Mixsets: Combining Annotative and Compositional Approaches to Variability and Product Lines

Abdulaziz Algablan

A thesis submitted to the Faculty of Engineering in

partial fulfillment of the requirements for the degree

Doctorate in Philosophy (PhD) in Computer Science

Ottawa-Carleton Institute for Computer Science

School of Electrical Engineering and Computer Science

University of Ottawa

Ottawa, Ontario, K1N 6N5

Canada

© Abdulaziz Algablan, Ottawa, Canada, 2021

Abstract

In this thesis, we present mixsets, an approach to combine annotative and compositional fragments for specifying code variants to form software product lines (SPLs). There are three key contributions of our research: introducing mixsets to represent software variability, extending mixsets to construct feature models, and improving software composition to achieve fine-grained variability.

The concept of mixsets is introduced in Umple as a conditional unit and a first-class entity to allow smoothly transitioning software to compositional SPLs. A mixset is a named set of mixins; each mixin belonging to the mixset is called a fragment. A mixset fragment can be a top-level entity that contains nested entities or can be embedded as a conditional fragment in other entities such as methods. Mixset content normally includes code blocks or statements of any type, and may include require statements, which describe explicit dependencies among mixsets. Mixsets can be used to specify product lines using code composition, code annotation or both. A strength of mixsets lies on the straightforward mechanism to transform annotated segments into compositional segments when used in a combined approach. Therefore, preplanning effort and time to transform annotative SPLs to compositional SPLs can be reduced.

Mixsets can provide a backbone structure to realize product line features in a feature-based SPL. Hence, a feature model can be formed using a subset of specific dependent mixsets. Feature interactions within an SPL can be identified and separated in specific modules by mixset nesting. Furthermore, product configuration, feature modeling analysis and generation of feature diagrams can be accomplished based on mixsets.

The thesis also demonstrates a method to enable the granularity of compositional approaches to be expressed at the statement level. This is achieved by allowing aspects to inject code into labelled places, or points of variation, within method code. Such injected code has a full access to the context in which its placed, such as access to local variables.

Mixsets are implemented in Umple, a model-driven development (MDD) technology that allows combining abstract models, such as associations and state machines, with pure code, and generating code in multiple programming languages. Mixsets can thus be used to describe variations in models, not just code.

The thesis uses a design science approach as its research methodology. The mixset concept is evaluated through three case studies. The first applies mixsets to the Oracle Berkeley Database SPL, which is used in other literature to evaluate SPL concepts. This study shows that the mixset implementation improves on the state of the art in several respects, such as elimination of complex workarounds that are required by other approaches. In the second study, mixsets help to refactor Umple into feature-oriented SPL. The case study shows how annotative fragments can be easily transformed into compositional counterparts. In the third study, mixsets help to present two alternative solutions the to the Rover Control Challenge Problem of the MDETools 2018 workshop.

Acknowledgment

All praises are due to God (Allah), the most merciful and most compassionate. I am so grateful to Allah for all his bounty bestowed upon me including the completion and writing this thesis. All perfect praises belong to the Almighty alone.

Throughout pursing my PhD degree and writing of this dissertation I have received a great deal of support and assistance.

First, I thank my supervisor, Professor Timothy Lethbridge, whose expertise has crucial significance in the research direction, the methodology, and the effectiveness of the thesis' writing. Dr. Lethbridge is a supervisor, a mentor, and a friend to whom I always enjoy talking. Thank you, Tim, for all insightful feedback that pushed me to sharpen my thinking. Honestly, your guidance brought the research to a higher level.

Second, I would like to thank my PhD thesis committee for their insightful comments and their fruitful discussions: Prof. Daniel Amyot, Prof. Thomas Kühne (Victoria Univ. of Wellington, New Zealand), Prof. Dwight Deugo (Carleton Univ.), Prof. Stéphane Somé. Special thanks to Dr. Amyot for his valuable feedback that he offered in the proposal; "it truly opened my eyes".

Third, I am infinitely grateful to my family and my friends. My mother, father, brothers, and sisters were all sources of love, support, and inspiration. May Allah bless you with good health, long life, and infinite success. My wife and my kids always encouraged me to accomplish this achievement; May Allah grant you Barakah in all your affairs. To all my dear friends, I'm so blessed of having you.

Last but not least, I would also like to acknowledge the financial support that I have received from Qassim University, the Ministry of Education, and their representatives in the Saudi bureau office here in Ottawa, Canada.

List of Abbreviations

The list below includes frequently used abbreviations:

AFM: Aspectual Feature Module

AOP: Aspect-oriented Programming

AST: Abstract Syntax Tree

CIDE: Colored Integrated Development Environment

CLI: Command Line Interface

CPP: C Preprocessor

CVL: Common Variability Language

DM: Decision modeling

DOP: Delta-oriented Programming

DSL: Domain Specific language

EBNF: Extended Backus–Naur Form

FM: Feature Modeling

FOP: Feature-oriented Programming

FOSD: Feature-oriented Software Development

GUI: Graphical User Interface

IDE: Integrated Development Environment.

JVM: Java Virtual Machine

MDSD (or simply MDD): Model-driven Software Development

OCL: Object Constraint Language

OOP: Object Oriented Programming

PLD: Product-line Development

SAT: Satisfiability Solver

SPL: Software Product Line

SPLE: Software Product Line Engineering

UML: The Unified Modeling Language

XML: Extensible Markup Language

Table of Contents

Abstract ... ii
Acknowledgment ... iv
Table of Contents ... vii
List of Figures ... xiii
List of Tables ... xv
Chapter 1 Introduction ... 1
1.1 Thesis Topic and Contributions ... 3
1.2 Motivation and Problem Statement ... 5
1.3 Research Methodology ... 6
1.4 Research Questions ... 8
1.5 Papers Published So Far About this Work ... 9
1.6 Research Outline ... 9
Chapter 2 Background ... 12
2.1 Conditional Compilation ... 12
2.1.1 The C Preprocessor ... 13
2.2 Model Driven Development (MDD) ... 15
2.3 Variability Modeling ... 16
2.4 Source-Level Software Composition ... 17
2.4.1 Syntactic Superimposition ... 17
2.4.2 Semantic Weaving ... 18
2.4.3 Syntax-directed Editing ... 18
2.4.4 Mixins ... 18
2.5 Reuse ... 19
2.6 Separation of Concerns ... 20
2.6.1 Variability within Aspect Models ... 20

