomial in $k^n$ (and thus exponential in $n$). Their main properties, i.e., approximation factor and time complexity, are summarized in the first four rows of Table 8.1.

The simplest approximation algorithm (Greedy) [24] picks tuples with the maximal weight out of the set of all possible tuples, disregarding tuples that contain the elements already picked in the previous iterations. This algorithm can clearly miss the optimal result: picking a tuple with the maximal weight without “looking ahead” can block selection of combinations with a higher total weight. Since a selection of a tuple $t$ with the weight $w$ can block the selection of up to $n$ other tuples whose weight cannot exceed $w$, the approximation factor of Greedy is $n$ [24], i.e., the weight that this algorithm computes is within $n$ times of the optimal. For the example in Figure 8.1(a), Greedy might pick the tuple $\langle 1,3,4 \rangle$ with the weight $\frac{14}{25}$, preventing the generation of the optimal solution with tuples $\langle 1,2 \rangle$ and $\langle 3,4 \rangle$ and weight $\frac{14}{25}$. For the example in Figure 8.1(c), Greedy does find the optimal solution – the tuple $\langle 7,9,10 \rangle$.

A number of approaches improve on Greedy by combining it with different types of local search [7, 24, 17]. They start from the solution found by Greedy and iteratively attempt to improve it by selecting $s$ disjoint tuples that are not part of the current solution and trying to swap them with a minimal set of tuples in the solution so that tuples in the solution remain disjoint. The algorithms vary by the selection criterion for swapping (either the total weight of the solution increases or the square of the weight increases), the selection of the swapping candidate (either the first set that satisfies the swapping criterion or the best one) and by the size $s$ of the candidate set. For the example in Figure 8.1(a), if Greedy produced the solution consisting of $\langle 1,3,4 \rangle$, such approaches can swap it with the pair of tuples $\langle 1,2 \rangle$ and $\langle 3,4 \rangle$, producing the optimal result. The approximation factors of these algorithms, as given by their corresponding authors [7, 24, 17], are summarized in Table 8.1.

Both Greedy and local search-based algorithms have high runtime and space complexity. In fact, even keeping all tuples in memory, let alone the information about which tuples are disjoint, is not possible for more than a few models with only around 30 elements each. In the last two columns of Table 8.1, we list time complexity for two version of each algorithm – the ones that keep all data in memory and the ones that do not.
CHAPTER 8. N-WAY MODEL MERGING

8.3.2 Preliminary Evaluation

The time complexity estimations given in Table 8.1 might be too high for real-life scenarios. Also, our weight and validity functions reduce the space of tuples considered by the algorithms by assigning zero weight to a large set of tuples. We thus experimented with applying the proposed approximation solutions to the n-way matching problem on real-life examples.

We implemented the algorithms in Java, introducing numerous optimizations, including multi-threading, to improve execution times. We executed these algorithms on an Intel Core2Quad CPU 2.33GHz machine using JVM version 7.

As subjects, we picked two sets of models reported in [123]. The first consists of eight different variants of a Hospital system modeled as UML class diagrams. The system handles services within the hospital and describes the role of the hospital staff, its wards, different types of patients, and more. A small snippet of the models is shown in Figure 8.1(a). In total, there are 221 classes in all eight models, with an average of 27.63 classes per model. The largest model has 38 classes while the smallest has 18. Classes have 4.76 attributes on average (including name, associations, inheritance relationships, etc.). The largest class has 15 attributes. Around 30% of the attributes appear in more than 5 models.

The second set consists of sixteen different variants of a Warehouse system, also modeled as UML class diagrams. The system is designed to handle orders, track their status, provide computer support for warehousing workers, keep track of inventory levels in the warehouse, etc. Warehouse models have 388 elements in total, with an average of 24.25 classes per model. The largest and the smallest models have 44 and 15 classes, respectively. Classes have 3.67 attributes on average, with 11 attributes in the largest class. Around 15% of the attributes appear in more than 5 models. A complete description of both case studies can be found in [123].

Our experiment showed that none of the algorithms scaled to operate on the complete set of the input models, i.e., none achieved termination after 5 hours. Instead, we tried running the algorithms on the first three, four, five and six models from each of the case studies, in the order of their appearance in [123]. Algorithms based on local search failed to terminate after 5 hours even on a three-model subset. Execution times for Greedy are shown in Table 8.2. $k_i$s capture the number of elements in each model.

Even though it might be possible to come up with more efficient implementations of the algorithms than ours, the results indicate that, generally, the algorithms do not scale well. Greedy seems to be the only feasible approach, and only for merging up to five small models (with 20-40 elements). This calls for a different solution for the n-way merging problem, polynomial in both $k$ and $n$. We explore it in the remainder of the chapter.

8.4 Polynomial-Time Approach to N-Way Matching

In this section, we discuss algorithms which are polynomial both in the number of input models and in their size. First, we define a set of solutions that incrementally combine all input models in small subsets and discuss their properties (Section 8.4.1). We then present a novel algorithm, $NwM$, that considers all input models simultaneously (Section 8.4.2). Unlike the approximation algorithms for the weighted set packing problem, the algorithms presented in this section are not guaranteed to produce an answer within a particular factor of the optimal. We empirically evaluate the algorithms in terms of their quality and scalability in Section 8.5.
8.4.1 Subset-Based Approaches

A straightforward solution for merging $n$ input models is to do so in smaller subsets, e.g., in pairs, performing $n-1$ pairwise combinations. To do so, we maintain a pool of models, with all input models initially in it. The algorithm iteratively selects and removes a subset of models from the pool, merges them together and puts the result back into the pool for further merging with additional models.

Subsets of size two can be merged using the bipartite graph matching algorithm [96] which produces a disjoint set of matches with the maximal total weight. The algorithm is based on combinatorial optimization techniques and solves the matching problem in time polynomial in size of the larger input model, returning an optimal result [45]. Larger subsets, of three or four models, can be merged using the Greedy algorithm described in Section 8.3.1. Applying Greedy on more than four models, or applying additional, more sophisticated, algorithms does not scale well, as shown in Section 8.3.2.

We thus define three subset-based algorithms: $PW$ – pairwise matching, $Gr3$ – Greedy on subsets of size 3 and $Gr4$ – Greedy on subsets of size 4. These are summarized in Table 8.1. The quality of the result produced by these algorithms, in terms of the total weight of the solution, largely depends on the order in which the models are picked. For the example in Figure 8.1(a), selecting models $M_1$ and $M_2$ first can result in merging elements #1 and #3 with each other, as this combination is indistinguishable from the more desired combination of #1 and #2 since both pairs of elements have the same weight. As the result of this selection, it is impossible to generate the merged model shown in Figure 8.1(b) where element #1 is combined with #2, while #3 is combined with #4. Picking models $M_1$ and $M_3$ first could produce a better result if the highly dissimilar elements #1 and #4 are not merged. Then, during the next iteration, these elements could be combined with elements #3 and #2 from $M_2$, respectively, producing the desired combination. The above discussion also makes it clear that subset-based incremental algorithms have no theoretical approximation factor since any merge of two elements produced in a given iteration can prevent future, more efficient, combinations.

To consider different orderings of input modes, we define and evaluate three variants of each subset-based algorithm. The first picks and processes the input models in the order of their appearance (i.e., at random). We denote such algorithms by $PW$, $Gr3$ and $Gr4$. The second variant arranges the input models by size in ascending order, with the corresponding algorithms denoted by $PW^\uparrow$, $Gr3^\uparrow$ and $Gr4^\uparrow$. The third variant arranges them in descending order, with the corresponding algorithms denoted by $PW^\downarrow$, $Gr3^\downarrow$ and $Gr4^\downarrow$. For the example in Figure 8.1(a), algorithm $PW$ picks models $M_1$ and $M_2$ first, $PW^\uparrow$ picks models $M_1$ and $M_3$, while $PW^\downarrow$ picks either $M_2$ and $M_3$, or $M_2$ and $M_1$. $Gr3$, $Gr3^\uparrow$ and $Gr3^\downarrow$ are equivalent in this case, as there are only three input models in total, so the ordering does not make any difference (and $Gr4$ algorithms are not applicable at all).

We evaluate the relative effectiveness of these algorithms in Section 8.5. We also experimented with the idea of ordering input models by their cohesiveness, i.e., first merging those that are most similar, but observed that there is a strong correlation between this approach and the size-based ordering: larger models produce more matches which increases the total weight of the overall result and thus the similarity of these models.

8.4.2 The Algorithm NwM

In this section, we present a novel algorithm for n-way merging, $NwM$, which considers all $n$ input models together and thus does not depend on any particular order of model selection (see Algorithm 1). Its main idea is based on picking optimal matches from distinct models and incrementally grouping them until a maximal set of tuples is produced. The algorithm obtains as input a set of tuples $\hat{T} \in T$ and outputs either a set of tuples $\hat{S} \in T$ over the same elements, with improved total weight: $w(\hat{S}) > w(\hat{T})$, or the input set $\hat{T}$, if such improvement is not
possible.

**Brief overview of the algorithm.** In the first iteration, elements of all input models $M_1 \ldots M_n$ are represented by individual, single-element, tuples. For the example in Figure 8.1(a), the tuples are: $\langle 1 \rangle$, $\langle 2 \rangle$, $\langle 3 \rangle$ and $\langle 4 \rangle$. Pairs of input tuples are assigned weights (Line 3) and matched using the bipartite graph match algorithm (Line 5). For each input tuple, the algorithm selects either zero or one matching counterpart, maximizing the total weight of the result. For the example in Figure 8.1(a), tuples $\langle 1 \rangle$ and $\langle 2 \rangle$, as well as $\langle 3 \rangle$ and $\langle 4 \rangle$, are matched with each other after the first iteration of the algorithm.

Matched tuples are further processed and unified (Lines 6-22), producing tuples $\langle 1,2 \rangle$ and $\langle 3,4 \rangle$ in the above example. The unified tuples are used as input to the next iteration of the algorithm (Line 27). The algorithm terminates when no matches of the input tuples can be made and thus no improvements can be achieved (Lines 24-25).
Detailed description of the algorithm. In every iteration, all pairs of input tuples are assigned weights. These weights are further used by the bipartite graph matching algorithm to produce an optimal match – a disjoint set of tuple pairs with the maximal total weight. Since matched tuples are further unified, we assign a pair of tuples \((\hat{t}_1, \hat{t}_2)\) the weight of the tuple that corresponds to their union (see the otherwise clause in Line 3 of Algorithm 1):

\[
w(\hat{t}_1, \hat{t}_2) = w(\hat{t}_1 \cup \hat{t}_2).
\]

For the example in Figure 8.1(a), \(w((1),(2)) = w((1),(3)) = w((2),(4)) = 4\), and \(w((3),(4)) = \frac{2}{3}\).

Some pairs correspond to tuples that should never be unified, e.g., when the unified tuple is invalid w.r.t. the validity function \(V\). For our example in Figure 8.1(a), the tuples \((1)\) and \((4)\) share no common properties and thus we treat their combination as invalid. While it is possible to filter such bad combinations after the matching is done, preventing their generation in the first place is preferred since the original tuples can then participate in more desired combinations. We thus assign weight 0 to combinations that are a priori “illegal”, relying on the bipartite graph matching algorithm’s ability to ignore pairs of elements with zero weight. Four types of such illegal combinations are described below (and encoded in Line 3 of Algorithm 1):

1. Pairs whose unification results in a tuple which is invalid w.r.t. the validity function \(V\), e.g., the pair \((1), (4)\).
2. Pairs for which the weight of the union is lower than the sum of input tuple weights – unifying such pairs is clearly not worthwhile. This situation cannot occur when unifying single-element tuples as their weight is zero and the unification can only increase it.
3. Pairs that contain elements from the same model, that is, \(\mu(\hat{t}_1) \cap \mu(\hat{t}_2) \neq \emptyset\). For example, \(\mu((1,2))\) in Figure 8.1(a) is \(\{M_1, M_2\}\), while \(\mu((3,4))\) is \(\{M_2, M_3\}\). Unifying these tuples would result in a tuple with two elements, \#2 and \#3, from \(M_2\), which is not allowed.
4. Pairs introducing “circular” dependencies between tuples, i.e., if a tuple \(\hat{t}_1\) is matched with \(\hat{t}_2\), \(\hat{t}_2\) is matched with \(\hat{t}_3\), and \(\hat{t}_1\) and \(\hat{t}_3\) contain elements from the same set of models, the unified tuple would be illegal. To limit circular dependencies, we introduce a partial order, \(\prec_o\), on the set of all tuples, such that \(\hat{t}_1 \prec_o \hat{t}_2\) iff \(\mu(\hat{t}_1)\) is smaller than \(\mu(\hat{t}_2)\) in the lexicographical order. For example, \((1) <_{o} (1,2,4) <_{o} (2)\). A pair of tuples \((\hat{t}_1, \hat{t}_2)\) for which \(\hat{t}_2 \prec_o \hat{t}_1\) is assigned zero weight, e.g., the pair \((2), (1)\) but not the symmetric pair \((1), (2)\).

For the example in Figure 8.1(a), in the first iteration of the algorithm, only four pairs of tuples get a non-zero weight: \((1,2), (1,3), (2,4), (3,4)\).

Results of match are represented by the map \(\hat{P}\) which relates matched tuples to each other (Line 5). We say that \(\hat{t}_1\) is matched with \(\hat{t}_2\) if \(\hat{P}(\hat{t}_1) = \hat{t}_2\). If a tuple \(\hat{t}\) is not matched with any element, \(\hat{P}(\hat{t})\) is null. The map is ordered by the weight of the matches, from the largest to the smallest, so that the “strongest” matches are retrieved first. For the example in Figure 8.1(a), two matches are produced in the first iteration of Algorithm 1: \(\hat{P}(\langle 3 \rangle) = \langle 4 \rangle\) and \(\hat{P}(\langle 1 \rangle) = \langle 2 \rangle\), with the weights \(\frac{2}{3}\) and \(\frac{4}{3}\), respectively. The example in Figure 8.1(c) also results in two matches: \(\hat{P}(\langle 9 \rangle) = \langle 10 \rangle\) and \(\hat{P}(\langle 7 \rangle) = \langle 9 \rangle\), with the weights \(\frac{4}{3}\) and \(\frac{2}{3}\), respectively.

Pairs of matched tuples are subject to internal processing. First, the tuples are “chained” towards possible unification that incorporates more than two tuples (Lines 10-17): if \(\hat{P}(\hat{t}_1) = \hat{t}_2\) and \(\hat{P}(\hat{t}_2) = \hat{t}_3\), the tuples are grouped into an ordered set \(c\) consisting of tuples \(\hat{t}_1, \hat{t}_2, \hat{t}_3\). More generally, for the processed pair of tuples \((\hat{t}, \hat{t}')\) and the existing chain \(c = [c_1, \ldots, c_n]\), if \(\hat{t} = c_n\), the last tuple of the pair can be appended to the chain, producing the chain \([c_1, \ldots, c_n, \hat{t}']\) (Lines 10-13). Otherwise, if \(\hat{t}' = c_1\), the first tuple of the pair can be prepended to the chain, producing the chain \([\hat{t}, c_1, \ldots, c_n]\) (Lines 14-17). For our example in Figure 8.1(c), the pair \((9), (10)\) is processed first; then the tuple \((7)\) from the pair \((7), (9)\) is prepended to it, producing the chain \((7), (9), (10)\). The chain corresponds to the tuple \((7,9,10)\) which contains all three elements.

During chaining, the algorithm checks whether the appended / prepended tuple causes the union of the chained elements to be invalid w.r.t. the validity function \(V\). Similarly, it checks whether the tuple intersects with at least
one other tuple of the chain w.r.t. their set of models: while we introduced a partial order on the set of tuples, for the pairs of matches $\hat{P}(\hat{t}_1) = \hat{t}_2$ and $\hat{P}(\hat{t}_2) = \hat{t}_3$, $\mu(\hat{t}_1) \cap \mu(\hat{t}_2) = \emptyset$ and $\mu(\hat{t}_2) \cap \mu(\hat{t}_3) = \emptyset$. In such cases, the tuple should not be added to the chain (Lines 11 and 15).