2.6.2 Synergy between Reuse, Separation of Concern and Variability 21
2.7 Software Product Lines 22
2.7.1 Feature-oriented Software Development (FOSD) 24
2.7.2 Models Used in Problem Space Variability 24
2.7.3 Mechanisms for Solution Space Variability 26
2.8 An Overview of Umple 27
Chapter 3 Introducing the Concept of Mixsets 32
3.1 Mixsets as Annotative Variability 32
3.2 Mixsets as Compositional Variability 34
3.3 Unified Representation for Mixsets 35
3.4 Refactoring Between Compositional and Annotative Mixsets 37
3.5 Feature Modeling Using Mixsets 38
3.6 Relationship and Nesting between Mixsets Types 41
3.7 Mixset Metamodel 47
3.8 Syntax Checking for Mixsets 48
3.9 Summary 51
Chapter 4 Umple Extensions to Incorporate Mixsets 52
4.1 Why Umple? 52
4.2 Mixins: The Foundation for Mixsets 53
4.3 Mixset Basics 55
4.4 Basic Mixset Cases 56
4.5 Grammar for Mixsets 57
4.6 General Algorithm for Processing Mixsets 58
4.7 Nesting of Mixsets 61
4.8 Dependencies among Mixsets 61
4.9 Feature Diagrams 63

Chapter 5 Mixsets for Fine-Grained Feature-Oriented Software Product Lines 65
5.1 Background 65
5.1.1 Compositional SPL Techniques 66
5.2 The Complexity of Handling Fine-grained Variability 72
5.2.1 Statement Extension Problem 73
5.2.2 Local Variables Access Problem 73
5.2.3 Local Variable Modification Problem 74
5.2.4 Scope-Sensitive Statement Problem 74
5.2.5 Exception Handling Problem 74
5.3 Handling Fine-grained Variability in Compositional Approaches 75
5.3.1 Hook Methods 75
5.3.2 Combining Annotation with Composition 75
5.4 Motivation and Idea 76
5.4.1 Code Labels to Support Fine Grained Variability 76
5.4.2 Code Injection Providing Direct Access to Method Context 78
5.4.3 HelloWorld SPL Example 79
5.4.4 Code Injection Grammar 81
5.5 Limitations 82
5.5.1 Expression-level Variability 82
5.5.2 Code Label Support in Programming Languages 83
5.5.3 AOP to Apply Statement Composition 83
5.5.4 Code Comprehension 84
5.5.5 Dynamic Variability 85
5.5.6 Type Checking 86
5.6 Summary 86

Chapter 6 Berkeley BD JE Compositional Case Study 87
6.1 Berkeley DB Java Edition 87
6.2 Procedure 88
6.2.1 Umplification of Berkeley DB JE 88
6.2.2 Retain Original Bodies Out from Chains of Hook Methods 89
6.2.3 Remove Static Inner Classes 89
6.2.4 Erase Customized Throw Statements 89
6.2.5 Example 90
6.2.6 Validation of Mixsets' Implementation 92
6.3 Analysis and Discussion 94
6.3.1 Labels vs. Hook Methods as Variability Mechanism 94
6.3.2 Comparison between Mixsets and FeatureHouse 95
6.3.3 Why Not AOP? 96
6.3.4 Generalization of Result 96
6.3.5 Mixsets Against CIDE 96
6.3.6 Possible Refactoring to Improve the FeatureHouse Implementation 97
6.4 Summary 97
Chapter 7 Refactoring Umple into a Feature-Oriented SPL 98
7.1 Refactoring Objectives 98
7.2 Overview of the Umple Codebase 99
7.2.1 Umple Architecture 99
7.3 The Refactoring Process 101
7.3.1 Scope of Refactoring 101
7.3.2 Procedure Followed for Refactoring 101
7.3.3 Feature Identification and Modeling 104
7.3.4 Feature Catalogue 106

7.4 Outcomes ... 107
7.4.1 Granularity of Feature Mixsets ... 108
7.4.2 A Deep Look at Fragments ... 109
7.4.3 The Argument for Fully Compositional Fragments ... 112
7.4.4 Feature Model Specification ... 112
7.5 Observation and Challenges ... 113
7.5.1 Feature Identification and Localization ... 113
7.5.2 Grammar Files ... 113
7.5.3 Test Units ... 114
7.5.4 Refactoring during Continuous Integration ... 115
7.5.5 GitHub Logs to Identify Features ... 115
7.6 Summary ... 116
Chapter 8 Applying Mixsets to the Rover Control Challenge Problem ... 118
8.1 Method Used to Develop the Solutions ... 119
8.2 Version 1: Converting to Umple and Getting a Basic Controller Working ... 119
8.3 Version 2: The ‘Frequency Modulation’ Solution ... 121
8.4 Version 2a: Improving Model Quality and Exploring Parameter Variation ... 123
8.5 Version 3: The ‘Amplitude Modulation’ Solution ... 125
8.6 Conclusions and Key Lessons Learned ... 127
Chapter 9 Evaluation ... 129
9.1 Comparing Mixsets Against Related SPL Techniques ... 129
9.1.1 Preplanning Effort ... 130
9.1.2 Adoptability ... 132
9.1.3 Separation of Concerns ... 132
9.1.4 Traceability ... 133
9.1.5 Information Hiding ... 133