When both tuples of the matched pair do not belong to any chain, a new chain is started (Lines 18-21). That is the case for the first pair of tuples $(\langle 9 \rangle, \langle 10 \rangle)$ in the example in Figure 8.1(c), as well as for both pairs of tuples in the example in Figure 8.1(a), $(\langle 1 \rangle, \langle 2 \rangle)$ and $(\langle 3 \rangle, \langle 4 \rangle)$.

Every tuple added to a chain is removed from $\hat{P}$ (Line 9); the chaining process continues until all tuples in $\hat{P}$ are processed (Line 7).

The chaining phase is followed by the optimization phase (Line 23), in which we check whether the chaining was inefficient, i.e., that it chained “too much” and splitting a chain $c$ into smaller “sub-chains” $c_1 \ldots c_p$ improves the weight of the result: $w(c) < \sum_{1 \leq i \leq p} w(c_i)$. During the optimization step, we only verify whether the chain can be broken into one or more parts, without trying to reshuffle tuples of the chain and check their possible combinations. This heuristic is reasonable as the chains are built while putting optimally matched tuples close to each other.

Optimized chains form tuples that are used as input to the next iteration of the algorithm (Line 27) and the process continues until no further improvements can be achieved. For the example in Figure 8.1(a), the algorithm stops after the first iteration, producing tuples $\langle 1,2 \rangle$ and $\langle 3,4 \rangle$. No further unification is possible as the combination of these two tuples is invalid – it contains two elements from the same model $M_2$. The result produced by the algorithm in this case corresponds to the desired solution in Figure 8.1(b). Likewise, for the example in Figure 8.1(c), the algorithm stops after producing the chained tuple $\langle 7,9,10 \rangle$, which also corresponds to the desired solution in Figure 8.1(d).

Validity of the algorithm. By construction, the algorithm ensures generation of valid matches, per Definition 19:

(a) All tuples are valid: Pairs of tuples that share elements from the same model are assigned zero weight and thus cannot be formed (see Line 3), while the chaining step ensures that chains do not contain elements from the same model (see Lines 11 and 15). Likewise, pairs of tuples that are invalid w.r.t. the validity function $\forall$ are assigned zero weight and thus cannot be formed (see Line 3), while both the chaining and the optimization steps ensure validity of tuples that correspond to the produced chains (see Lines 11 and 15).

(b) All tuples are disjoint: In the first iteration, all input model elements belong to separate tuples. In subsequent iterations, a tuple can be added to one chain only (see Lines 9-19) and thus becomes part of only one tuple. By induction, an element cannot belong to more than one output tuple.

(c) The set $\hat{T}$ is maximal: The algorithm starts with the set of all input model elements, thus all elements are part of at least one tuple. It is not possible to expand the solution without violating the above disjointness constraint (b).

Clearly, the algorithm is heuristic and does not produce optimal results in all cases. Yet, it is reasonable to believe that it has a high chance of producing good matches because it simultaneously considers best combinations across different models, rather than limiting itself to any particular ordering. We evaluate the algorithm and compare it to subset-based approaches in the next section.
Figure 8.3: Results of comparing NwM against subset-based algorithms.
Table 8.3: Weights and execution times of \textit{NwM} compared to subset-based approaches. For each case, the results of \textit{NwM} and the best subset-based approach are boldfaced.

<table>
<thead>
<tr>
<th></th>
<th>\textit{NwM}</th>
<th>\textit{Subset-based approaches}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\textit{PW}</td>
<td>\textit{PW}_↑</td>
</tr>
<tr>
<td>Time</td>
<td>42.7s</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>Warehouse</td>
<td>1.522</td>
<td>0.973</td>
</tr>
<tr>
<td>Time</td>
<td>2.9m</td>
<td>1.4s</td>
</tr>
<tr>
<td>Random models</td>
<td>0.979</td>
<td>0.835</td>
</tr>
<tr>
<td>Time</td>
<td>1.6m</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>“Loose” scenario</td>
<td>0.980</td>
<td>0.772</td>
</tr>
<tr>
<td>Time</td>
<td>1.5m</td>
<td>&lt; 1s</td>
</tr>
<tr>
<td>“Tight” scenario</td>
<td>0.941</td>
<td>0.947</td>
</tr>
<tr>
<td>Time</td>
<td>1.6m</td>
<td>&lt; 1s</td>
</tr>
</tbody>
</table>

8.5 Evaluation

In this section, we discuss runtime and space complexity of the polynomial algorithms described in Section 8.4 and then report on an empirical evaluation of the effectiveness of these approaches.

8.5.1 Theoretical Evaluation

As can be seen in Table 8.1, the runtime complexity of our \textit{NwM} algorithm is bounded by $O(n^4 \times k^4)$, making the algorithm polynomial in both $n$ (the number of input models) and $k$ (the size of the largest input model): there are a total of $n \times k$ elements considered by the algorithm and each iteration reduces the number of input tuples by at least one. Bipartite graph matching of $n \times k$ elements in our implementation is bounded by $(n \times k)$ [96] (even though more efficient implementations are also possible [45]); the chaining phase is quadratic in the size of the input; and the optimization phase is quadratic to the maximal length of the chain. Since all tuples in a chain contain elements from distinct models, the length of the chain is bounded by $n$. The space complexity of the algorithm is bounded by the maximal number of matched pairs, that is, $n^2 \times k^2$.

The runtime complexity of all \textit{PW} algorithms is $O(n \times k^3)$: they perform up to $n$ iterations, each of which takes $O(k^3)$. The space complexity of these algorithms is $O(k^3)$, making them more efficient than \textit{NwM} both w.r.t. time and memory.

The runtime complexity of algorithms \textit{Gr3} and \textit{Gr4} is $O(n \times n^3 \times k^7)$ and $O(n \times n^3 \times k^9)$, respectively, with the space complexity of $O(1)$ for both cases (we implemented the memory efficient version of the algorithms). That is, the \textit{Gr3} algorithms have similar complexity to \textit{NwM}, while the \textit{Gr4} algorithms are more expensive.

As discussed earlier and indicated in Table 8.1, all of the above algorithms do not have a theoretical approximation factor – any selection they make can result in matches that, while reasonable in the current iteration, “block” selected elements from participating in tuples with a higher weight.
8.5.2 Empirical Evaluation

We now report on an empirical evaluation of the polynomial-time n-way merging approaches described in Section 8.4. Specifically, our aim was to answer the following research questions:

RQ1. How does the performance of \( NwM \) compare to the subset-based approaches? What are the conditions under which each algorithm is better?

RQ2. How does the size of the selected subset affect the quality of the results produced by the subset-based approaches?

RQ3. How does the order in which input models are picked affect the quality of the results produced by these approaches?

Subjects. For our evaluation, we used the Hospital and Warehouse examples from [123], described in Section 8.3.2. Before merging, we performed a simple pre-processing step on the models, unifying syntactically similar properties, e.g., “patient” and “patients”, since implementing compare algorithms is out of scope of this work. We augmented the set of subjects by 300 randomly generated models, divided into 30 sets of 10 models each. The first 10 sets of 10 models mimic the real-life examples in terms of the number of models and their sizes, the number of properties for each element, the total number of properties in all models and their distribution into shared and unshared (see Section 8.3.2). We refer to those as the Random case. Based on our experience with the Random case, and to gain deeper insights in the qualities of the evaluated algorithms, as well as the conditions that affect their performance, we generated two additional random cases, containing 10 sets of 10 models each. The cases, referred to as the “Loose” scenario and the “Tight” scenario, vary in the number of classes in each model, the total number of properties in all models, and the number of properties in each class, as discussed below. All models are available at http://www.cs.toronto.edu/~mjulia/NwM.

Methodology and Measures. We implemented all algorithms in Java and executed them on an Intel Core2Quad CPU 2.33GHz machine using JVM version 7. For fairness, we used the same implementation of compare and the same validity criteria for all algorithms (see Section 8.2.2): each element of a tuple in the solution must share at least one property with other elements of the tuple, and the more properties are shared the higher is the tuple weight.

For each algorithm, we measured the execution times and the weights of the produced solution for each of the case studies. We focused our evaluation on the matching stage rather than the perceived “appeal” of the result, which is largely domain-specific and depends on the definition of compare. Hence, we considered an algorithm to be better if it produces a solution with a higher total weight. We represented the result returned by each subset-based approach as a set of tuples, calculating the weight of each tuple and the weight of the overall solution as if they were produced by \( NwM \).

The results are summarized in Table 8.3. The first two rows show the results for the Hospital and Warehouse cases, while the other three show average results for the Random, the “Loose” scenario and the “Tight” scenario. For those cases, we merged each set of 10 models individually and averaged the results for all sets within a case. For comparison, we mark in bold the results produced by \( NwM \) as well as the best result achieved by any of the subset-based algorithms on the same case.

To compare \( NwM \) to each of the subset-based approaches further, we calculated the percentage by which it improved (or degraded) the weight of the solution, compared to the other algorithms. For example, in the Hospital case, \( NwM \) found a match with the total weight of 4.595, compared to 4.473 found by PW. This improves the matching by 2.7%. Gr4↓ performed the best of the subset-based algorithms, achieving the result of 4.566. \( NwM \)
improves that by only 0.6%. \textit{Gr4$\uparrow$} performed the worst, and \textit{NwM} improves its result by 14%. For the Warehouse case, \textit{NwM} finds a solution with the total weight of 1.522 which improves the best result of 1.126 found by \textit{PW$\downarrow$} by 35%.

For the random cases, we calculated the percentage of weight increase / decrease of each run individually and summarized the distribution of results as boxplots in Figure 8.3, separately for each case. On the horizontal axis, we distinguish the nine algorithm variants used for comparison. On the vertical, we show the percentage of weight increase / decrease produced of our algorithm compared to each variant. For instance, compared to \textit{PW}, the results achieved by \textit{NwM} in the Random case (Figure 8.3(a)) range from 5.2% to 26.9% improvement (indicated by the thin black vertical line); half of the cases fall into a range between 17.3% to 20.9% improvement (indicated by the grey box showing the upper and lower quartile). The average improvement in this case is also 17.3% (indicated by a diamond symbol) and the median is 18.2% (indicated by a horizontal line within the grey box). In addition, we also plot the results of the Hospital and the Warehouse cases (indicated by triangle and square symbols, respectively). The Hospital and the Warehouse results are also shown in Figure 8.3(b) and (c), together with the respective results for the “Loose” and the “Tight” scenarios.

\textbf{Analysis and Results.} The results show that for both the Hospital and Warehouse cases, \textit{NwM} outperforms the existing approaches w.r.t. the weight of the found solution, i.e., the relative values are positive and appear above the 0\% line in Figure 8.3. This result is also seen in the Random case (see Figure 8.3(a)): \textit{NwM} is able to achieve 30\% improvement on average compared to \textit{PW$\uparrow$} and at least 13.5\% on average compared to \textit{Gr4$\downarrow$}.

We also noticed that in the Warehouse case, \textit{NwM} was able to achieve a much more substantial improvement than in the Hospital case, with Random being “in the middle”. We analyzed the differences between these two cases in more detail. It appears they mostly differ in the total number of properties in all input models (159 properties in total in the Hospital example, compared to 338 for the Warehouse) and the average number of properties in each class (4.76 for the Hospital case and 3.67 for the Warehouse). Also, the range of model sizes differs from 18-38 classes in each model in the Hospital case to 15-44 classes in the Warehouse case. Thus, the Hospital case appears to be more “tight” – the model size distribution is smaller, there is a smaller number of properties in total, with more properties that appear in each class. The Warehouse case, on the other hand, is more “loose” w.r.t. these parameters.

We thus produced the “Loose” and “Tight” scenarios, varying the number of the above discussed parameters. Indeed, our algorithm was able to achieve much more substantial improvements in the “Loose” case (see Figure 8.3(b)): more than 40\% on average compared to \textit{PW$\uparrow$} and at least 14.3\% on average compared to \textit{Gr3$\downarrow$}. In the “Tight” case (see Figure 8.3(c)), the best improvement of 4.8\% on average was achieved compared to \textit{Gr3$\uparrow$}, while \textit{Gr4$\downarrow$} outperforms our algorithm by 1.7\%.

It is not surprising that subset-based approaches perform reasonably well in “tight” combinations, with \textit{Gr4$\downarrow$} being the best in most cases: the merged elements are more similar to each other, thus allowing these algorithms to form valid matches in each step. For example, even if a subset-based algorithm makes a “wrong” selection, the remaining tuples still have a high enough weight as their elements are close to each other. For such cases, \textit{NwM} performs similarly to the subset-based algorithms.

With respect to the execution time of the approaches, \textit{PW} algorithms are faster than the others, terminating in less than one second (see Table 8.3), as expected. Execution times for \textit{G3} ranged between 14 and 40 seconds, while for \textit{NwM} they were 43 seconds to 2.9 minutes. \textit{G4} was significantly slower than the other algorithms, taking from 4.2 minutes on the Hospital case to 27.9 minutes in the Warehouse case.

\textit{Conclusion – RQ1:} Our experiments confirm that the \textit{NwM} algorithm produces better results the subset-based
approaches in the majority of cases, especially in the more “loose” combinations. This includes the real-life models that we used as input. The weight increases achieved by the algorithm are substantial while the decreases are rare and minor. Furthermore, the running time of NwM is feasible.

Comparing the subset-based approaches to each other, it appears that the Greedy approaches perform better than Pairwise. On average, Gr4↓ outperformed the other algorithms in the Hospital and “Tight” cases, while Gr4 outperformed the others in the Random case. Gr3↓ was better than the alternative subset-based approaches in the “Loose” case. Warehouse was the only case in which PW↓ outperformed the rest of the subset-based algorithms.

To investigate whether it is beneficial to take a more “global” view, i.e., by combining input models into larger subsets, we compared Gr4↓ to the other approaches on the Hospital, Warehouse and Random cases, presenting the results in the boxplot view in Figure 8.4. The results appear inconclusive – most of the average improvements are only slightly higher than 0%, so there appears to be no benefit using this algorithm compared to the others. The figure also shows, again, that NwM performs significantly better than Gr4.

Conclusion – RQ2: Selecting larger subsets of input models, e.g., 4, to merge does not have a significant effect on the results.

For each of the subset-based approaches, we also evaluated the correlation between the quality of the produced match and the order in which the subset of input models is picked. As shown in Table 8.3, in all but two cases the strategy of combining models by size in the descending order, i.e., from the largest to the smallest, produced better results than those using the ascending order. Intuitively, we believe that handling larger models first produces a large model as a result, providing more “choices” for subsequent iterations which leads to better matches. While we saw some outliers, in the majority of cases, arranging models by size in the descending order was a more beneficial strategy.

Conclusion – RQ3: For subset-based approaches, the quality of the produced result is sensitive to the order of input models, with arranging them by size in the descending order being more beneficial.
8.5.3 Threats to Validity

Threats to external validity are most significant for our work. These arise when the observed results cannot generalize to other case studies. We attempted to mitigate these threats by using two real-life case studies and a considerable number of randomly generated models that mimic the properties of the real ones.

Using automatically generated models as a basis for evaluation is by itself a yet another threat to validity. We attempted to mitigate this by basing our generator on characteristics taken from the real-life models and using a combination of generated and real-life models for evaluation.

We experimented with a particular weight function, described in Section 8.2. Thus, the results might not generalize to other possible weight calculations. Yet, we believe that specific calculations of weights are orthogonal to our work, as they have a similar effect on all of the compared approaches.