9.1.6 Granularity 134

9.1.7 Uniformity 134

9.1.8 Language Independence 135

9.2 Tool Support Comparison 135

9.2.1 Comparison Criteria 136

9.3 Limitations 140

9.3.1 Labeled Aspect Breaching of Information Hiding 140

9.3.2 Insufficient Evaluation for Mixsets as Proactive SPL 141

9.3.3 Biases of Mixsets Evaluation 142

9.3.4 Limitation of Mixsets Currently to Umple 142

Chapter 10 Conclusions and Future Work 144

10.1 Contributions 145

10.2 Future Work 147

List of Figures

Figure 1: Relationship between domains, models, and metamodels in MDD (Stahl et al., 2006). 16
Figure 2: Overview of domain engineering and application engineering process in SPL adapted from Apel et al. (Apel et al., 2013, p. 20). 23
Figure 3: Example of feature model for a mobile phone product line taken from (Czarnecki et al., 2012) 25
Figure 4: Assisting the variant selection based on quality objectives in Clafer (Murashkin et al., 2013). 26
Figure 5: Annotated class diagram (Czarnecki and Antkiewicz, 2005). 27
Figure 6: A class digram, genereted by Umple, for the bank class. 30
Figure 7: Bank system with MultiBranch and OverdraftsAllowed. 33
Figure 8: Bank system with neither optional feature. 33
Figure 9: Rewriting the inline mixset "Multibranch" into a compositional form. On the right, the abstract syntax tree shows the modification at the token level. 36
Figure 10: The inline mixset "HalfOpenFeature" is nested in two levels in the state machine model. The blue highlighted code belongs to HalfOpen feature in both the code and the diagram.. 37
Figure 11: Two ways of refactoring of mixsets. 38
Figure 12: The corresponding feature diagram notation for the first require statement. 39
Figure 13: Feature models will be represented in the solution space as mixsets instead of being configuration parameters (dark scribbled rectangle). 42
Figure 14: Direct mapping between of SPL features in the problem space and feature mixset (right). 43
Figure 15: A Feature may map to multiple mixsets in the Strict mode. 44
Figure 16: Three modes to include mixsets in source code. Flexibility increases downward. 47
Figure 17: Mixsets metamodel augmented with a simplified fragment of the original Umple metamodel (highlighted in the blue box). 48
Figure 18: GarageDoor state machine model on left. Green shaded area contains pure Java code added to the model. 50

Figure 19: A trait implementing an observer design pattern. 54
Figure 20: The feature diagram generated from the mobile SPL code mentioned in Section 3.5 63
Figure 21: Dependencies among mixsets is shown in a separate diagram using UmpleOnline... 64
Figure 22: Jak composition to form Java files (Batory et al., 2004). 70
Figure 23: Java code and its corresponding FST (Apel, Kästner, et al., 2009) 71
Figure 24: Umple code in (a) and the generated Java code in (b), showing activation of features with fine-grained variability. 78
Figure 25: Feature diagram generated by Umple for the SPL according to lines 36-38 of the code. The SPL base is the default-selected feature (gray color), while selected features are in green. 80
Figure 26: Example of multiple labels found in a method taken from the compositional UmpleSPL. 85
Figure 27: Feature diagram for Berkeley DB JE SPL as illustrated in (Kästner et al., 2007). 88
Figure 28: Excerpt of original implementation of the BIN class. 92
Figure 29:Types of injection used in mixset approach in the implementation of Berkeley DB JE. The right circle breaks out the 911 after cases. 95
Figure 30: The blue highlighted code of the top-left rectangle is refactored into an inline mixset and then into a compositional mixset. 103
Figure 31: Umple menu headings. 105
Figure 32: Umple feature model. 106
Figure 33:Git log for the user "opeyemiAdesina", who implemented some functionality related to alloy and NuXmV, generator. 116
Figure 34: Umple-generated diagram of version 1 of the solution to the challenge problem with a basic dumb follow method 123
Figure 35: Class diagram from version 2 with a moderately intelligent follow method 124
Figure 36: File and feature model. 125
Figure 37: Direction control state machine as a diagram 126
Figure 38: A modified DoC example to show the idea of restricting the variability inside methods using Contracts for Java, or Cofoja (Finney, 2011). 141

List of Tables

Table 1: Common CPP directives 14
Table 2 : Separation of concern/reuse techniques and variability mechanisms in compositional paradigms. 22
Table 3: A decision model for the mobile phone SPL shown in Figure 3 (Schaefer et al., 2012). 25
Table 4: Nesting mixsets. 45
Table 5: Extension types resulting from refactoring Berkeley DB JE into FOSD. 88
Table 6: FeatureHouse v.s. Mixset implementation of IsValidDelete() in Berkeley Latch feature. 93
Table 7: Mixset v.s. FeatureHouse implementation of Berkeley DB JE. 95
Table 8: Description of refactored features in Umple. 106
Table 9: Total number of fragments of each feature mixset. 107
Table 10: Total number of fragments of each non-feature mixset. 108
Table 11: Fragment type of feature mixsets. 109
Table 12: Fragment details 110
Table 13. Test results for FM versions 2 and 2a (10 tries each) 123
Table 14. Test results for AM version 3 (10 tries) 127
Table 15: A comparison of our approach to CPP, FOP, AFM, and FeatureCoPP. 129
Table 16. Advantages of certain approaches to managing variants and product lines 140

Chapter 1 Introduction

In this thesis, we introduce the concept of *mixsets*, a flexible textual approach to separation of concerns that can be applied to both models and code in software product line development. A mixset is a set of independent fragments of code, including textual representations of model elements, that can be combined in various ways to produce features or variants of products.

Many software projects need to create variants, which are different versions of a product or component, targeted at different clients or hardware. Variation among versions of software systems can be achieved in several ways. For instance, using version control technology that makes certain branches active, design patterns, conditional compilation, invocation parameters at run time, and particular frameworks. All these techniques come with advantages and disadvantages related to pre-planning, understandability, feature separation, composability, and defect-proneness.

Software product line engineering (SPLE), product-line development (PLD) or *product family* development refers to the disciplines to develop of a portfolio of related software systems called a software product line (SPL) using a common set of reusable assets in a prescribed way (Pohl et al., 2005). SPLE capitalizes on the systematic breakup of software systems into common and variant components, through which generation of diverse products is feasible. Approaches to SPL development with a focus on *features* as essential units that span the software development cycle, are called *feature-oriented software development (FOSD)* (Apel and Kästner, 2009). In FOSD, the features can be added to a common code codebase to form variant software systems. Both SPLE and FOSD are cases where SPLs can be systematically generated, whereas *feature separation* as *concerns* is central for the later. A feature in this context is an increment to the software system's functionality (Batory, 2005).

There are two main techniques that enable systematic SPL design and implementation: code *composition* and code *annotation* (Kästner and Apel, 2008a). In code composition, SPLs are formed by dividing software into separate pieces of code. A variant, or a product, is built by combining some or all pieces together. The special reusable piece which is a shared by all variants

is often called the SPL base. In code annotation, specific markings (annotations) are used to identify parts of code that belong to a certain feature in one monolithic source code.

Compositional approaches to SPL provide useful benefits to product line systems; these include separation of concern and clear traceability of SPL features. Composition is advocated in the SPL literature as a solution to the drawbacks of annotative approaches such as hard-to-understand code, lack of separation of concerns, and lack of reusability. In practice, however, annotative approaches still prevail (Kästner et al., 2012; Liebig et al., 2010; Spinellis, 2008; Zhang et al., 2016)– notably for open-source projects. The first reason for the prevalent use of annotative approaches is the straightforward transformation of a unitary software system into product lines: A new feature or variant simply requires adding extra fragments, annotated to mark them as optional. Optional elements can be included into a configuration via conditional compilation in annotative approaches. The second reason for embracing annotative approaches is their support for fine-grained variability: Even a very small adjustment to an expression such as negation of a Boolean expression can be annotated so that the negation only occurs in certain configurations. This can result in less effort to make changes and requires less *pre-planning*. By pre-planning we mean determining how to organize the code before writing it or before creating multiple variants.

A further desirable approach to software development is *model-driven software development* (MDSD or simply MDD) (Atkinson and Kühne, 2003). This means building the software from high-level abstractions that can be in general be expressed as diagrams, although perhaps also textually. SPLE can utilize MDD to construct SPLs through sets of model variants, which could be behavioral, structural, or non-functional models, that are managed, linked, and configured by variability models (Czarnecki et al., 2005).

Still another desirable approach to facilitate software development is to enable all software elements to be expressible in a human-editable textual form that can be subject to version-control, commenting as well as textual search, and can be edited by a wide variety of tools (Lethbridge et al., 2016). In such a case, the input language would need to consist of a blend of textual modeling constructs and code written in one or more programming language.

There are many languages and technologies, described in Chapter 2 , which enable FOSD, PLD or other forms of separation of concerns; there are numerous modeling languages or tools

that allow expression of software at a high level of abstraction; and there are many text-based languages for both modeling (Merkle, 2010) and variants (Eichelberger and Schmid, 2013). But prior to the work described in this thesis we believe that there was no approach that provided a SPL variability mechanism allowing annotation and composition uniformly at the same time. We will treat the feasibility of achieving this as a hypothesis to be tested in this research.

This thesis therefore has the objective of creating a capability to enable as many forms of separation of concerns as possible that all operate harmoniously together with each other, in a textual language that can incorporate both modeling abstractions and code. A further objective is to compare the developed capability with existing tools, in terms of its comprehensiveness and applicability, using case studies.

The research method we are following is design-based, case-study-based and technology-comparison based. In other words, the designed technology has to a) meet the above objective, documenting the rationale and alternatives considered; b) demonstrate its novelty to the SPL development through a series of case studies; and c) show notable benefits when it is compared to other technologies and tools available.

1.1 Thesis Topic and Contributions

There are three key contributions of this thesis: introduce mixsets to represent software variability, extend mixsets to construct feature models, and improve software composition to achieve fine-grained variability.

The concept of mixsets stems from the well-known conditional compilation mechanism of the C preprocessor (CPP), with its famous "#ifdef" statements (Stallman and Weinberg, 1987), as well as the concept of mixins as found in several programming languages (Batory et al., 2004). Conditional compilation is very useful and used heavily in C/C++ programs to offer variability (Liebig et al., 2010). The notion of mixins is extended, beyond inclusion of components (e.g. class fields and methods) into classes, to be a superimposition technique (Apel and Lengauer, 2008) to merge elements based on matching criteria.

As we will demonstrate later, mixsets are similar to #ifdef statements as conditional units; however, they are first-class entities that are partially parsed even when they are not activated. A

mixset is a named set of fragments, each fragment encapsulates either a set of entities that can appear at the top level of a language (e.g. classes), or a set of inner elements of a language construct (e.g. instance variables of a class). The content of mixsets can be classified into core content and metacontent. Content adds entities to the source code while metacontent establishes relationships among mixsets. By default, a mixset' core content is ignored during language parsing unless the mixset is *activated*. However, the metacontent is always parsed, but does contribute directly to the executable system.

Therefore, an *activation* construct for mixsets is necessary in any mixset implementation. Boolean expressions can be used to specify explicit dependencies among mixsets. Violation of mixsets' dependencies should raise warnings or error messages.

Mixsets allow software variability in both an annotative and a compositional form. Unlike annotative approaches, fragments of mixsets represented in annotative form have a direct compositional match. By this we mean that the fragment can be positioned annotatively in place with other code, or else extracted such that it is separate from the other code, requiring composition to weave it back into the base code at compile time; the resulting compiled system would be the same in either case. Since the decision of how to organize the code does not have to be made before writing the code, planning effort and time can be saved when using mixsets to create SPLs.

Mixsets explicitly capture dependencies and relationships among the various elements through require statements. Feature modeling can be also achieved using a subset of specific dependent "feature" mixsets. Require statements, nesting, and activation of mixsets via use statements can be further restricted for an SPL through mixset modes.

The thesis also demonstrates a method to extend the granularity of compositional approaches to the statement-level. Hence, aspects specified in compositional fragments are directly able to inject code into labelled places, or points of variation, within method code.

As a key part of this research, mixsets have been implemented in Umple, a model-driven development (MDD) technology that allows combining abstract models, such as associations and state machines, with pure code, and generating code in multiple programming languages. Mixsets

can thus be used to describe variations in models, not just code. We will discuss Umple in detail in Section 2.8 of Chapter 2 .

The evaluation phase of this research is by case studies, including applying mixsets to the Oracle Berkeley Database SPL (Kästner et al., 2007), which is used in other literature to evaluate SPL concepts. This study shows that the mixset implementation improves on the state of the art in several respects, such as elimination of complex workarounds that are required by other approaches.

1.2 Motivation and Problem Statement

The motivation and initial idea of an improved generic variability mechanism emerged from several experiences.