8.6 Related Work

Numerous approaches for model merging focus on merging two inputs with each other [178, 85, 112]. Similarly, model and code differencing approaches [72, 54, 6] focus on identifying differences in two, usually subsequent, versions of the input artifacts. Some works, e.g., [93], propose techniques for detecting many-to-many matches between the elements of two input models. Our work differs from those by addressing the problem of merging \( n \) input models together, for any \( n > 2 \).

Duszynski et al. [42] emphasize the need for simultaneous analysis of multiple source code variants, identify commonality and variability of those variants, and propose an approach for comprehensible visualization of the analysis results. This work does not attempt to study and optimize the matching step though, but rather greedily finds a match for each input element.

Easterbrook et al. [44] discuss the problem of merging multiple viewpoints, showing via an empirical study that determining a good match requires an extensive discussion with domain experts. Having that discussion helps to clarify concepts and detect inconsistencies between the viewpoints. The authors also mention that tool support for comparing the viewpoints and keeping track of relationships between their elements would facilitate the viewpoint merging process. Our NwM algorithm attempts to provide such tool support; its output can be used for facilitating discussions with the relevant stakeholder.

Approximation algorithms for the weighted set packing problem, applicable for the n-way matching step, are the most relevant to our work (see also Section 8.3.1). Arkin and Hassin [7] propose an algorithm based on local search. The algorithm starts from any solution, e.g., an empty set or a solution found by Greedy algorithm. It then iteratively attempts to improve the solution by selecting \( 2 \leq s \leq n \) disjoint tuples that are not part of the it, and trying the swap them with a subset of tuples in the solution, if that increases the solution’s weight while still keeping it disjoint.

Chandra and Halldorsson [24] further improve that algorithm: instead of doing any local improvement that increases the weight of the solution, the authors propose to find and use the best possible improvement. The algorithm also assumes that all improvements are of size \( n \), which makes it more expensive computation-wise.

Berman [17] proposes a yet another improvement to the above algorithm: instead of performing the swap with the best possible improvement, the algorithm performs a swap if the square of tuple weights is improved. Again, this increases the computational complexity of the algorithms.

The exact approximation factors and time complexity of the above algorithms are shown in Table 8.1. As discussed in Section 8.3.2, these algorithms do not scale for more than a small number of small models, while we aim to provide a practical solution that can be applied for merging real-life software models.
8.7 Summary

In this chapter, we extended the approach presented in Chapter 7 by refining the model merging problem to consider $n$ inputs – a common case when combining multiple products into an SPL representation. We focused on the most challenging step of the merging process – matching – and showed that the problem is NP-hard. We surveyed and evaluated state-of-the-art approximations developed in the literature, as well as current practices of incrementally merging $n$ input models together in smaller subsets. Based on this experience, we proposed a novel, polynomial-time heuristic algorithm $NwM$ that considers all $n$ input models simultaneously. We evaluated our approach on a large set of benchmarks, including two real-life examples, and showed that it achieves substantial improvements over approaches that merge input models in subsets, without a substantial degradation in performance.
Chapter 9

Conclusions and Future Work

Numerous Software Product Line Engineering (SPLE) approaches promise to ease the product line development and maintenance effort, reduce costs and improve quality of the developed products. Yet, the adoption of such approaches in industry is still limited. Instead, products are often established ad-hoc, e.g., by copying existing variants and modifying them to fit the requirements of a new customer or market. In this thesis, we focused on exploring causes for this lack of adoption and suggesting approaches for improving development experience in organizations that employ cloning to realize their product lines.

9.1 Summary

In Chapter 2, we described an exploratory study that we conducted in order to investigate the cloning culture in six industrial team that use cloning to realize their product lines. The study revealed that, despite its negative reputation, cloning is still perceived as a favorable and natural reuse approach by the majority of practitioners in the studied companies. Our interviewees stated that cloning is a rapidly available mechanism, which allows them to start from an already specified and verified functionality, while leaving the freedom and independence to make any necessary modifications to it. However, in the long run, cloning might result in difficulties to perform maintenance and evolution tasks.

Based on the observations collected during the study, we discussed issues preventing the adoption of SPLE approaches in organizations that employ cloning. These include lack of planning; lack of evidence and clear measurement models quantifying improvements achieved by SPLE adoption; organization-level difficulties related to facilitating the adoption; and technical difficulties related to performing the transition to a structured SPLE approach itself. We noted that the overhead related to the transition tasks should be comparably lower than the overhead related to the management of existing clones, as the desire of practitioners to abandon cloning practices is not obvious. Moreover, exploring directions that improve cloning practices rather than attempt to eliminate them might be beneficial.

Subsequent to Chapter 3 were we fixed terminology and formalized the notations that we use, Chapter 4 focused on systematizing technical support required by companies that use cloning. Our own experience with partner companies and related reports from the literature showed that even if a company decides to transform its development practices and to adopt a structured SPLE approach, the transition process is a long effort which usually takes several years. During that time, existing products are still developed and maintained as clones. As such, helping organizations to improve their cloning experience is essential.
We stated that an efficient management of clones relies on two directions: the first focuses on refactoring cloned variants into a single-copy product line representation (a.k.a. merge-refactoring), while the second – on improving the development experience when maintaining existing clones (a.k.a. clone-based SPLE). In the remainder of the chapter, we proposed a framework that consists of seven conceptual operators which can be composed to realize development activities related to both these directions. Further, we outlined options for implementing these operators, identified obvious gaps and discussed benefits of the operator-based view.

In Chapter 5, we grounded the proposed framework on empirical evidence by analyzing in detail development activities in three large industrial organizations that use (or used) cloning to develop their product lines. We considered both the case when cloned products are refactored into a single-copy software product line representation and the case when they are maintained as distinct clones. We broke the development activities of the studied organizations into individual steps and showed that these steps can be expressed as instances of the operators, thus validating the conceptual framework proposed by our work. We also discussed organization-specific automation opportunities and presented the limitations of our study.

In Chapters 6-8, we contributed to the implementation of some of the operators. Specifically, Chapter 6 focused on the findFeatureImplementation operator (a.k.a. feature location) which allows establishing traceability links between a feature and the artifacts that implement it. Even though there exist more than 20 different feature location techniques, most of them are tailored for specific cases, without giving an accurate specification of the kind of features they detect. In the general case, these techniques are highly imprecise. We then focused on a subset of the general feature location problem: detecting distinguishing product line features – those that are present in one product variant but are absent in another. Distinguishing features are of particular importance in SPLE scenarios as they correspond to variable SPL functionality. As such, they are candidates for selective propagation from one program to another and require attention when refactoring program variants into single-copy SPL representations.

We proposed a set of heuristics for improving the accuracy of existing feature location techniques when locating distinguishing features. Our heuristics are based on diff sets – the information obtained when comparing program variants to each other. We empirically studied the nature of the proposed heuristics, and evaluated the contribution of each heuristic as well as their combination, on a large number of case studies. We showed that in most cases the combination of the heuristics enables substantial improvements in the accuracy of feature location (up to several hundred percent), while the cases of reduced accuracy are rare and minor.

Chapter 7 contributed an implementation for another operator: merge. Our implementation can be applied to UML models containing class and statechart diagrams – a common choice for companies that use commercial tools such as IBM Rhapsody to automatically generate fully-functional code from their models. It merge-refactors a set of UML product variants, producing an annotative SPL representation. Our approach was based on comparing and matching artifacts of existing variants and then composing those deemed similar while explicitly annotating those that vary. We provided a formal specification of the merge-refactoring algorithm called merge-in, and stated sufficient conditions of model compare, match and compose operators that allow the application of merge-in. Based on these, we formally proved that the merge-in algorithm is safe and semantically correct, i.e., the resulting SPL representation is equivalent to the set of original products.

We also pointed out that merge-in can generate several semantically correct, yet syntactically different SPL representations. Depending on the goal of the refactoring, one result can be preferred to another. We thus proposed to capture the goal using a syntactic quality function and use that function to guide the merge strategy. We used the merge-in algorithm as the foundation of a quality-based merge-refactoring approach for UML models that guides the refactoring process towards a result that best fits the user’s intention. We implemented the proposed approach and evaluated it on a set of case-studies, including a large-scale example contributed by an industrial partner. Our
experience showed that the framework helps producing the desired result.

Chapter 8 took a closer look at the problem of merging multiple models. Since most of the existing approaches are applicable to merging two inputs only, the common way to deal with multiple models is to combine them incrementally in a pairwise manner. More generally, models can be combined in small subsets – an approach that we referred to as subset-based.

We showed that subset-based approaches are less efficient than considering all input models simultaneously. We also showed that the latter is equivalent to the known and extensively studied NP-hard problem of weighted set packing. We then evaluated existing approximations of the weighted set packing problem and concluded that they do not scale for models of a realistic size. We thus developed our own algorithm, \(NwM\), that considers all input models simultaneously and evaluated the algorithm on a large set of cases, including two real-life examples. Our evaluation showed that the proposed algorithm achieves substantial improvements over approaches that merge input models in subsets, without sacrificing performance.

9.2 Conclusions

This thesis took a systematic approach to identifying and organizing the technical support required by organizations that rely on cloning to realize their product lines. We empirically investigated the cloning culture in industry, performing both qualitative studies that examine practitioners’ viewpoints and perceptions, and studies that involve detailed analysis of development artifacts and activities. Based on the empirically collected knowledge, we built the foundations for the management of cloned product variants, contributing a framework that specifies a basic set of operators required for variant maintenance and evolution. We showed that our operators provide the necessary building blocks that can be composed to realize complex scenarios related to the management of clones, during both the unification and the maintenance of existing variants.

Identifying conceptual operators and their parameters allows classifying existing solutions, comparing them to each other, and identifying gaps that are still to be addressed. We thus believe that our work helps transforming the effort of improving reuse practices in organizations that employ cloning from largely opportunistic to predicted, controlled and reproducible. In particular, it allows organizations to locate and reuse existing work, estimate the investment required to implement the missing functionality, and share experiences with each other using a common vocabulary. It also allows researchers and solution developers to focus their work on contributing the functionality that is indeed missing.

Our work also highlighted the importance of building an efficient management infrastructure on top of existing cloned variants rather than trying to eliminate them: in reality, companies still employ cloning due to its benefits such as simplicity, availability and developer independence. We also suggested developing SPLE approaches that are based on cloning. This direction was recently augmented by other research groups as well [3].

Contributing to the implementation of the operators, we developed an approach for locating features in a family of cloned product variants. Following our original publication on this topic [139], this problem was further explored by several other research groups [184, 179, 101, 169]. We also developed an approach for merge-refactoring multiple UML models containing class and statechart diagrams into an annotative SPL representation. Finally, we redefined the problem of model merging to focus on multiple inputs simultaneously rather than processing them in any particular order and contributed an algorithm that is superior to other practical approaches used for that purpose in terms of its accuracy.
9.3 Future Research Directions

We divide the discussion on future research directions into short- and long-term goals. The former describe “immediate” improvements for the techniques developed in this thesis. The latter focus on “larger” and longer-term goals that contribute to the field.

9.3.1 Short-Term Goals

The work on feature location in Chapter 6 can further be extended by developing more sophisticated heuristics that are based on the information available when comparing the variant that contain the feature of interest to the one that does not, i.e., the information stored in diff sets. For example, one can leverage the knowledge about diff set connectivity – whether elements in a diff set are “discrete” or form strongly connected components, to further influence the program traversal process. It might also be interesting to experiment with comparing multiple variants to each other, producing more sophisticated diff sets that can further partition the program of interest. Developing diff set-based heuristics to the improve effectiveness of additional types of feature location techniques, e.g., those based on dynamic program analysis, could also be beneficial.

The merge operator developed in Chapter 7 can further be extended by exploring refactoring techniques that allow to create new products by “mixing” features from different original variants. Enhancing model merging techniques with additional capabilities, e.g., using code-level clone detection techniques for comparing statechart actions and activities, as well as devising alternative methods of calculating graph similarity, e.g., by counting the number of identical or similar sub-graphs, also seems to be valuable. Furthermore, it is desirable to improve match algorithms by allowing the user to affect the results of this function, e.g., by setting negative matches.

In addition, enhancing our understanding of product line quality considerations, e.g., by performing user studies for evaluating the quality of annotative SPL models can help with assessing different SPL representations, produced either automatically or manually. The quality functions proposed in Chapter 7 can then be extended to consider additional quality attributes, allow the user to set and/or interactively choose different quality goals for different regions within the model, incorporate user feedback and more. Future research is needed though to investigate the translation of a “semantic” notion of quality into a set of syntactic indicators. Exploring the use of more advanced optimization techniques, such as cross-entropy \[147\], for adjusting compare and match parameters based on the quality function is also a subject for possible future work.

Finally, it is worthwhile to explore additional heuristics that could further improve the quality of the merge produced by the \( NwM \) algorithm presented in Chapter 8. For example, one could experiment with the idea of reshuffling chained tuples (rather than only breaking the chain into pieces) or even reshuffling elements from distinct tuples of the chain. Augmenting the work by integrating it with domain-specific compare and compose approaches and by extending it beyond a numerical definition of matching quality, e.g., providing means to differentiate between match solutions with identical total weight, could also be desired. Further analyzing subset-based approaches, e.g., comparing all input models to each other and merging them in the order induced by an algorithm for finding a spanning tree, is a yet another direction for possible future work. Enhancing subset-based approaches with an option to perform backtracking, thus breaking their “sequentiality”, could also be beneficial. Future research is needed to develop such algorithms and compare them to \( NwM \).

9.3.2 Long-Term Goals

Taking a global perspective, we see three main directions for continuing the effort started in this work:
1. Economic Effectiveness of SPLE.

As mentioned above, very little quantitative data is available to back up the claimed benefits of SPLE approaches – improved time-to-marked and product quality, better market penetration and more. The majority of existing reports on successful adoption of SPLE practices, including those published in the SPLC Hall of Fame\(^1\), present mostly qualitative data that indicates or hints towards possible improvements. There are also no reports comparing the effort involved in maintaining a managed SPL versus an ad-hoc (e.g., cloned) one. It is of a little surprise then that “convincing” industrial organizations to convert their development practices to follow SPLE is a challenging task. Furthermore, there is usually no way to compare the situational context in which product line approaches have been introduced, making it difficult to know if the approaches being touted are even relevant to a specific organization. All these factors impede adoption of SPLE in industry and also prevent a realistic comparison of numerous existing approaches to each other.

As part of future work, it is essential to engage the community in a more rigorous measurement and reporting system for quantifying business benefits associated with the introduction of SPLE approaches and measuring the effort of a transition from ad-hoc to well-managed reuse. Approaches and financial models for tackling technical debt can serve as an inspiration here [34, 94]. Comparing different transition strategies, e.g., a bottom-up one that starts from analyzing and re-engineering code artifacts to a top-down strategy that starts from documentation and product requirements, would be beneficial. Furthermore, quantifying business benefits (or losses) associated with SPLE adoption and the context in which SPLE was applied will also allow comparing different SPLE technique to each other, assisting practitioners in making educated decisions when choosing an approach that best fits their goals and requirements. Such efforts are expected to benefit researchers, solution developers and industrial organizations by allowing them to understand, measure and report on the return on investment when shifting to SPLE.

2. Clone-Based SPLE Approaches.

Our study on cloning practices in Chapter 2 highlighted several qualities of cloning that are appealing to practitioners, such as availability and independence of developers. Leveraging these qualities and developing structured SPLE approaches that are based on cloning (rather than trying to eliminate cloning) appears to be a direction that is worth investigating further. While we started to explore the space of clone-based SPLE approaches and defined the initial set of operators required for their management, this set is most likely incomplete and a more focused view on clone-based SPLE practices is required.

In addition, in practice, most clone variants are maintained as branches/streams in Software Configuration Management (SCM) systems. Integrating SPLE management mechanisms on top of existing SCM systems seems to be a promising direction, which we only started to explore [146]. SCM-based approaches will allow the user to reason about the developed product line in terms of features rather than individual code changes made in distinct branches, will detect and maintain semantic dependencies and inconsistencies in implementations of features and will contain feature provenance information.

Earlier works that focused on improving the maintenance of SCM branches might become relevant for this task. For example, Sarma et al. [151, 65] suggest promoting team awareness by sharing information about changes and potential conflicts across branches of an SCM system. Both Gulla et al. and Zeller et al. propose to capture composition constraints between different versions of software components that are stored in an SCM system and to allow assembling a configuration containing just those component versions that satisfy the constraints [64, 182]. It might be beneficial to explore the applicability of these works to the SPLE context.


\(^1\)http://splc.net/fame.html.
In this thesis, we showed that the current set of operators is reasonable for expressing development activities related to clone management in the studied organizations. Yet, additional effort is required to refine the set of operators and their interfaces based on a larger set of case studies. Identifying and classifying contexts in which the operators are automatable as well as quantifying the cost of providing such automation is also still needed. Future studies addressing these questions might be of a value.

A substantial amount of work also remains in devising techniques that realize the operators, in cases where such realizations are missing. The realization of some operators, such as findFeatures, was, to the best of our knowledge, never attempted. A possible realization of this operator can, for example, rely on artifact comparison, partitioning, and summarization techniques. A realization of dependsOn? for code-level artifacts – yet another one where support is missing – can employ code analysis approaches, such as slicing, data-flow analysis and symbolic execution. While feature location is largely studied for code, similar techniques for model-level artifacts are hardly investigated. We recently performed some initial attempts to close this gap [162] by proposing an approach for feature-based model splitting, but further investigation of the problem is still required. “Smart” implementations of the operators that can work incrementally, to help address incremental changes in the development artifacts, as well as implementations that are able to simultaneously consider artifacts of several types, would also be highly useful.

Our framework attempts to capture the “essence” of SPL by allowing to detect product line features, their relationships and their corresponding implementation artifacts. We thus believe that a similar set of operators would also be adequate for merge-refactoring approaches that aim at producing a compositional product line architecture [82] rather than an annotative one that we assumed in our work. Yet, additional research is required to confirm or refute this assumption.

We invite the product line community to join our effort and further improve the work started in this thesis by refining the set of the operators, studying their applicability, organizing existing work around the established common terminology and providing missing solutions.
Bibliography


Bibliography


Appendix A

A Survey of Feature Location Techniques

In this appendix, we provide an overview of feature location techniques, based on our publication in [141]. Rajlich and Chen [27] represent a feature (a.k.a. a concept) as a triple consisting of a name, intension and extension. The name is the label that identifies the feature; intension explains the meaning of the feature; and extension is a set of artifacts that realize the feature. Location: intension $\rightarrow$ extension, is identified by the authors as one of the six fundamental program comprehension processes. Its application to features is the subject of this survey.

In the remainder of the chapter, we illustrate the surveyed concepts using a problem of locating the automatic save file feature, previously studied in [134], in the code of the Freemind open source mind-mapping tool. A snippet of Freemind’s call graph is shown in Figure A.1. Shaded elements in the graph contribute to the implementation of the automatic save file feature – they are the feature extension which we want to locate. Feature intension can be given, for example, by the natural language query “automatic save file”, describing the feature.

The feature is mainly implemented by two methods of the MindMapMapModel sub-class doAutomaticSave: the constructor and the method run (elements #1 and #2). doAutomaticSave class is initiated by the MindMapMapModel’s constructor (element #4), as shown in Figure A.2. The constructor assigns values to several configuration parameters related to the automatic save file function and then registers the doAutomaticSave class on the scheduling queue. This initiates the class’s run method (element #1) which subsequently calls the saveInternal method (element #3) responsible for performing the save operation.

Obviously, not all program methods contribute to the automatic save file feature. For example, element #3 also initiates a call to FreeMindNodeModel’s save( Writer, MindMapMapModel) method (element #5), which, in turn, calls element #6 – save(Writer, MindMapMapModel). Both of these methods are irrelevant to the specifics of the automatic save file implementation. Element #3 itself is called by element #7 (MindMapMapMode’s save(File) method), which is called by element #8 (MindMapController’s actionPerformed(ActionEvent)). These methods are also not relevant to the feature implementation because they handle a user-triggered save operation instead of automatic save. In fact, element #8 initiates calls to an additional twenty-four methods, all of which are irrelevant to the implementation of the feature. In Figure A.1, irrelevant methods are not shaded.

While all feature location approaches share the same goal – establishing traceability between a specific feature of interest specified by the user (feature intension) and the artifacts implementing that feature (feature extension), they differ substantially in the underlying design principles, as well as in assumptions they make on their input (representation of the intension). In this chapter, we provide an in-depth description of twenty-four existing feature

---

1The rest of the thesis used the terms label, description and implementation to refer to the corresponding concepts.
2http://freemind.sourceforge.net
3We denote features by a sans-serif font, place natural language queries “in quotes”, and denote code elements by a monospaced font.
A systematic literature survey of eighty-nine articles related to feature location is available in [39]. That survey provides a broad overview of existing feature definition and location techniques, techniques for feature representation and visualization, available tools and performed user studies. The purpose of that work is organizing, classifying and structuring existing work in the field and discussing open issues and future directions. Even though 22 out of the 24 techniques surveyed here are covered by [39], our work has a complementary nature. We focus only on automated feature location techniques while providing insights about the implementation details, exemplifying the approaches and discussing how to select one in real-life settings. The intended audience of our survey is practitioners aiming to apply a feature location technique for establishing traceability between the features of their products and the implementation of these features. As such, these practitioners have to understand the implementation details and properties of the available approaches in order to choose one that fits their needs.

The rest of the chapter is organized as follows. We start by introducing basic technologies used by several feature location techniques in Section A.1. Section A.2 introduces the classification that we use for the surveyed feature location techniques. A detailed description of the techniques themselves is provided in Section A.3. We discuss criteria used when selecting a feature location technique in Section A.4. Section A.5 concludes the survey.

A.1 Basic Underlying Technologies

In this section, we introduce basic technologies commonly used by feature location techniques, describe each technology and demonstrate it on the example in Figure A.1.

A.1.1 Formal Concept Analysis (FCA)

Formal Concept Analysis (FCA) [57] is a branch of mathematical lattice theory that provides means to identify meaningful groupings of objects that share common attributes. Groupings are identified by analyzing binary relations between the set of all objects and all attributes. FCA also provides a theoretical model to analyze hierarchies of these identified groupings.
The main goal of FCA is to define a concept as a unit of two parts: extension and intension\(^4\). The extension of a concept covers all the objects that belong to the concept, while the intension comprises all the attributes, which are shared by all the objects under consideration. In order to apply FCA, the formal context of objects and their respective attributes is necessary. The formal context is an incidence table indicating which attributes are possessed by each object. An example of such a table is shown in Figure A.3, where objects are names of methods in Figure A.1 and attributes are individual words obtained by tokenizing and lower-casing these names. For example, object \(o_1\) corresponds to element #1 in Figure A.1 and is tokenized to attributes automatic, do, map, mind, model, run, save, which are “checked” in Figure A.3.

From the formal context, FCA generates a set of concepts where every concept is a maximal collection of objects that possess common attributes. Figure A.4(a) shows all concepts generated for the formal context in Figure A.3.

Formally, given a set of objects \(O\), a set of attributes \(A\), and a binary relationship between objects and attributes \(R\), the set of common attributes is defined as \(\sigma(O) = \{a \in A \mid (o, a) \in R \forall o \in O\}\). Analogously, the set of common objects is defined as \(\rho(O) = \{o \in O \mid (o, a) \in R \forall a \in A\}\). For example, for the relationship \(R\) encoded in Figure A.3, \(\sigma(o_4) = \{\text{map}, \text{mind}, \text{model}\}\) and \(\rho(\text{automatic}, \text{do}) = \{o_1, o_2\}\).

A concept is a pair of sets \((O, A)\) such that \(A = \rho(O)\) and \(O = \sigma(A)\). \(O\) is considered to be the extension of the concept and \(A\) is the intension of the concept. The set of all concepts of a given formal context forms a partial order via the superconcept-subconcept ordering \(\leq\): \((O_1, A_1) \leq (O_2, A_2) \iff O_1 \subseteq O_2\), or, dually, \((O_1, A_1) \leq (O_2, A_2) \iff A_2 \subseteq A_1\).

The set of all concepts of a given formal context and the partial order \(\leq\) form a concept lattice, as shown in Figure A.4(b). In our example, this lattice represents a taxonomy of tokens used for naming the methods – from the most generic used by all methods (the root element \(c_1\), which represent the term mind used in all names) to the more specific names depicted as leaves (e.g., \(c_6\) which represents unique terms action, controller and performed used in the name of element #8).

\(^4\)These are not to be confused with the extension and intension of a feature.
A.1.2 Latent Semantic Indexing (LSI)

Latent Semantic Indexing (LSI) [98] is an automatic mathematical/statistical technique that analyzes the relationships between queries and passages in large bodies of text. It constructs vector representations of both a user query and a corpus of text documents by encoding them as a term-by-document co-occurrence matrix. Each row in the matrix stands for a unique word, and each column stands for a text passage or a query. Each cell contains the frequency with which the word of its row appears in the passage denoted by its column.

\[
A = \begin{pmatrix}
d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 \\
\end{pmatrix}
\]

\[
q = \begin{pmatrix}
g_1 \\
\end{pmatrix}
\]
Figure A.6: Vectorial representation of the documents and the query in Figure A.5.

Figure A.5 shows such an encoding for the example in Figure A.1, where "documents" are method names, the query “automatic save file” is given by the user, and the set of all terms is obtained by tokenizing, lower-casing and alphabetically ordering strings of both the documents and the query. In Figure A.5, matrix $A$ represents the encoding of the documents and matrix $q$ represents the encoding of the query. Vector representations of the documents and the query are obtained by normalizing and decomposing the term-by-document co-occurrence matrix using a matrix factorization technique called singular value decomposition [98]. Figure A.6 shows the vector representation of the documents $d_1 \ldots d_8$ and the query $q$ in Figure A.5 in a three dimensional space.

The similarity between a document and a query is typically measured by the cosine between their corresponding vectors. The similarity increases as the vectors point “in the same general direction”, i.e., as more terms are shared between the documents. For the example in Figure A.6, document $d_2$ is the most similar to the query, while $d_8$ is the least similar. The exact similarity measures between the document and the query, as calculated by LSI, are summarized in Table A.1. It is common to consider documents with positive similarity values as related to the query of interest (i.e., $d_1$, $d_2$, $d_5$ and $d_6$ in our example), while those with negative similarity values (i.e., $d_3$, $d_4$, $d_7$ and $d_8$) – as unrelated.

Table A.1: Similarity of the documents and the query in Figure A.5 as calculated by LSI.

<table>
<thead>
<tr>
<th></th>
<th>$d_1$</th>
<th>$d_2$</th>
<th>$d_3$</th>
<th>$d_4$</th>
<th>$d_5$</th>
<th>$d_6$</th>
<th>$d_7$</th>
<th>$d_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_1$</td>
<td>0.6319</td>
<td>0.8897</td>
<td>-0.2034</td>
<td>-0.5491</td>
<td>0.2099</td>
<td>0.2099</td>
<td>-0.1739</td>
<td>-0.6852</td>
</tr>
</tbody>
</table>

A.1.3 Term Frequency - Inverse Document Frequency Metric

Term frequency – inverse document frequency (tf-idf) is a statistical measure often used by IR techniques to evaluate how important a term is to a specific document in the context of a set of documents (corpus). Intuitively, the more frequently a term occurs in the document, the more relevant the document is to the term. That is, the relevance of a specific document $d$ to a term $t$ is measured by document frequency ($tf(t,d)$). For the example in Figure A.5 where “documents” are names of methods in Figure A.1, the term save appears twice in $d_2$, thus $tf(\text{save}, d_2) = 2$.

The drawback of term frequency is that uninformative terms appearing throughout the set $D$ of all documents can distract from less frequent, but relevant, terms. Intuitively, the more documents include a term, the less this term discriminates between documents. The inverse document frequency, $idf(t)$, is then calculated as follows: $idf(t) = \log\left(\frac{|D|}{|\{d \in D | t \in d\}|}\right)$. The tf-idf score of a term w.r.t. a document is calculated by multiplying its $tf$ and
idf scores: $\text{tf-idf}(t, d) = \text{tf}(t, d) \times \text{idf}(t)$. In our example, $\text{idf}(\text{save}) = \log(\frac{8}{5}) = 0.12$ and $\text{tf-idf}(\text{save, } d_2) = 2 \times 0.12 = 0.24$.

Given a query which contains multiple terms, the $\text{tf-idf}$ score of a document with respect to the query is commonly calculated by adding $\text{tf-idf}$ scores of all query terms. For example, the $\text{tf-idf}$ score of $d_2$ with respect to the query “automatic save file” is 1.44, while $d_3$ score with respect to the same query is 0.12.

A.1.4 Hyper-link Induced Topic Search (HITS)

Hyper-link Induced Topic Search (HITS) is a page ranking algorithm for Web mining introduced by Kleinberg [89]. The algorithm considers two forms of web pages – hubs (pages which act as resource lists) and authorities (pages with important content). A good hub points to many authoritative pages whereas a good authority page is pointed to by many good hub pages.

The HITS algorithm operates on a directed graph, whose nodes represent pages and whose edges correspond to links. Authority and hub scores for each page $p$ (denoted by $A_p$ and $H_p$, respectively) are defined in terms of each other: $A_p = \sum\{q \mid q \text{ points to } p\} H_q$ and $H_p = \sum\{q \mid p \text{ points to } q\} A_q$. The algorithm initializes hub and authority scores of each page to 1 and performs a series of iterations. Each iteration calculates and normalizes the hub (authority) value of each page. It does so by dividing the value by the square root of the sum of squares of all hub (authority) values for the pages it points to (pointed by). The algorithm stops when it reaches a fixpoint or a maximum number of iterations.

When applied to code, HITS scores methods in a program based on their “strength” as hubs – aggregators of functionality, i.e., methods that call many others, and authorities – those that implement some functionality without aggregation. For the example in Figure A.1, elements #2 and #6 are authorities as they do not call any other methods and thus their hub score is 0. Elements #1 and #8 are hubs as they are not called by other methods. Thus, their authority score is 0. Elements #3 and #4 get a higher authority score than other elements as they are called by two methods each, while elements #7 and #8 get a higher hub score than the rest as they call two methods each.

A.2 Classification and Methodology

In this section, we discuss the classification of feature location techniques that we use for organizing our survey. We also discuss main properties that we highlight for each technique.

Primarily, feature location techniques can be divided into dynamic which collect information about a program at runtime, and static which do not involve program execution. The techniques also differ in the way they assist the user in the process of interpreting the produced results. Some only present an (unsorted) list of artifacts considered relevant to the feature of interest; we refer to these as plain output techniques. Others provide additional information about the output elements, such as their relative ranking based on the perceived relevance to the feature of interest or automated and guided output exploration process which suggests the order and the number of output elements to consider; we refer to these as guided output techniques. Figure A.7 presents the surveyed techniques, dependencies between them and their primary categorization.

Feature location approaches can rely on program dependence analysis (PDA) that leverages static dependencies between program elements; information retrieval (IR) techniques – in particular, LSI, tf-idf and others, that leverage information embedded in program identifier names and comments; change set analysis that leverages historical information and more. While dynamic approaches collect precise information about the program execution, they are safe only with respect to the input that was actually considered during runtime to gather the information,
Figure A.7: Surveyed techniques and their categorization.

and generalizing from this data may not be safe [92]. In addition, while generally a feature is a realization of a system requirement – either functional or non-functional – executable test-cases or scenarios can exhibit only functional requirements of the system that are visible at the user level. Thus, dynamic feature location techniques can detect only functional features. On the other hand, static approaches can locate any type of feature and yield safe information, but because many interesting properties of programs are statically undecidable in general, static analysis is bound to approximate solutions which may be too imprecise in practice. Dynamic analysis yields “under-approximation” and thus might suffer from many false-negative results while static analysis yields “over-approximation” and thus might have many false-positives. In order to find a middle ground, hybrid approaches combine several techniques.