Firstly, we wanted a way to create generic models of systems that could become the basis of numerous products. For example, Levin investigated how various designs of time and activity tracking systems could be generated from a common model (Levin, 2009). Our first step in this direction was a tool called VML4Umple (Levin, 2009). VML4Umple defined variability in containment hierarchal units called concerns. A concern consists of the variation type (such as required, optional, alternative, etc.) as well as available implementation for each option. VML4Umple described variations in a rigid structure that was not composable and was not sufficient to express variability at a fine level of granularity including inner entities and variation in source code. We thus needed an approach that was more flexible.

A second motivation was that we wanted to be able to simplify the understanding of Umple itself (which is written in Umple). We wanted to be able to work with simpler variants of Umple, including to draw diagrams of these, or to create versions that were simpler for the purpose of experimentation.

The problem which this thesis seeks to solve can be stated as follows:

Existing software modelling technologies make it difficult for software developers to build systems with separate concerns or variants that can be combined to create product lines, and that allow conditional incorporation of both model elements and traditional source code

elements. For example, there may need to be alternative algorithms for different platforms, or different features deployed on different hardware. Product line technology for managing code is well developed, including traditional C conditional compilation; however, this technology has not yet been extended to work with software modeling constructs such as state machines and UML associations.

The answer this thesis provides to the problem statement is an effective and lightweight mechanism that enables variability in existing unitary software systems with minimal effort and changes. The required variability mechanism is expressive enough to handle fragments specified in both annotative and in compositional forms, operates on both models and code (including at fine-grained granularity within methods), is designed to be simple to use, and has properties that make it clearly better in other ways than existing approaches, such that it being orthogonal to existing concern separation mechanisms (aspects, traits, interfaces) and has the potential to be widely adopted.

Compositional approaches to SPL provide useful benefits such as separation of concerns and enabling traceability of SPL features. However, SPL research shows that compositional approaches are not well-adopted in practice to construct SPLs. There are several reasons for this low adoption that are discussed in the SPL literature: the complexity to add fragments of features to existing code using separate modules in certain situations and the limited support for fine-grained variability. The effort to construct SPLs using software composition would require considerable changes and preplanning, especially in the case of transforming singular software into SPL, or when augmenting an SPL with a new feature.

1.3 Research Methodology

In this thesis, we follow the design science research process as research methodology. This approach offers a concise conceptual framework and useful guidelines for understanding, executing, and evaluating research in the information systems and software engineering disciplines (Hevner et al., 2004). Design science research is a problem-solving paradigm that iterates over two basic activities: building and evaluation. The building phase is a process to design an artifact that serves a specific purpose; the evaluation phase assesses the effectiveness of the proposed design and offers feedback for improvement.

The process starts with identifying research problem; we expressed this in the previous section. In simple terms it is to find a new and creative solution to SPL variability representation.

As we will show, the design and implementation of mixsets were shaped iteratively. We started by identifying weaknesses in existing technology, including our own VML4Umple, and explored various syntactic and semantic alternatives. We implemented various ideas in Umple, and we worked through various case studies and examples adjusting our approach as we uncovered strengths, weaknesses and ideas for improvement.

We made various decisions, such as:

- To reuse existing syntactic and semantic concepts in Umple, and make our mixset implementation an extension of Umple. This was decided since Umple is open source, we have control over it, we want variability in Umple anyway, and it has properties we needed such as managing both models (e.g. state machines) and traditional code uniformly. This decision has the following implications:
  - The *use* statement is already used in Umple to include Umple files, which end with the ".ump" extension. We decided to use same keyword to activate mixsets since we realized they are conceptually like virtual files, and indeed could be converted to and from files.
  - The *require* keyword is used in Umple trait models to specify required elements such as states and events for state machines. The same keyword is used for mixsets' require statement, but to specify dependencies among mixsets.
  - Square brackets are used to specify constraints in OCL-type Boolean constraints that Umple support. We use square brackets to specify constraints among mixsets in require statements.
- To reuse existing concepts in programming languages. This is applied to:
  - Reuse aspect injection to insert code inside methods, since aspect-oriented programming (AOP) provides the lowest granularity for a compositional approach (Kästner, Apel, and Kuhlemann, 2008).

- Utilize code labels in programming languages to indicate places of variability inside methods.

- To introduce keywords for new concepts with names that are meaningful. The concept of mixsets is new to Umple and it means a group (set) of mixins. We intend this term to be generic, and not specific to Umple.

1.4 Research Questions

Our objectives in this research are to answer the following research questions. We will elaborate and summarize the final answers to these questions in the concluding chapter (Chapter 10):

RQ1: What should be the syntax and semantics of a text-based language easily usable by developers that can operate on models and code to enable effective SPL development enabling SPL composition and annotation to work synergistically together to form SPLs?

There are several motivations for this research question. Firstly, we want to overcome some of the limitations of existing SPL mechanisms that follow purely compositional or purely annotative approaches, that we will discuss in this thesis. Secondly, we want to create an approach that can operate on and blend with multiple languages simultaneously including modeling languages, to help overcome challenges in adoption of model-driven technology. Thirdly, we want a language that is not intended primarily for machines to read and write (such as XML). Finally, we want the language to be textual so that it can work well with the other textual tools widely used by developers.

RQ2: How do existing approaches compare with the language designed as the answer to the first research question, RQ1, in terms of the types of capabilities available?

The motivation for this question is that we want the technology we develop to be clearly better, in ways such as comprehensiveness, adoptability, reduction in need for pre-planning, and granularity, than alternative technologies, so we can claim we have made a useful advance on the state-of-the-art that might either be adopted directly in industry, or serve to inspire further advances.

1.5 Papers Published So Far About this Work

So far, we have published two papers related to this work. Some material in this thesis is adapted from those documents.

The first paper (Lethbridge and Algablan, 2018b) was published and presented at the Isola conference, and covers the core concepts of mixsets. The author of this thesis performed the literature review (which forms much of Chapter 2 of this thesis) and the comparative analysis (which forms much of Section 9.2 in this thesis). The author of this thesis was responsible for all the implementation of mixsets and the examples as described in the paper, which form much of Chapter 3 of this thesis. The supervisor helped substantially with other parts of the paper.