Based on the chosen implementation technique, the analyzed program can be represented as a program dependence graph (PDG), a set of text documents representing software elements, an instrumented executable that is used by dynamic techniques and more. Figure A.8 provides detailed information about each of the surveyed techniques, listing its underlying technology, the chosen program representation, the type of user input, as well as the amount of required user interaction, ranging from low (denoted by ‘+’) to high (denoted by ‘+++’).
### Appendix A. A Survey of Feature Location Techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Underlying Technology</th>
<th>Program Representation</th>
<th>Input</th>
<th>User Interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chen et al. 2000 [27] (CLDS)</td>
<td>PDA</td>
<td>PDG</td>
<td>method or global variable</td>
<td>+++</td>
</tr>
<tr>
<td>Walkinshaw et al. 2007 [172]</td>
<td>PDA</td>
<td>call graph</td>
<td>two sets of methods</td>
<td>+</td>
</tr>
<tr>
<td>Shepherd et al. 2007 [156]</td>
<td>PDA, NLP</td>
<td>AOIG</td>
<td>query</td>
<td>++</td>
</tr>
<tr>
<td>Zhao et al. 2006 [183] (SNIAFL)</td>
<td>TF-IDF, vector space model, PDA</td>
<td>BRCG</td>
<td>set of queries</td>
<td>+</td>
</tr>
<tr>
<td>Robillard et al. 2008 [133]</td>
<td>clustering algorithms</td>
<td>change history transactions</td>
<td>set of elements</td>
<td>+</td>
</tr>
<tr>
<td>Trifu 2009 [168]</td>
<td>PDA</td>
<td>concern graph</td>
<td>set of variables</td>
<td>+++</td>
</tr>
<tr>
<td>Robillard et al. 2005 [132]</td>
<td>PDA</td>
<td>PDG</td>
<td>set of methods and global variables</td>
<td>++</td>
</tr>
<tr>
<td>Saul et al. 2007 [152]</td>
<td>PDA, web-mining algorithm</td>
<td>call graph</td>
<td>method</td>
<td>++</td>
</tr>
<tr>
<td>Marcus et al. 2004 [105, 106]</td>
<td>LSI</td>
<td>text docs for software elements</td>
<td>query</td>
<td>+</td>
</tr>
<tr>
<td>Poshyvanyk et al. 2007 [121]</td>
<td>LSI, FCA on retrieved docs</td>
<td>text docs for software elements</td>
<td>query</td>
<td>+</td>
</tr>
<tr>
<td>Shao et al. 2009 [155]</td>
<td>LSI, PDA</td>
<td>call graph, text docs for software elements</td>
<td>query</td>
<td>+</td>
</tr>
<tr>
<td>Hill et al. 2007 [69] (Dora)</td>
<td>PDA, TF-IDF</td>
<td>call graph, text docs for software elements</td>
<td>method, query</td>
<td>+</td>
</tr>
<tr>
<td>Chen et al. 2001 [25] (CVSSearch)</td>
<td>textual search</td>
<td>lines of code, CVS comments</td>
<td>query</td>
<td>+</td>
</tr>
<tr>
<td>Wilde et al. (1995) [176]</td>
<td>trace analysis</td>
<td>executable</td>
<td>set of test cases</td>
<td>+++</td>
</tr>
<tr>
<td>Wong et al. 1999 [177]</td>
<td>trace analysis</td>
<td>executable</td>
<td>set of test cases</td>
<td>+++</td>
</tr>
<tr>
<td>Eisenbarth et al. 2003 [48]</td>
<td>trace analysis (FCA), PDA</td>
<td>executable, PDG</td>
<td>set of scenarios</td>
<td>+++</td>
</tr>
<tr>
<td>Koschke et al. 2005 [92]</td>
<td>trace analysis (FCA), PDA</td>
<td>executable, statement dependency graph</td>
<td>set of scenarios</td>
<td>+++</td>
</tr>
<tr>
<td>Asadi et al. 2010 [8]</td>
<td>trace analysis, LSI, genetic optimization</td>
<td>executable, text docs for methods</td>
<td>set of scenarios</td>
<td>++</td>
</tr>
<tr>
<td>Eisenberg et al. 2005 [49]</td>
<td>trace analysis</td>
<td>executable</td>
<td>set of test cases</td>
<td>++</td>
</tr>
<tr>
<td>Poshyvanyk et al. 2007 [120]</td>
<td>trace analysis, LSI</td>
<td>executable, text docs for methods</td>
<td>set of scenarios, query</td>
<td>+++</td>
</tr>
<tr>
<td>Liu et al. 2007 [102]</td>
<td>trace analysis, LSI</td>
<td>executable, text docs for methods</td>
<td>scenario, query</td>
<td>+</td>
</tr>
<tr>
<td>Rohatgi et al. 2008 [135]</td>
<td>trace analysis, impact analysis</td>
<td>executable, class dependency graph</td>
<td>set of scenarios</td>
<td>++</td>
</tr>
<tr>
<td>Eaddy et al. 2008 [43] (Cerberus)</td>
<td>PDA, trace analysis, TF-IDF, vector space model</td>
<td>PDG, executable, text docs for software elements</td>
<td>set of queries, set of scenarios</td>
<td>+++</td>
</tr>
<tr>
<td>Reveille et al. 2010 [128]</td>
<td>trace analysis, LSI, web-mining algorithm</td>
<td>executable, text docs for methods</td>
<td>scenario, query</td>
<td>+</td>
</tr>
</tbody>
</table>

Figure A.8: Underlying technology, program representation and input type of the surveyed techniques.

#### A.3 Feature Location Techniques

In this section, we describe automated feature location techniques from the literature. As discussed earlier, we focus on those techniques that assist the user with feature location rather than feature definition or visualization. Static approaches (those that do not require program execution) are described in Section A.3.1; dynamic are in Section A.3.2.
A.3.1 Static Feature Location Techniques

In this section, we describe techniques that rely on static program analysis for locating features in the source code.

Plain Output

Chen et al. [27] present one of the earliest static computer-assisted feature location approaches based on program dependence analysis (PDA). The analyzed program is represented as a program dependence graph (PDG) whose nodes are methods or global variables and edges are method invocations or data access links (the paper refers to the PDG as the abstract system dependence graph). Given an initial element of interest – either a function or a global variable – the approach allows the user to explore the PDG interactively, node-by-node, while storing visited nodes in a search graph. The user decides whether the visited node is related to the feature and marks related nodes as such. The process stops when the user is satisfied with the set of found nodes, and outputs the set of relevant nodes aggregated in the search graph. For the example in Figure A.1, the system generates the depicted call graph from the source code and interactively guides the user through its explanation. The technique relies on extensive user interaction (denoted by ‘+++’ in Figure A.8), and thus provides the user with “intelligent assistance” [22] rather than being a heuristic-based technique aiming to determine relevant program elements automatically.

Walkinshaw et al. [172] provide additional automation to the feature location process based on PDA. The analyzed program is represented as a call graph – a subgraph of PDG containing only methods and method invocations. As input, the system accepts two sets of methods: landmark – thought to be essential for the implementation of the feature, and barrier – thought to be irrelevant to the implementation. For the example in Figure A.1, landmark methods could be elements #1 and #2, while barrier methods – #5 and #7. The system computes a hammock graph which contains all of the direct paths between the landmarks. That is, a method call belongs to the hammock graphs only if it is on a direct path between a pair of landmark methods. Additional potentially relevant methods are added to the graph using intra-procedural backward slicing [165] (with the call sites that spawn calls in the hammock graph as slicing criteria). Since slicing tends to produce graphs that are too large for practical purposes, barrier methods are used to eliminate irrelevant sections of the graph: all incoming and outgoing call graph edges of barrier methods are removed, and thus these are not traversed during the slice computation. The approach outputs all elements of the resulting graph as relevant to the feature.

In our example in Figure A.1, no direct call paths exist between elements #1 and #2; thus, the approach is unable to find additional relevant elements under the given input. The technique is largely automated and does not require extensive user interaction (denoted by ‘+’ in Figure A.8) other than providing and possibly refining the input sets of methods.

Shepherd et al. [158] attempt to locate action-oriented concepts in object-oriented programs using domain knowledge embedded in the source code through identifier names (methods and local variables) and comments. It relies on the assumption that verbs in object-oriented programs correspond to methods, whereas nouns correspond to objects.

The analyzed program is represented as an action-oriented identifier graph model (AOIG) [159] where the actions (i.e., verbs) are supplemented with direct objects of each action (i.e., objects on which the verb acts). For example, the verb save in Figure A.1 can act on different objects in a single program, such as MindMapMapModel and MindMapNodeModel; these are the direct objects of save. An AOIG representation of a program contains four kinds of nodes: verb nodes, one for each distinct verb in the program; direct object (DO) nodes, one for each unique direct object in the program; verb-DO nodes, one for each verb-DO pair identified in the program (a verb or a direct object can be part of several verb-DO pairs); and use nodes, one for each occurrence of a verb-DO pair in
comments or source code of the program. An AOIG has two kinds of edges: *pairing edges* connecting each verb or DO node to verb-DO pairs that use them, and *use edges* connecting each verb-DO pair to all of its use nodes.

As an input, the user formulates a query describing the feature of interest and *decomposes* it into a set of pairs (verb, direct object). The technique helps the user to refine the input query by collecting verbs and direct objects that are similar (i.e., different forms of words, synonyms, etc.) to the input verbs and direct objects, respectively, as well as words collocated with those in the query, based on the verb-DO pairs of the program AOIG. For example, `MindMapMapModel` is collocated with `MindMapNodeModel` in verb-DO pairs for the verb `save`. The collected terms are ranked by their “closeness” to the words in the query based on the frequency of collocation with the words in the query and on configurable weight given to synonyms. Ten best-ranked terms are presented to the user. The system then recommends that the user augment the query with these terms as well as with program methods that match the current query.

Once the user is satisfied with the query, the system searches the AOIG for all verb-DO pairs that contain the words of the query. It extracts all methods where the found pairs are used and applies PDA to detect call relationships between the extracted methods. The system then generates the *result graph* in which nodes represent detected methods and edges represent identified structural relationships between them. The graph is returned to the user.

For our example in Figure A.1, the input query (`doAutomaticSave, MindMapMapModel`) might get expanded by the user with the terms `save` and `saveInternal`, because they are collocated with `MindMapMapModel`. Then, the system outputs elements #1 through #4 and #7, together with the corresponding call graph fragment. The technique requires a fair amount of user interaction to construct and refine the input query, and thus is marked with ‘++’ in Figure A.8.

Zhao et al. [183] accept a set of feature descriptions as input and focus on locating the *specific* and the *relevant* functions of each feature using PDA and IR technologies. The specific functions of a feature are those definitely used to implement it but are not used by other features. The relevant functions of a feature are those involved in the implementation of the feature. Obviously, the specific function set is a subset of the relevant function set for every feature.

The analyzed program is represented as a *Branch-Reserving Call Graph (BRCG)* [122] – an expansion of the call graph with branching and sequential information, which is used to construct the pseudo execution traces for each feature. Each node in the BRCG is a function, a branch, or a return statement. Loops are regarded as two branch statements: one going through the loop body and the other one exiting immediately. The nodes are related either sequentially, for statements executed one after another, or conditionally, for alternative outcomes of a branch.

The system receives a paragraph of text as a description of each feature. The text can be obtained from the requirements documentation or be provided by a domain expert. It transforms each feature description into a set of index terms (considering only nouns and verbs and using their normalized form). These will be used as documents. The system then extracts the names of each method and its parameters, separating identifiers using known coding styles (e.g., using the underline ‘_’ to separate words) and transforms them into index terms. These will be used as queries.

To reveal the connections between features and functions, documents (feature descriptions) are ranked for each query (function) using the vector space model [10, pp. 27-30] – a technique which, similarly to LSI, treats queries and documents as vectors constructed by the index terms. Unlike LSI, the weights of index term in documents and queries are calculated using the *tf-idf* metric (see Sec. A.1.3) between the term and the document or query, respectively. For the example in Figure A.1, `automatic save file` could be a document while “`mind map model do automatic save`” could be a query corresponding to the element #2. For the vector space model, the weight of the
term `save` in the query is 0.24, as calculated in Sec. A.1.3. Note that LSI calculates this weight as being 2 (see the value of the term `save` in the column that corresponds to $d_2$ in Figure A.5).

Similarity between a document and a query is computed as a cosine of the angle between their corresponding vectors, as for LSI. For each document (feature), the system creates a sorted list of queries (functions), ranked by their similarity degrees and identifies a pair of functions with the largest difference between scores. All functions before this pair, called a division point, are considered initial specific functions to the feature. In our example, these are elements #1 and #2.

Next, the system analyzes the program’s BRCG and filters out all branches that do not contain any of the initial specific functions of the feature, because those are likely not relevant; all remaining functions are marked as relevant. Functions relevant to exactly one feature are marked as specific to that feature.

The system also builds pseudo-execution traces for each feature by traversing the pruned BRCG and returns those to the user. For our example in Figure A.1, BRCG is rooted in element #8. Since there is no direct call to element #1 (the call is performed via an event queue – see the last statement in Figure A.2), the technique returns only those branches that contain element #2, that is, elements #8, #7 and #4. The technique requires no user interaction besides the definition and the refinement of the input feature descriptions, as reflected by ‘+’ in Figure A.8.

Robillard et al. [133] propose searching the change history (change transactions) of a software system to identify clusters of program elements related to a task. The analyzed program is represented as a set of program elements such as fields and methods, as well as change history transactions that capture modifications of these elements. The system considers all available transactions and filters out those with more than twenty or fewer than four elements. The thresholds are set empirically: experiments revealed that large transactions generate overly large clusters that would require developers to spend an unreasonable amount of effort to study, while small transactions cannot be clustered efficiently. The system then clusters the remaining transactions based on the number of overlapping elements using a standard clustering algorithm.

Next, given a small set of elements related to a feature of interest (usually two or three), the system extracts clusters containing all input elements and removes those satisfying the following conditions:

1. An input element appears in at least 3% of the transactions of the cluster. The rationale is that querying the change history for elements that are being continuously modified (and thus are central or critical elements to the entire system) returns too many recommendations to be useful.

2. The degree of overlap between elements that correspond to the transactions in a cluster is lower then 0.6. The rationale is that these clusters do not represent changes that are associated with a high-level concept.

3. The number of transactions in a cluster is less than 3. The rationale is to avoid returning results that are single transactions or very small groups of transactions which may have been spontaneously clustered. However, using a value higher than 3 as a threshold produces too few recommendations to be useful.

All elements of the resulting clusters are returned to the user. The technique requires no user interaction besides the definition and the refinement of the input elements, as reflected by ‘+’ in Figure A.8.

Unfortunately, the evaluation of the proposed technique which is included in the paper shows that the benefits of using change clusters are relatively small: the analysis of almost 12 years of software change data for a total of seven different open-source systems showed that fewer than 12% of the studied changes could have benefited from finding elements relevant to the change using change clusters.

Trifu [168] proposes an approach that uses static dataflow information to determine the concern skeleton – a data-oriented abstraction of a feature. The analyzed program is represented as a concern graph whose nodes are variables found in the source code and whose edges are either dataflow relationships that capture value transfer
between variables or *inheritance relationships* that insure consistent handling of variables defined in polymorphic methods. A path between two variables indicates that the start variable is used to derive the value of the end variable.