The second paper (Lethbridge and Algablan, 2018a) was published in the MDETools workshop at Models 2018 and is a response to a modeling challenge in which we applied mixsets. The author of this thesis provided the mixsets implementation on which this case study was based; that case study is presented Chapter 8 of this thesis.

1.6 Research Outline

The rest of the thesis is organized according to this outline:

- Chapter 2 : Background.

  This chapter provides the core background material, which is needed to understand the thesis, including conditional compilation, MDD, variability modeling, software reuse, separation of concern, source-level software composition, SPL, and an overview of Umple.

- Chapter 3 : Introducing the Concept of Mixsets.

  The chapter's goal is to show how mixsets are used to express annotative and compositional variability together and to configure software product variants.

- Chapter 4 : Umple Extensions to Incorporate Mixsets

  In this chapter we demonstrate how mixsets are implemented in Umple. It covers: the foundational mixins as in Umple, the grammar and use cases of mixsets, the algorithm to process them, and specification of dependencies and feature diagramming using mixsets.

- Chapter 5 : Mixsets for Fine-Grained Feature-Oriented Software Product Lines.

  This chapter draws attention to the workarounds in current compositional approaches: their causes, and their negative impact on code understandability and code size. Then it discusses the way in which mixsets (as a compositional approach) achieve fine-grained granularity at the statement level.

- Chapter 6 : Berkeley BD JE Compositional Case Study.

  The Java version of the Berkeley DB is taken as a case study. The chapter shows how mixsets eliminate the need to have many workarounds when applying statement level changes. The Berkeley DB is a known case study and has been referred in different annotative and compositional SPL techniques.

- Chapter 7 : Refactoring Umple into a Feature-Oriented SPL.

  Chapter 7 elaborates on the experience and the challenges of refactoring a portion of Umple from a unitary software system into a feature-oriented SPL using mixsets. The case study aims to contribute to the SPL community with some insights about refactoring a modeling tool into a feature-oriented SPL.

- Chapter 8 : Applying Mixsets to the Rover Control Challenge Problem.

  Mixsets represent two SPL variants; one contains pure code and the second integrates state machines as modeling constructs. They are used to provide solutions to the Challenge Problem for the MDETools 2018 workshop. The mixsets are used to switch different control algorithms, and their dependencies on or off.

- Chapter 9 : Evaluation.

  This chapter evaluates mixsets as a combined SPL variability approach against annotative and compositional SPL technologies in first part of the chapter. The second part is a comparison between mixsets and related techniques based on tool support. The last part discusses the potential limitations and the weaknesses of mixsets as a concept, our implementation and of the thesis in general. In addition, we will offer some suggested ways to mitigate these limitations.

- Chapter 10 : Conclusions and Future Work.

  This chapter concludes the thesis, summarizes the contributes, and discusses potential future work.

Chapter 2 Background

This chapter explores the core background material needed to understand the thesis. It reviews the following topics: conditional compilation, MDD and variability modeling, software reuse and separation of concerns, superimposition of software artifacts, software product lines, and Umple. These topics are important to understand the context and technologies forming the motivation, foundation and platform for the mixsets capability.

2.1 Conditional Compilation

Conditional compilation is a mechanism to embed variation in a language via its compiler or a preprocessor through including, or excluding, certain pieces of code based on some parameters and on expressions forming Boolean expressions involving those parameters. These parameters are either specified by the user or determined by querying the operating system. Conditional compilation has proved vital to implement software systems that are compiled for multiple hardware, operating systems, and virtual-machine platforms, as well as to systems that allow fine-tuned customization (Dimovski et al., 2018).

Programming languages such as C/C++, C#, and Swift natively support conditional compilation. However, this is not the case for most language compilers. For example, the compiler of Java does not support any kind of preprocessor directives. Therefore, Java users may rely on a third-party processor, implemented specifically to work on Java source files, such as Manifold (McKinney, 2019), JCPP (Mankin, 2008) and Antenna (Pleumann et al., 2002). Conditional compilation is not always tied to specific programming languages, software product line development tools such as pure::variants (Beuche, 2003) and Gears (Krueger and Clements, 2013) employ this mechanism to support software variability (Kästner and Apel, 2008a).

Although conditional compilation offers simple yet scalable variability in an extremely fine-grained and efficient way, it has gained a lot of criticism due to the tangled and scattered nature of the resulting code chunks (Adams et al., 2009; Medeiros et al., 2015). Early experience with conditional compilation led to code that is hard to understand, maintain, and to reason about because of conditional fragments (Spencer and Collyer, 1992). Furthermore, finding errors is

tedious and time-consuming for conditional code (Schulze et al., 2014). On the other hand, some authors argue that the harms that #ifdef directives bring to software systems might be exaggerated because they are merely based on experience reports, opinions, and narrowed-scope empirical studies (Fenske et al., 2020). In practice, the technique remains one of the favored variability techniques, especially for software systems written in C and C++ (Adams et al., 2009; Ernst et al., 2002).

2.1.1 The C Preprocessor

The C preprocessor (CPP) is a textual processing tool that performs preliminary operations on C/C++ source code before the code is parsed. It empowers the C and C++ languages with capabilities to specify conditional regions in code, to make brief abbreviations for extended constructs, and to include files, particularly files representing other architectural units of the system, and those containing declarations of libraries to be used.

CPP can be used on top of source files of any language because it is a generic preprocessor. It has been widely adopted for C/C++ in open-source projects and in industry (Hunsen et al., 2016; Liebig et al., 2010). It enables software variability for a long list of popular software systems in the open-source world, including the Linux kernel, Python, Gimp, and PHP (Liebig et al., 2010).

A conditional compilation fragment in CPP begins with a hash symbol (#) followed by a keyword called a preprocessing directive. Some directives require a closing keyword to mark the end of scope.