The approach treats a feature as an implementation of functionality needed to produce a given set of related values. It receives as input a set of variables that store key results produced by the feature of interest – *information sinks* – and computes a *concern skeleton* which contains all variables in the concern graph that have a path to one of the information sinks. The approach can be optionally provided with an additional set of input variables – *information sources* – that act as cutting points for the incoming paths leading to an information sink. That is, the computed concern skeleton includes only portions of the paths from the given information sources to the given information sinks. The computed concern skeleton is returned to the user.

The approach provides some help in identifying the input set of information sinks by computing a *reduced concern graph* in which certain variables are filtered out (e.g., those that have no incident edges in the concern graph). Still, identifying information sinks is not a trivial task which involves semantic knowledge about what the system does. Also, the user has to do the mapping from variables of the resulting concern skeleton to program statements that use them. Thus, the technique relies on extensive user interaction, as indicated by ‘+++’ in Figure A.8.

**Guided Output**

*Robillard [132]* leverages static program dependencies analysis to find elements that are related to an initial *set of interest* provided by the user. The analyzed program is represented as a PDG whose nodes are functions or data fields and edges are function calls or data access links. Given an input *set of interest* – a set of functions and data fields that the user considers relevant to the feature of interest, the system explores their neighbors in the dependency graph and scores them based on their *specificity* – an element is specific if it relates to few other elements, and *reinforcement* – an element is reinforced if it is related to other elements of interest. For the example in Figure A.1, if the initial set of interest contains elements #3 and #4, reinforcement of element number #7 is high as two of its three connections are to elements of interest. Reinforcement of element #1 is even higher, as its sole connection leads to an element of interest. Yet, specificity of element #7 is lower than that of element #1 since the former is connected to three elements whereas the latter – just to one.

The set of all elements related to those in the initial set of interest is scored and returned to the user as a sorted *suggestion set*. The user browses the result, adds additional elements to the set of interest and reiterates. The amount of the required user interaction in this approach is moderate, as indicated by ‘++’ in Figure A.8: the technique itself only browses the direct neighbors of the elements in the input *set of interest* while the user is expected to extend this set interactively, using the results generated by the previous step.

*Saul et al. [152]* build on Robillard’s technique [132]) and introduce additional heuristics for scoring program methods. The proposed approach consists of two phases: in the first, a set of potentially relevant methods is calculated for an input method of interest. These are the union of caller and callee methods (“parents” and “children”), methods called by the caller functions (“siblings”) and methods that call the callee methods (“spouses”). For example, for the element #4 in Figure A.1, elements #2, #3, #7 and #8 are potentially relevant.

The calculated set of potentially relevant methods is then scored using the HITS web mining algorithm (see Sec. A.1.4) based on their “strength” as *hubs* (methods that aggregate functionality, i.e., call many other methods) or *authorities* (methods that largely implement functionality without aggregating). The calculated authority score is used to rank the results returned by the algorithm. That is, a method gets a high score if it is called by many high-scored hub methods. In our example, element #7 has a lower score than #4, because the former is called only
by method #8 which is a low-scored hub method as it calls only one method. Element #4 has a higher score because
(1) it is called by both elements #7 and #8, and (2) element #7 has a higher hub score as it calls two methods rather
than one.

Similar to [132], the technique requires a moderate amount of user interaction, as indicated by ‘++’ in Figure A.8.

**Marcus et al. [105, 106]** introduce one of the first approaches for using IR techniques for feature location. The
approach is based on using domain knowledge embedded in the source code through identifier names and internal
comments.

The analyzed program is represented as a set of text documents describing software elements such as methods
or data type declarations. To create this set of documents (corpus), the system extracts identifiers from the source
code and comments, and separates the identifiers using known code styles (e.g., the use of underline ‘_’ to separate
words). Each software element is described by a separate document containing the extracted identifiers and
translated to LSI space vectors (see Sec. A.1.2) using identifiers as terms.

Given a natural language query containing one or more words, identifiers from the source code, a phrase or even
short paragraphs formulated by the user to identify a feature of interest \(^5\), the system converts it into a document in
LSI space, and uses the similarity measure between the query and documents of the corpus in order to identify the
documents most relevant to the query.

In order to determine how many documents the user should inspect, the approach partitions the search space
based on the similarity measure: each partition at step \(i + 1\) is made up of documents that are closer than a given
threshold \(\alpha\) to the most relevant document found by the user in the previous step \(i\). The user inspects the suggested
partition and decides which documents are part of the concept. The algorithm terminates once the user finds no
additional relevant documents in the currently inspected partition and outputs a set of documents that were found
relevant by the user, ranked by the similarity measure to the input query.

For the example in Figure A.1, assume that similarities between documents and a query are calculated as
specified in Sec. A.1.2 and summarized in Table A.1. That is, only terms from method names (and not from method
bodies) are used. Under this setting, if \(\alpha\) is set to 0.3, the first partition will contain only document \(d_2\) and the
second – only \(d_1\). No other document is within 0.3 of \(d_1\) and thus the algorithm will terminate and output \(d_1\) and
\(d_2\).

The technique requires no user interaction besides the definition and the refinement of the input query, and thus
is marked with ‘+’ in Figure A.8.

**Poshyvanyk et al. [121]** extend the work of Markus et al. [105, 106] with Formal Concept Analysis (see Sec. A.1.1)
to select most relevant, descriptive terms from the ranked list of documents describing source code elements. That
is, after the documents are ranked based on their similarity to the input query using LSI, as in [105, 106], the system
selects the first \(n\) documents and ranks all terms that appear uniquely in these documents. The ranking is based on
the similarity between each term and the document of the corpus, such that the terms that are similar to those in the
selected \(n\) documents but not to the rest are ranked higher. Terms that are similar to documents not in the selected
\(n\) results are penalized because they might be identifiers for data structures or utility classes which would pollute
the top ranked list of terms. For the example in Figure A.1, given the LSI ranking with respect to the automatic
save file query shown in Table A.1, if \(n\) is set to 2, documents \(d_1\) and \(d_2\) are selected. The unique terms in these are
“automatic”, “do” and “run”, all ranked high as they are not similar to any of the terms in the rest of the documents.

After the unique terms are ranked, the system selects the top \(k\) terms (attributes) from the first \(n\) documents

---

\(^5\)Several approaches, e.g., [11, 29], address the problem of input query definition. They consider not only the query but also related terms
when evaluating the document models. As discussed earlier, these approaches are out of the scope of this chapter.
(objects) and applies FCA (see Sec. A.1.1) to build the set of concepts. For the three terms in our example, the
concepts are $\{d_1, d_2\}, \{\text{automatic, do}\}$ and $\{d_1\}, \{\text{automatic, do, run}\}$. The terms describe the resulting
documents. The user can inspect the generated concepts – the description and links to actual documents in the
source code – and select those that are relevant. Similar to [105, 106], the technique requires a low amount of user
interaction, as indicated by ‘+’ in Figure A.8.

Shao et al. [155] introduce another approach that extends the work of Marcus et al. [105, 106] by completing the
LSI ranking with static call graph analysis. Each method of the analyzed program is represented by a document
containing its identifiers. After the LSI rank for each document with respect to the input query is calculated, the
system builds a set of methods corresponding to documents ranked above a certain threshold and computes a set
of all callers and callees of these methods. The LSI score of the elements in the computed set is augmented to
represent their call graph proximity to one of the methods ranked high by LSI. The algorithm outputs a list of all
methods organized in a descending order by their combined ranking. For the example in Figure A.1, element #3 is
ranked low by LSI with respect to the query “automatic save file” (-0.2034 in Table A.1). However, it is called by
element #1 which has a high LSI rank (0.6319 in Table A.1). Thus, the score of element #3 will be augmented and
it will be ranked higher.

The technique requires no user interaction except defining and refining the input query describing the feature of
interest, as indicated by ‘+’ in Figure A.8.

Hill et al. [69] combine call graph traversal with the \textit{tf-idf}-based ranking (see Sec. A.1.3). The analyzed program
is represented as a call graph and a set of text documents. Each document corresponds to a method of the program
and includes all identifiers used in the method. The user provides an initial query that describes the feature, a seed
method from which the exploration starts, and the exploration depth which determines the neighborhood to be
explored (i.e., a maximal distance of explored methods from the seed).

Starting from the input seed method, the system traverses the program call graph and calculates the relevance
score of each explored method by combining the following three parameters: (1) the \textit{tf-idf} score of the identifiers
in the method name; (2) the \textit{tf-idf} score of the identifiers in the method body; and (3) a binary parameter specifying
whether the method is from a library or part of the user code. If the score of a method is higher than a preset
relevance threshold, the method is marked as relevant. If the score is higher than a preset exploration threshold
(which is usually lower than the relevance threshold) and the distance of the element from the seed is lower than
the exploration depth, the system continues exploring the neighborhood of this element. Otherwise, the element
becomes a “dead-end”, and its neighborhood is not explored. When there are no additional elements to explore
for the given exploration depth, the system outputs the call-graph neighborhood of the seed method in which all
elements are scored and relevant elements are marked.

For the example in Figure A.1, if the element #1 is used as a seed and the exploration depth is set to 3, all
elements other than #2 can be potentially explored. For the sake of the example, we disregard the terms that appear
in method bodies and assume that the program does not use any binary methods. In such a case, the calculated
score of element #3 is based on the \textit{tf-idf} similarity of the method name to the input query – 0.12 for the input
query “automatic save file”, as shown in Sec. A.1.3. Thus, setting the exploration threshold above this value results
in not exploring the part of the graph starting with element #1, and thus no elements are returned to the user. The
exploration threshold of up to 0.12 results in further exploration of the call graph.

The relevance threshold specifies which of the explored elements are considered relevant. Both relevance and
exploration thresholds are set empirically, based on the experience with programs under analysis. The technique
requires no user interaction besides the definition and the refinement of the input feature description and seed
method, and thus is marked with ‘+’ in Figure A.8.
Chen et al. [25] present a technique for retrieving lines of code that are relevant to an input query by performing textual search on the cvs comments associated with these lines of code. The analyzed program is represented as a set of lines for a newest revision of each file. The system examines changes between subsequent versions of each file using the cvs diff command, and associates the corresponding comment with each changed line. It stores all associated cvs comments for each line of a file in a database and retrieves all lines whose cvs comments contain at least one of the input query’s words. The results are scored to indicate the quality of the match: the more query words appear in the comment, the higher is the score. In addition, the system searches the source code to find lines containing at least one of the query’s words. It outputs a sorted list of files so that those with the highest number of matches appear first. Within each file, a sorted list of all lines that either match the query or are associated with a cvs comment that matches it is presented.

The technique is largely automated and requires no user interaction other than providing and possibly refining the input query, as indicated by ‘+’ in Figure A.8.

A.3.2 Dynamic Feature Location Techniques

In this section, we describe techniques that rely on program execution for locating features in source code. The majority of such techniques address the feature location task for sequentially executed programs, thus we focus the section on those techniques. We note that some of the described approaches have been extended, e.g., [46, 4], to handling distributed and multi-threaded systems as well.

Plain Output

Widle et al. [176] introduced one of the earliest feature location techniques taking a fully dynamic approach. The main idea is to compare execution traces obtained by exercising the feature of interest to those obtained when the feature of interest is inactive. Towards this end, the program is instrumented so that the components executed on a scenario / test case can be identified. The granularity of components, e.g., methods or lines of code, is defined by the user. The user specifies a set of test cases that invoke each feature. The system runs all input test cases and analyzes their execution traces, identifying common components – executed by all test cases. In addition, for each feature, it identifies (1) potentially involved components – executed by at least one test case of the feature; (2) indispensably involved components – executed by all test cases of the feature; and (3) uniquely involved components – executed by at least one test case of the feature and not executed by any test case of the other features. The system outputs sets of potentially involved, indispensably involved and uniquely involved components for each feature, as well as the set of all common components.

For the example in Figure A.1, the execution trace of the automatic save file feature can be compared to the that of the manual save file feature. In this case, elements #3, #5 and #6 are considered common, since the automatic save file feature relies on the execution of manual save file and, thus, these methods are executed in both scenarios. Element #1 is considered uniquely involved as it is executed by the automatic save file feature only.

Since the user is required to define two sets of scenarios for each feature – those that exercise it and those that do not, the technique requires heavy user involvement and we assess it as ‘+++’ in Figure A.8.

Wong et al. [177] present ideas similar to [176]. Its main contribution is in analyzing data flow dependencies in addition to the control flow (method calls) and in presenting a user-friendly graphical interface for visualizing features.

Eisenbarth et al. [48] attempts to address one of the most significant problems of dynamic approaches discussed above – the difficulty of defining execution scenarios that exercise exactly one feature. Their work relies on the
assumption that execution scenarios can implement more than one feature and a feature can be implemented by more than one scenario. The work extends [176] with FCA (see Section A.1.1) to obtain both computation units for a feature as well as the jointly and distinctly required computation units for a set of features.

The analyzed program is represented by an instrumented executable and a static program dependence graph whose nodes are methods, data fields, classes, etc. and whose edges are function calls, data access links and other types of relationships obtained by static analysis. While in general the technique is applicable to computation units on any level of granularity, the approach is implemented and evaluated for method-level components. The system first executes all given input scenarios, each of which can invoke multiple features. Optionally, users can identify special start and end scenarios whose components correspond to startup and shutdown operations and are excluded from all executions.

Users select a subset of execution scenarios they wish to investigate. Then, the approach uses FCA (see Section A.1.1), where computation units are objects, scenarios are attributes and relationships specify whether a unit is executed when a particular scenario is performed, to create a concept lattice. Based on the lattice, the following information is derived: (1) a set of computation units specific to a feature – those used in all scenarios invoking the feature, but not in other scenarios; (2) a set of computation units relevant to a feature – used in all scenarios invoking the feature, and possibly in other scenarios; (3) a set of computation units conditionally specific to a feature – those used in some scenarios invoking the feature, but not in scenarios that do not invoke the feature; (4) a set of computation units conditionally relevant to a feature – those used in some scenarios invoking the feature, and possibly in other scenarios that do not invoke the feature; and (5) a set of computation units irrelevant to a feature – those used only in scenarios that do not invoke the feature. In addition, for each feature, the system builds a starting set in which the collected computation units are organized from more specific to less. It also builds a subset of the program dependency graph containing all transitive control flow successor and predecessors of computation units in the starting set (i.e., method callers and callees). The graph is annotated with features and scenarios for which the computation units were executed.

The user inspects the created program dependency graph and source code in the order suggested by the starting set, in order to refine the set of identified computation units for a feature by adding and removing computational units. During the inspection, the system also performs two further analyses to assist with the call graph inspection: strongly connected component analysis and dominance analysis. The former is used for identifying cycles in the dependency graph. If there is one computation unit in the cycle that contains feature-specific code, all computation units of the cycle are related to the feature because of the cycle dependency. The purpose of the latter is to identify computation units that must be executed in order to reach one of the computation units containing feature-specific code. All such computation units are related to the feature as well.

At the end of the process, a set of components deemed relevant for each feature is generated. Even though the technique attempts to assist the user in defining input scenarios, the required level of user interaction in defining the scenarios, selecting the order in which the scenarios are processed, as well as interactively inspecting and refining the produced result is still high, as indicated by ‘+++’ in Figure A.8.

Koschke et al. [92] extend the work of Eisenbarth et al. [48] by considering statement-level rather than method-level computation units.

Asadi et al. [8] propose an approach which combines IR, dynamic-analysis and search-based optimization techniques to locate cohesive and decoupled fragments of traces that correspond to features. The approach is based on the assumptions that methods responsible for implementing a feature are likely to share some linguistic information and be called close to each other in an execution trace.

For an input set of scenarios that exercise the features of interest, the system collects execution traces and
prunes methods invoked in most scenarios (e.g., those related to logging). In addition, it compresses traces to remove repetition of one or more method invocations and keeps one occurrence of each method. Next, it tokenizes each method’s source code and comments, removing special characters, programming language keywords and terms belonging to a stop-word list for the English language (e.g., ‘the’, ‘is’, ‘at’). The remaining terms are tokenized separating identifiers using known coding styles. The terms belonging to each method are then ranked using the $tf-idf$ metric (see Section A.1.3) with respect to the rest of the corpus. For the example in Figure A.1, when considering only terms of the method names, the term `mind` appears in all documents and thus is ranked 0, while the term `controller` appears only in one document (that corresponds to element #8) and thus gets a higher rank – 0.9. The obtained term-by-document co-occurrence matrix is transformed to vectors in the LSI space (see Section A.1.2). A cosine similarity between two vectors in LSI space is used as a similarity measure between the corresponding documents (methods).

Next, the system uses genetic optimization algorithm [61] – an iterative procedure that searches for the best solution to a given problem by evaluating various possible alternatives using an objective function, in order to separate each execution trace into conceptually-cohesive segments that correspond to the features being exercised in a trace. In this case, an optimal solution is defined by two objectives: maximizing segment cohesion – the average similarity between any pair of methods in a segment, and minimizing segment coupling – the average similarity between a segment and all other segments in a trace, calculated as average similarity between methods in the segment and those in different ones. That is, the algorithm favors merging of consecutive segments containing methods with high average similarity.

The approach does not rely on comparing traces that exercise the feature of interest to those that do not and does not assume that each trace corresponds to one feature. Thus, the task of defining the execution scenarios is relatively simple. However, the approach does not provide any assistance in helping the users to understand the meaning of the produced segments and tracing those to the features being exercised in the corresponding scenario; thus, this step requires a fair amount of user interaction. In Figure A.8, we rate this approach as ‘++

Guided Output

Eisenberg et al. [49], similar to Eisenbarth et al. [48], present an attempt to deal with the complexity of scenario definition. The approach assumes that the user is unfamiliar with the system and thus should use pre-existing test suites, such as those typically available for systems developed with a Test-Driven Development (TDD) strategy. It accepts as input a test suite that has some correlation between features and test cases (i.e., all features are exercised by at least one test case). Tests that exhibit some part of a feature functionality are mapped to that feature and referred to as its exhibiting test set. Tests which are not part of any exhibiting test set are grouped into sets based on similarity between them and are referred to as the non-exhibiting test set.

For each feature, the system collects execution traces obtained by running all tests of the feature’s exhibiting test set and generates a calls set which lists $<$caller, callee$>$ pairs for each method call specified in the collected traces. It then ranks each method heuristically based on the following parameters: (1) multiplicity – a relationship between the percentage of tests in the exhibiting test set of the feature that execute the method and the percentage of tests in non-exhibiting test sets that execute that method; (2) specialization – the percentage of test sets that exercise the method. (If a method is exercised by many test sets, it is more likely to be a utility method); and (3) depth – the call depth (the number of stack frames from the top) of the method in the exhibiting test set compared to that in non-exhibiting test sets. The rationale behind these heuristics is that the exhibiting test set focuses on the feature in the most direct way. This is correlated with the call depth of the methods that implement this feature – the more “directly” a method is exercised, the lower its call depth.
For each feature, both the ranked list of methods and the generated call set are returned to the user. The goal of the former is to rank methods by their relevance to a feature, whereas the goal of the latter is to assist the user in understanding why a method is relevant to a feature. With respect to the required level of user interaction, we assess the technique as ‘++’ in Figure A.8 because of the effort involved in creating test scenarios, if they are not available.

Poshyvanyk et al. [120] combine the techniques proposed in Marcus et al. [106] and Antoniol et al. [4] to use LSI (see Section A.1.2) and execution-trace analysis to assist in feature location. The analyzed program is represented by a set of text documents describing software methods and a runnable program instrumented so that methods executed on any scenario can be identified.

Given a query that is formulated by the user to identify a given feature and two sets of scenarios – those that exercise the feature of interest and those that do not, the system first ranks input program methods using LSI. Then, it executes input scenarios, collects execution profiles and ranks each executed method based on the frequency of its appearance in the traces that exercise the feature of interest versus traces that do not. The final rank of each method is calculated as a weighted sum of the above two ranks. The system outputs a ranked list of methods for the input feature.

For the example in Figure A.1, element #1 is executed only in scenarios that exercise automatic save file. Thus, its LSI score (0.6319, as calculated in Table A.1) will be increased, while the score of element #5 (0.2099, as calculated in Table A.1) will be decreased to reflect the fact that it is executed in both scenarios that exercise the automatic save file feature and those that do not.

Similar to other dynamic approaches, this approach requires an extensive user involvement for defining scenarios that exercise the feature of interest and those that do not and, therefore, we assess the level of the necessary user interaction for this technique as ‘++’ in Figure A.8.

Liu et al. [102], similar to Poshyvanyk et al. [120], combine the use of LSI and execution-trace analysis. However, this work proposes operating on a single trace rather than on multiple traces that exercise / do not exercise the feature of interest.

Given a query that is formulated by the user to identify a feature of interest and a single scenario capturing that feature, the system executes the input scenario and ranks methods executed in the scenario using LSI with respect to the input query as in [106]. A ranked list of executed methods is returned to the user. For our example in Figure A.1, a scenario that executes the automatic save file feature invokes elements #1, #3, #6 and #7. These elements are returned to the user together with their LSI ranking, shown in Table A.1.

Since the user is only required to provide a single scenario that exercises each feature of interest and a natural language description of that feature, we assess the level of the necessary user interaction for this technique as ‘+’ in Figure A.8.

Rohatgi et al. [135] present a technique that is based on dynamic program analysis and static program dependence graph analysis. The technique operates on a class level, where the analyzed program is represented by an instrumented executable and a static program class dependency graph whose nodes are classes and whose edges are dependency relationships among these classes such as method calls, generalization and realization.

As input, the system obtains a set of scenarios that invoke the features of interest. It executes all input scenarios, collects execution profiles on a class level and uses impact analysis to score the relevance of the classes to the feature of interest: classes that impact many others in the system are ranked low as these classes are likely not feature-specific but rather “utility” classes implementing some core system functionality. The technique outputs a set of classes produced by the dynamic trace analysis, ranked by their relevance as calculated using impact analysis.
We assess the level of the necessary user interaction for this technique as ‘++’ in Figure A.8 because it requires only a set of scenarios that invoke the features of interest and not those that don’t.

Eaddy et al. [43] present the PDA technique called prune dependency analysis which is based on the assumption that an element is relevant to a feature if it should be removed or otherwise altered if the feature is removed from the program. The program is represented as a program dependence graph whose nodes are classes and methods, and whose edges are method invocations, containment relationships between a class and its methods, or inheritance relationships between classes. The system calculates the set of all elements affected by removing at least one element from the seed input set. For the example in Figure A.1, removing element #2 requires removing or altering element #4 that initiates a call to it in order to avoid compilation errors. Thus, element #4 is related to the feature that involves execution of element #2. Removing element #4 requires removing elements #7 and #8. The latter does not trigger any additional removals.

Furthermore, the work suggests combining the proposed technique with existing dynamic- and IR-based feature location approaches to achieve better accuracy. The dynamic feature location can use the approaches proposed in [176, 49] or others. These either produce a ranked set of methods, as in Eisenberg et al. [49] or a unsorted list of relevant elements, as in Wilde et al. [176]. In the latter case, an elements is assigned the score 1 if it is executed only by scenarios exercising the feature of interest, or 0 otherwise. The IR-based feature location uses the approach of Zhao et al. [183]: program elements are ranked with respect to feature descriptions (extracted from requirements) using the vector space model. It calculates the cosine of the distance between the corresponding vectors of terms, each of which first weighted using the tf-idf metric.

For each software element, the resulting score is calculated by normalizing, weighing and adding the similarity scores produced by the IR and the dynamic techniques, as in Poshyvanyk et al. [120]. Then, similarly to Zhao et al. [183], the system applies a threshold to identify highly relevant elements. These are used as input to the prune dependency analysis which produces the set of additional relevant elements. The resulting set, ranked by the combination of scores produced by IR and dynamic techniques, is returned to the user.

For our example in Figure A.1, elements #1 and #2 are ranked high by the vector space model for the query “automatic save file”. Since element #1 is executed only by scenarios that exercise the automatic save file feature, it is also ranked high by a dynamic analysis-based technique. Prune dependency analysis uses these two as the input seed set and adds elements #4, #7 and #8, so the result becomes {#1, #2, #4, #7, #8}. Since the technique requires two sets of scenarios for each feature – those that exercise it and those that do not, we assess the level of the necessary user interaction for this technique as ‘+++’ (see Figure A.8).

Revelle et al. [128] propose improving the feature location accuracy by combining IR, dynamic and web-mining analysis. Similarly to Liu et al. [102], the proposed system obtains as input a single scenario that exercises the feature of interest and a query that describes that feature. It runs the scenario and constructs a call graph from the execution trace, which is a subgraph of the static call graph and contains only the methods that were executed. Next, the system assigns each method of the graph a score using one of the existing web-mining algorithms – either HITS (see Section A.1.4) or the PageRanked algorithm developed by Brin and Page [21], which is also based on similar ideas of citation analysis. The system then either filters out low-ranked methods (e.g., if the HITS authority score was used, as in Saul et al. [152]) or high-ranked methods (e.g., if the HITS hub score was used, as high-ranked methods represent common functions). The remaining set of elements is scored using LSI (see Section A.1.2) based on their relevance to the input query describing the feature. The ranked list of these elements is returned to the user.

For the example in Figure A.1, elements #1, #3, #5 and #6 are invoked when the scenario exercising the automatic save file feature is executed. Assuming these elements are scored using HITS authority values, filtering
out low-scored methods removes element #1 from the list of potentially relevant elements as its authority score is 0, as shown in Section A.1.4. The remaining elements, #3, #5 and #6, are scored using LSI with respect to the query “automatic save file” (these scores are given in Table A.1) and are returned to the user.

Similar to [102], since the user is only required to provide a single scenario for each feature of interest and a natural language description of that feature, we assess the level of the necessary user interaction for this technique as ‘+’ in Figure A.8.

A.4 Which Technique to Prefer?

As the survey shows, there is large variety in existing approaches and implementation strategies for feature location. We believe that trying to identify a single technique that is superior to the rest would be impractical. Clearly, there is no “silver bullet”, and the performance of each technique largely depends on its applicability to the analyzed input programs and the quality of the feature description (feature intension) provided by the user. In this section, we discuss considerations and provide explicit guidelines for practitioners who need to choose a particular feature location technique to apply.

The chosen technique should first and foremost be suitable to the program being analyzed: specifically, if the studied program contains no documentation and no meaningful identifier names, IR-based feature location techniques will be unable to achieve high-quality results. Similarly, if the implementation of a feature is spread across several program modules or is hooked into numerous extension points provided by the platform on which the program is built (e.g., invoking methods via event queues), techniques based on program dependency analysis will either be unable to find all elements that relate to the implementation of the feature or will find too many unrelated elements. When program execution scenarios are unavailable or it is cumbersome to produce scenarios that execute a specific set of features (e.g., because the feature of interest is not a functional feature that is “visible” at the user level), dynamic feature location techniques will not be applicable. Figure A.9 assesses the surveyed feature location techniques based on the above selection criteria.

For our example in Figure A.1 and A.2, program elements have meaningful names (“file” vs. “f” or “property” vs. “prp”). Thus, it is reasonable to choose one of the techniques that rely on that quality, as marked in the corresponding column of Figure A.9. Since the implementation of the Freemind software is asynchronous and relies on event queues to perform method invocation, techniques that analyze call graph dependency might be less efficient. In addition, defining a scenario that triggers the automatic save file feature might not be trivial – there is no user operation that directly invokes the automatic save (as opposed to the manual save) functionality. Therefore, techniques that do not require program execution are a better choice which leads us to the approaches in Shepherd et al. [158], Marcus et al. [105, 106] or Poshyvanyk et al. [121].

With respect to the quality of a feature intent provided by the user, IR-based techniques are usually most sensitive to the quality of their input – the query that describes the feature of interest. The results produced by these techniques are often as good as the query that they use. Input query definition and the user assistance during that process are further discussed by [11, 29] and others. Techniques based on comparing dynamic execution traces are also sensitive to the nature of their input – if execution scenarios do not cover all aspects of the located feature, the accuracy of the feature location will likely be low.

The approaches also differ in the required level of user interaction (see the last column of Figure A.8). We assess the level of user interaction based on the effort that the user has to invest in operating the technique. This includes the effort involved in defining the input feature intension (e.g., a set of scenarios exercising the features of interest), interactively following the location process (e.g., filtering intermediate results produced by the technique).
### Figure A.9: Criteria for selecting a feature location technique.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Strongly Coupled Implementation</th>
<th>Meaningful Names</th>
<th>Change Histories</th>
<th>Execution Scenarios</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly Coupled</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Walkinshaw et al. 2007 [172]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shepherd et al. 2007 [158]</td>
<td>(Find-Concept)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zhao et al. 2006 [163]</td>
<td>(SNIAFL)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Robillard et al. 2008 [133]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trifu 2009 [166]</td>
<td>(✓)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Guided</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Robillard et al. 2005 [132]</td>
<td>(Suade)</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Saul et al. 2007 [152]</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Marcus et al. 2004 [105, 106]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Poshyvanyk et al. 2007 [121]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shao et al. 2009 [155]</td>
<td>(✓)</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Hill et al. 2007 [69]</td>
<td>(Dora)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Chen et al. 2001 [25]</td>
<td>(CVSSearch)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Plain                      |                                  |                  |                  |                     |
| Wilde et al. 1995 [176]    | (Sw. Reconnaissance)             |                  |                  | ✓                   |
| Wong et al. 1999 [177]     |                                  |                  |                  | ✓                   |
| Eisenbarth et al. 2003 [48]| (✓)                             |                  |                  | ✓                   |
| Koschke et al. 2005 [92]   | (✓)                             |                  |                  | ✓                   |
| Asadi et al. 2010 [8]      |                                  |                  |                  | ✓                   |

| Dynamic Guided             |                                  |                  |                  |                     |
| Eisenberg et al. 2005 [49] |                                  |                  |                  | ✓                   |
| Poshyvanyk et al. 2007 [120]|                                  |                  |                  | ✓                   |
| Liu et al. 2007 [102]      | (SITIR)                          |                  |                  | ✓                   |
| Rohatgi et al. 2008 [135]  |                                  |                  |                  | ✓                   |
| Eaddy et al. 2008 [43]     | (Cerberus)                       |                  |                  | ✓                   |
| Reveille et al. 2010 [128] |                                  |                  |                  | ✓                   |

and interpreting the produced results (e.g., mapping retrieved variables to the code statements that use them).

Since more highly automated techniques are easier to execute, their “barrier to entry” – the effort required to produce the initial approximation of the result – is lower and thus their adoption is easier. On the other hand, the techniques that require more user interaction are usually able to produce better results because they harvest this “human intelligence” for the feature location process.

Furthermore, automated techniques could be a better choice for the users that seek an “initial approximation” of the results and are able to complete them manually since they are familiar with the analyzed code. On the other hand, users that cannot rely on their understanding of the analyzed code should probably choose a technique that is more effective at producing relevant results, even though operating such a technique requires a more intensive investment of time and effort.

### A.5 Summary and Conclusions

In this chapter, we provided a detailed description of twenty-four feature location techniques and discussed their properties. While all of the surveyed approaches share the same goal – establishing traceability between a specific feature of interest that is specified by the user and the artifacts that implement that feature, their underlying design principles, their input, and the quality of the results which they produce differ substantially. We discussed those in detail and identified criteria that can be used when choosing a particular feature location technique in a practical setting. We also illustrated the techniques on a common example in order to improve the understandability of their
underlying principles and implementation decisions.