Table 1: Common CPP directives

Directive	Description	Example
#include <[*header_file*]>	Include a particular header from another file.	#include <stdio.h>
#define [*identifier*] [*value*]	Define a preprocessor macro.	#define PI 3.1415 #define MULT(x, y) x * y

#ifdef [*identifier*] [*value*]
#endif
	A conditional control evaluates whether a macro is defined.	#ifdef DEBUG
		printf("….");
		#endif

#ifndef [*identifier*] [*value*] #endif	It is opposite to #ifdef. It evaluates whether a macro is not defined before.	#ifndef __A_GUARD #define __A_GUARD #endif

Table 1 demonstrates four common directives in CPP. The #include directive logically copies the content of a file to the file where #include is found. The #define directive defines macros of object-like or function-like types. An object macro is an identifier that will be replaced by its value during preprocessing. A function-like macro such as `MULT(x, y)` can be used like a C function but with limited tool support to check syntax and data types. The #ifdef directive inserts conditional source code into a program if a #define for the same identifier has been encountered. The #ifndef is opposite to #ifdef in that the inclusion of conditional code occurs when there is no #define for the identifier.

Compound conditions in CPP can be formed by directive nesting or through #if directive, which can be followed by multiple #defined(…) statements joined by logical and comparison operators as follows:

- Conjunction: #if defined(*identifier1*) && defined(*identifier2*)
- Disjunction: #if defined(*identifier1*) || defined(*identifier2*)
- Exclusive disjunction (XOR): #if defined(*identifier1*) ^ defined(*identifier2*)
- Comparison operators: #if (*identifier1)* >= (*identifier2)*

CPP directives in practice often form *disciplined annotations*, while a few make up *undisciplined annotations* (Liebig et al., 2011). A disciplined annotation designates a complete

syntactic structure of a language like a whole method or a sequence of lines in a method. An undisciplined annotation contains a partial structure (tokens) such as a bracket. Undisciplined annotations are considered a bad practice and result in code that is hard to understand and error-prone (Malaquias et al., 2017).

2.2 Model Driven Development (MDD)

Software modeling reduces the complexity of software systems during development and maintenance by capturing abstract aspects of a software system from various viewpoints. Model driven development (MDD) helps in automating aspects of software production by generation of source code in whole, or in part, starting with *models* rendered either in general-purpose modeling languages like UML (*Unified Modeling Language*, 2017), SysML (Hause and others, 2006) or Umple (Lethbridge et al., 2016), or else in domain specific modeling languages (DSLs). Such languages describe the software at a higher level of abstraction than traditional source code (Brown et al., 2004; Selic, 2006)

The goals of modeling with high-level abstractions in MDD are to simplify and also to formalize (as standardizing enables automation) various activities in the software development cycle (Hailpern and Tarr, 2006). This simplification and formalization should also lead to software with fewer errors that is developed more quickly. Models are primary artifacts in the MDD process and they play an important role to increase the return on investment in software development (Atkinson and Kühne, 2003; Brambilla et al., 2017). The Object Management Group (OMG) introduced many standards including Model Driven Architecture (MDA) to standardize and realize MDD (Soley and others, 2000). MDD is also an integral part in agile model-driven development (Ambler, 2004) , and domain-oriented programming (Hailpern and Tarr, 2006; Thomas and Barry, 2003).

Modeling languages are in part described by *metamodels* that capture the abstract syntax of the language (i.e. the entities and relationships that can be modelled). A good MDD tool ensures models' well-formedness against their metamodels (Stahl et al., 2006).

A domain-specific language (DSL or DSML) is a modeling language targeted at a particular domain (e.g. e-commerce, automotive control) which is expressed by a metamodel that precisely describes key parts of the domain including its semantics and constraints (Schmidt, 2006).

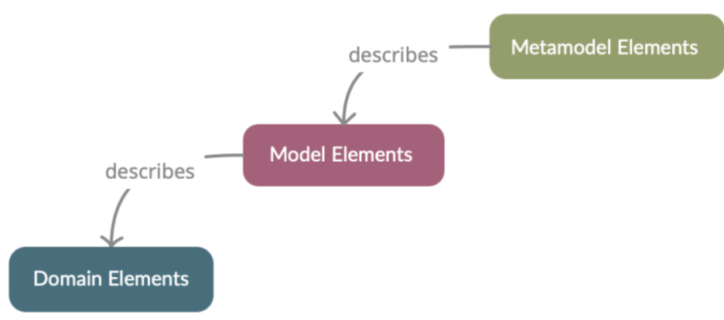

Figure 1: Relationship between domains, models, and metamodels in MDD (Stahl et al., 2006).

MDD transformation engines and generators take as input supplied models, and synthesize new models, code, or other types of artifacts (Stahl et al., 2006). MDD best practices discourage software engineers from applying changes to auto-generated code; modifications should be applied to models instead.

Models are realized through a textual notation, graphical notation, or both forms. Textual syntax encodes information serially and sequentially processed by humans whereas graphical syntaxes encode information as spatial arrangements of symbols (often combined with text) that can be perceived in a parallel manner (Moody, 2009). Text-based models take the advantages of easing consistency checking, modifiability, tool adoption, and integration with version control systems (Grönninger et al., 2014; Meliá et al., 2016). On the other hand, visual-based models are often more intuitive (Grönninger et al., 2014).

2.3 Variability Modeling

In the software product line context, variability models characterize commonalties and variabilities of a software system's artefacts. They describe domain-specific properties and their dependencies (Schaefer et al., 2012). They capture explicit relationships among related models to construct different systems (Czarnecki, 2013; Schaefer et al., 2012). The interest in modeling variability within SPLs has resulted in development of a multitude of variability modeling

languages (mostly textual) differing in capabilities and scale (Eichelberger and Schmid, 2015). Different techniques have been developed to allow people to interact with the models, including filters, zooming, and cross tree views used in visual variability models, but visualization can be difficult for large variability models (Bosch et al., 2015).

Variability modeling constructs can be included within software artifacts or can be specified in separate models that relate the artifacts; examples of this are the orthogonal variability model (OVL) (Pohl et al., 2005) and CVL (Haugen et al., 2013).

2.4 *Source-Level Software Composition*

In this section we will discuss composition of software, where the elements being composed are source code or models. The term software composition also has a broader meaning that includes composition of systems from compiled components, such as when one 'wires' or 'orchestrates' the components via network connections, pipes or sockets. We will consider the latter to be out of scope in this thesis.