Even though the area of feature location is mature, there is variety in existing techniques, which is caused by the common desire to achieve high accuracy: automatically find a high number of relevant elements (high recall) while maintaining a low number of false-positive results (high precision). As discussed in Section A.4, since there is no optimal technique, each of the approaches proposes heuristics that are applicable in a particular context, making the technique efficient in these settings.
Appendix B

Models Used for Evaluation in Chapter 7

This appendix provides the full set of models used as case studies in the merge-refactoring approach discussed in Chapter 7.
B.1 Washing Machine

Washing Machine Model A

Figure B.1 shows washing machine model A: a class diagram describing its structure and three state machines describing the behavior of its Controller, Washer and Dryer classes.

![Class Diagram](image)

(a) Class Diagram.

![Controller State Machine](image)

(b) Controller State Machine.

![Washer State Machine](image)

(c) Washer State Machine.

![Dryer State Machine](image)

(d) Dryer State Machine.

Figure B.1: Washing Machine Model A.
Washing Machine Model B

Figure B.2 shows washing machine model B: a class diagram describing its structure and three state machines describing the behavior of its Controller, HeatedWasher and Timer classes.

Figure B.2: Washing Machine Model B.
Washing Machine Model C

Figure B.3 shows washing machine model C: a class diagram describing its structure and three state machines describing the behavior of its Controller, DoubleRinseWasher and Beeper classes.

(a) Class Diagram.  
(b) Controller State Machine.

(c) DoubleRinseWasher State Machine.  
(d) Beeper State Machine.

Figure B.3: Washing Machine Model C.
Washing Machine Model A+B [threshold ≤ 0.70]

Figure B.4 shows the result of combining models A and B with the class similarity threshold of 0.70 or lower. The Controller, Motor, Faucet and DetergentSupplier classes of model A are merged with their corresponding versions in model B. The Washer and HeatedWasher classes, as well as the Dryer and Timer classes, are merged with each other.

![Class Diagram](image)

(a) Class Diagram.

![Controller State Machine](image)

(b) Controller State Machine.

![Washer/HeatedWasher State Machine](image)

(c) Washer/HeatedWasher State Machine.

![Dryer/Timer State Machine](image)

(d) Dryer/Timer State Machine.

Figure B.4: Washing Machine Model A+B [threshold ≤ 0.70]
Washing Machine Model (A+B)+C [threshold ≤ 0.70]

Figure B.5 shows the result of combining models (A+B) and C with the class similarity threshold of 0.70 or lower. The Controller, Motor, Faucet and DetergentSupplier classes of model A+B are merged with their corresponding versions in model C. The Washer/HeatedWasher and DoubleRinseWasher classes, as well as the Dryer/Timer and Beeper classes, are merged with each other.

Figure B.5: Washing Machine Model (A+B)+C [threshold ≤ 0.70]
Washing Machine Model \( A+B \) [threshold = 0.75]

Figure B.6 shows the result of combining models A and B with the class similarity threshold of 0.75. The Controller, Motor, Faucet and DetergentSupplier classes of model A are merged with their corresponding versions in model B. The Washer and HeatedWasher classes are merged with each other, while the Dryer and Timer classes are copied to the result as is.

Figure B.6: Washing Machine Model \( A+B \) [threshold = 0.75]
Washing Machine Model (A+B)+C [threshold = 0.75]

Figure B.7 shows the result of combining models (A+B) and C with the class similarity threshold of 0.75. The Controller, Motor, Faucet and DetergentSupplier classes of model A+B are merged with their corresponding versions in model C. The Washer/HeatedWasher and DoubleRinseWasher classes are merged with each other, while the Dryer/Beeper and Timer classes are copied to the result as is.

(a) Class Diagram.

(b) Controller State Machine.

(c) Washer/HeatedWasher/DoubleRinseWasher State Machine.

(d) Dryer/Beeper State Machine.

(e) Timer State Machine.

Figure B.7: Washing Machine Model (A+B)+C [threshold = 0.75]
Washing Machine Model A+B [threshold = 0.78]

Figure B.8 shows the result of combining models A and B with the class similarity threshold of 0.78. The Controller, Motor, Faucet and DetergentSupplier classes of model A are merged with their corresponding versions in model B. The Washer and HeatedWasher classes are merged with each other, while the Dryer and Timer classes are copied to the result as is.

Figure B.8: Washing Machine Model A+B [threshold = 0.78]
Washing Machine Model (A+B)+C [threshold = 0.78]

Figures B.9 and B.10 show the result of combining models (A+B) and C with the class similarity threshold of 0.78. The Controller, Motor, Faucet and DetergentSupplier classes of model A+B are merged with their corresponding versions in model C. The Washer/HeatedWasher and DoubleRinseWasher classes are merged with each other, while the Dryer, Beeper and Timer classes are copied to the result as is.

Figure B.9: Washing Machine Model (A+B)+C [threshold = 0.78] - Part 1
(a) Washer/HeatedWasher/DoubleRinseWasher State Machine.

(b) Dryer State Machine.

(c) Timer State Machine.

(d) Beeper State Machine.

Figure B.10: Washing Machine Model (A+B)+C [threshold = 0.78] - Part 2
Washing Machine Model A+B [threshold = 0.80]

Figure B.11 show the result of combining models A and B with the class similarity threshold of 0.80. The Controller, Motor, Faucet and DetergentSupplier classes of model A are merged with their corresponding versions in model B. The Washer and HeatedWasher classes are merged with each other, while the Dryer and Timer classes are copied to the result as is.

(a) Class Diagram.

(b) Controller State Machine.

(c) Washer/HeatedWasher State Machine.

(d) Dryer State Machine.

(e) Timer State Machine.

Figure B.11: Washing Machine Model A+B [threshold = 0.80]
Washing Machine Model (A+B)+C [threshold = 0.80]

Figures B.12 and B.13 show the result of combining models (A+B) and C with the class similarity threshold of 0.80. The Controller, Motor, Faucet and DetergentSupplier classes of model A+B are merged with their corresponding versions in model C. The remaining classes – Washer/HeatedWasher, DoubleRinseWasher, Dryer, Beeper and Timer are copied to the result as is.

Figure B.12: Washing Machine Model (A+B)+C [threshold = 0.80] - Part 1
Figure B.13: Washing Machine Model (A+B)+C [threshold = 0.80] - Part 2

(a) Washer/HeatedWasher State Machine.

(b) DoubleRinseWasher State Machine.

(c) Dryer State Machine.

(d) Timer State Machine.

(e) Beeper State Machine.
Washing Machine Model A+B [threshold = 0.85]

Figures B.14 and B.15 show the result of combining models A and B with the class similarity threshold of 0.85. The Controller, Motor, Faucet and DetergentSupplier classes of model A are merged with their corresponding versions in model B. The remaining classes – Washer, HeatedWasher, Dryer, Timer are copied to the result as is.

Figure B.14: Washing Machine Model A+B [threshold = 0.85] - Part 1
Figure B.15: Washing Machine Model A+B [threshold = 0.85] - Part 2
Washing Machine Model (A+B)+C [threshold = 0.85]

Figures B.16 and B.17 show the result of combining models (A+B) and C with the class similarity threshold of 0.85. The Motor, Faucet and DetergentSupplier classes of model A+B are merged with their corresponding versions in model C. The remaining classes – both Controller versions, Washer, HeatedWasher, DoubleRinseWasher, Dryer, Beeper and Timer are copied to the result as is.

(a) Class Diagram.

(b) Controller State Machine.

(c) Controller C State Machine.

Figure B.16: Washing Machine Model (A+B)+C [threshold = 0.85] - Part 1
Appendix B. Models Used for Evaluation in Chapter 7

Figure B.17: Washing Machine Model (A+B)+C [threshold = 0.85] - Part 2
Washing Machine Model A+B [threshold ≥ 0.90]

Figures B.18 and B.19 show the result of combining models A and B with the class similarity threshold of 0.90 or higher. The Motor, Faucet and DetergentSupplier classes of model A are merged with their corresponding versions in model B. The remaining classes – both Controller versions, Washer, HeatedWasher, Dryer, Timer are copied to the result as is.

![Diagram A](image1)

(a) Class Diagram.

![Diagram B](image2)

(b) Controller A State Machine.

![Diagram C](image3)

(c) Controller B State Machine.

Figure B.18: Washing Machine Model A+B [threshold ≥ 0.90] - Part 1
Figure B.19: Washing Machine Model A+B [threshold ≥ 0.90] - Part 2
Washing Machine Model \((A+B)+C\) [threshold \(\geq 0.90\)]

Figures B.20 and B.21 show the result of combining models \((A+B)\) and \(C\) with the class similarity threshold of 0.90 or higher. The \texttt{Motor}, \texttt{Faucet} and \texttt{DetergentSupplier} classes of model \((A+B)\) are merged with their corresponding versions in model \(C\). The remaining classes -- all \texttt{Controller} versions, \texttt{Washer}, \texttt{HeatedWasher}, \texttt{DoubleRinseWasher}, \texttt{Dryer}, \texttt{Beeper} and \texttt{Timer} are copied to the result as is.

![Class Diagram](image)

**Figure B.20**: Washing Machine Model \((A+B)+C\) [threshold \(\geq 0.90\)] - Part 1
Figure B.21: Washing Machine Model (A+B)+C [threshold $\geq 0.90$] - Part 2
B.2 Microwave Oven

The microwave oven example was taken from [63].

Microwave Oven Model A

Figure B.22 shows microwave oven model A: a class diagram and a state machine describing the behavior of class OvenTimer.

![Class Diagram](a) Microwave Oven A – Class Diagram.

![State Machine](b) Microwave Oven A – Oven Timer State Machine.

Figure B.22: Microwave Oven Model A
Microwave Oven Model B

Figure B.23 shows microwave oven model B: a class diagram and a state machine describing the behavior of class OvenTimer&MinutePlus.

(a) Microwave Oven B – Class Diagram.

(b) Microwave Oven B – Oven Timer & Minute Plus State Machine.

Figure B.23: Microwave Oven Model B
Microwave Oven Model C

Figure B.24 shows microwave oven model C: a class diagram and a state machine describing the behavior of class Oven&RecipeTimer.

![Class Diagram](image)

(a) Microwave Oven C – Class Diagram.

![State Machine](image)

(b) Microwave Oven C – Oven & Recipe Timer State Machine.

**Figure B.24: Microwave Oven Model C**
Microwave Oven Model D

Figure B.25 shows microwave oven model D: a class diagram and a state machine describing the behavior of class Oven&RecipeTimer&MinutePlus.

(a) Microwave Oven D – Class Diagram.

(b) Microwave Oven D – Oven & RecipeTimer & MinutePlus State Machine.

Figure B.25: Microwave Oven Model D
**Microwave Oven Model C+D [threshold \(\leq 0.70\)]**

Figure B.26 shows the result of combining models C and D with the class similarity threshold of 0.70 or lower. The MicrowaveOven class of model C is merged with its corresponding version in model D. The Oven&RecipeTimer and Oven&RecipeTimer&MinutePlus classes are merged with each other.

![Class Diagram](image1)

(a) Class Diagram.

![State Machine Diagram](image2)

(b) Oven & RecipeTimer/Oven & RecipeTimer & MinutePlus State Machine.

**Figure B.26: Microwave Oven Model C+D [threshold \(\leq 0.70\)]**
Microwave Oven Model A+B [threshold $\leq 0.70$]

Figure B.27 shows the result of combining models A and B with the class similarity threshold of 0.70 or lower. The MicrowaveOven class of model A is merged with its corresponding version in model B. The OvenTimer and OvenTimer&MinutePlus classes are merged with each other.

(a) Class Diagram.

(b) OvenTimer/OvenTimer & MinutePlus State Machine.

Figure B.27: Microwave Oven Model A+B [threshold $\leq 0.70$]
**Microwave Oven Model (C+D)+(A+B) [threshold \( \leq 0.70 \)]**

Figure B.28 shows the result of combining models C+D and A+B with the class similarity threshold of 0.70 or lower. The MicrowaveOven classes of all models are merged with each other. The Oven&RecipeTimer/Oven&RecipeTimer&MinutePlus class is merged with OvenTimer/OvenTimer&MinutePlus.

![Class Diagram](image1)

**Figure B.28: Microwave Oven Model (C+D)+(A+B) [threshold \( \leq 0.70 \)]**
Microwave Oven Model C+D [0.75 ≤ threshold ≤ 0.85]

Figure B.29 shows the result of combining models C and D with the class similarity threshold between 0.75 and 0.85. The MicrowaveOven class of model C is merged with its corresponding version in model D. The Oven&RecipeTimer and Oven&RecipeTimer&MinutePlus classes are merged with each other.

(a) Class Diagram.

(b) Oven & RecipeTimer/Oven & RecipeTimer & MinutePlus State Machine.

Figure B.29: Microwave Oven Model C+D [0.75 ≤ threshold ≤ 0.85]
**Microwave Oven Model A+B \[0.75 \leq \text{threshold} \leq 0.85\]**

Figure B.30 shows the result of combining models A and B with the class similarity threshold between 0.75 and 0.85. The MicrowaveOven class of model A is merged with its corresponding version in model B. The OvenTimer and OvenTimer&MinutePlus classes are merged with each other.

(a) Class Diagram.

(b) OvenTimer/OvenTimer & MinutePlus State Machine.

**Figure B.30: Microwave Oven Model A+B \[0.75 \leq \text{threshold} \leq 0.85\]**
**Microwave Oven Model (C+D)+(A+B) [0.75 ≤ threshold ≤ 0.85]**

Figure B.31 shows the result of combining models C+D and A+B with the class similarity threshold between 0.75 and 0.85. The Microwave Oven classes of all models are merged with each other. The Oven & RecipeTimer/Oven & RecipeTimer & MinutePlus and OvenTimer/OvenTimer & MinutePlus classes are copied to the result as is.

![Class Diagram](image1)

(a) Class Diagram.

![OvenTimer/OvenTimer & MinutePlus State Machine](image2)

(b) OvenTimer/OvenTimer & MinutePlus State Machine.

![Oven & RecipeTimer/Oven & RecipeTimer & MinutePlus State Machine](image3)

(c) Oven & RecipeTimer/Oven & RecipeTimer & MinutePlus State Machine.

**Figure B.31: Microwave Oven Model (C+D)+(A+B) [0.75 ≤ threshold ≤ 0.85]**
Microwave Oven Model C+D [threshold ≥ 0.90]

Figures B.32 and B.33 show the result of combining models C and D with the class similarity threshold of 0.90 or higher. The MicrowaveOven class of model C is merged with its corresponding version in model D. The Oven&RecipeTimer and Oven&RecipeTimer&MinutePlus classes are copied to the result as is.

Figure B.32: Microwave Oven Model C+D [threshold ≥ 0.90] - Part 1
Figure B.33: Microwave Oven Model C+D [threshold $\geq 0.90$] - Part 2
Microwave Oven Model A+B [threshold ≥ 0.90]

Figure B.34 shows the result of combining models A and B with the class similarity threshold of 0.90 or higher. The MicrowaveOven class of model A is merged with its corresponding version in model B. The OvenTimer and OvenTimer&MinutePlus classes are copied to the result as is.

Figure B.34: Microwave Oven Model A+B [threshold ≥ 0.90]
Microwave Oven Model (C+D)+(A+B) [threshold ≥ 0.90]

Figures B.35 and B.36 show the result of combining models C+D and A+B with the class similarity threshold of 0.90 or higher. The MicrowaveOven classes of all models are merged with each other. The remaining classes – OvenTimer, OvenTimer&MinutePlus, Oven&RecipeTimer and Oven&RecipeTimer&MinutePlus are copied to the result as is.

Figure B.35: Microwave Oven Model (C+D)+(A+B) [threshold ≥ 0.90] - Part 1
(a) Oven & RecipeTimer State Machine.

(b) Oven & RecipeTimer & MinutePlus State Machine.

Figure B.36: Microwave Oven Model (C+D)+(A+B) [threshold ≥ 0.90] - Part 2