Starting from bottom to top, software systems compose code units such as methods, classes, interfaces, packages, components up to services. Software composition in general describes the process of bundling a set of software artifacts for the sake of building software systems. An artifact includes any kind of information that results from software system development activities such as source code units (packages, classes, methods, property files, test cases, etc.) and supporting documents (models, documentation, build files, etc.) (Glinz, 2011).

2.4.1 *Syntactic Superimposition*

Superimposition is a technique to implement software composition through matching and merging of software artifacts according to their names and positions within hierarchical structures (Apel and Lengauer, 2008). For example, superimposing two Java files that contain definitions of classes with the same name, but with different fields, methods and nested classes, results in one Java file. This file contains a single class with all the fields, methods, and nested classes (the merge is recursively applied to inner elements). In this paradigm, artifact merging requires static matching of some key elements; hence, it is a syntax-driven technique (Apel and Lengauer, 2008). Superimposition is expressive and powerful enough to merge diverse models and artifacts (Apel,

Janda, et al., 2009). It offers a foundation for collaboration-based designs (Smaragdakis and Batory, 2002), feature-oriented programming (Prehofer, 2006), subject-oriented programming (Harrison and Ossher, 1993), and aspect-oriented programming (Apel and Lengauer, 2008; Mezini and Ostermann, 2003).

2.4.2 *Semantic Weaving*

Semantic-based weaving has been used to compose aspects in design models (Jézéquel, 2008; Klein et al., 2006; Mussbacher et al., 2009). The merge mechanism in semantic weaving goes beyond syntactic structural matching and requires more sophisticated algorithms to properly inject aspects into target models. The merge process takes into consideration semantics-related issues such multiple occurrences, loops, and naming variations. Although requirement and design models may benefit from semantics-based weaving, we have observed no application for semantics-based weaving in merging code at the implementation level.

2.4.3 *Syntax-directed Editing*

Composition of models and code can be also achieved via *syntax-directed editing* (Behringer et al., 2017; Voelter et al., 2014). This technique maps artifacts to internal representations that assign a unique ID for each element of an artifact; it uses IDs as refences to accomplish direct editing. For example, a model represented as an XML file can be composed with a fragment that consists of a new XML tag and the element ID on the target model. A key property of the syntax-directed editing is that changes (and new increments also) are applied directly to the internal representations. If the internal representation is an abstract syntax tree (AST), appending it with new elements does not require re-parsing the AST to accommodate the modifications. Although syntax-directed editing is suitable for manipulating graphical notations in IDEs, it has gained only very limited popularity to manipulate textual languages due to its lack of integration with mainstream software development tools (Voelter et al., 2014).

2.4.4 *Mixins*

In this research, the notion of *mixins* is used as a superimposition foundation for model composition. A mixin is a repeated occurrence of a top-level entity with a matching name to add new content to the named entity (i.e. mixes it in). It represents a generic unit of change composable to classes, interfaces, state machines, and other modeling elements. In a similar way, module

refinements in feature-oriented programming (FOP) are called mixins (Batory et al., 2004). For example, a mixin may consist of a set of extensions to a model that reflects a use-case in the software design document (SDD).

In the broader programming language context, however, the term mixin has various partially-conflicting meanings. These meanings include:

- A linearization mechanism, in which a class will compose methods hailing from separate superclasses (in a language permitting multiple inheritance) (Bracha and Cook, 1990). This meaning is applicable to virtual methods in Python and C++.
- Implementation of a subclass which extends a parameterized superclass (Bracha and Cook, 1990; Smaragdakis and Batory, 2002). Templates in C++ can implement this type of mixin.
- Attachment of methods to classes at runtime without inheritance. JavaScript and Ruby allow this kind of dynamic binding of methods (Osmani, 2012, p. 83).
- The inclusion process of traits (Schärli et al., 2003). For example, the process to include traits to classes uses the term "mixin" in Scala (Odersky et al., 2008, p. 696).

2.5 Reuse

Software reuse is important to increase productivity and quality of software systems during development (Frakes and Kang, 2005). The economic factor of minimizing time to design and implementation and thus reducing time to market is a primary factor to adopt software reuse (Capilla et al., 2019); another factor is avoiding defects, because the more code present, the higher the chance of defects.

The spectrum of software reuse ranges from code modules to components and commercial off-the-shelf (COTS) approaches. The potential of reuse can further include documentation, designs, and models. Reuse of code and components is known to be easier and more productive that other forms of reuse (Capilla et al., 2019).

Reuse can be facilitated by various separation-of-concerns methods discussed in this chapter. For example, traits can be used to group pure methods as building blocks to be reused by multiple classes in a programming language that does not support multiple inheritance (Schärli et al., 2003).

Bettini et al. showed the feasibility of traits to model commonality and variability of SPLs (Bettini et al., 2015).

2.6 Separation of Concerns

A fundamental principle to modularize design and implementation of software systems is the *separation of concern* principle (Dijkstra, 1968), indicating that items belonging to related concerns should be placed together in distinct code units like procedures, modules, and classes, whereas items belonging to different concerns should be separated. The separation can be based on several reasons including logical, architectural, and organizational factors. However, there are often concerns that are inherently difficult to separate in mainstream programming languages because their code is scattered over multiple hierarchies and block structures. These concerns are known as *crosscutting* concerns. Crosscutting concerns are challenging because of what is known as the *tyranny of the dominant decomposition:* programming languages generally permit a single way to decompose programs (Tarr et al., 1999).

2.6.1 Variability within Aspect Models

Aspect-oriented programming is a mechanism that utilizes *aspects* to separate and modularize crosscutting concerns (Kiczales et al., 1997). Aspect-oriented modeling (AOM) further expands the notion, by capturing crosscutting concerns within software models at various levels of abstraction (Klein and Kienzle, 2007). AOM is a compositional mechanism, and several approaches to achieve it have been introduced. In particular, the aspect notation has been often used to capture and document variability to implement software product lines.

There are approaches that demonstrate the usefulness of aspects to model variable and common concerns in the contexts of software product lines (Heo and Choi, 2006; Voelter and Groher, 2007). Noda et al. used a variability mechanism that captures explicit variability rules and constraints to manage optional and alternative aspects (Noda and Kishi, 2008). Reusable Aspect Models (RAM) integrates three AOM design-level techniques (class diagram, sequence diagram and state diagram) in reusable aspects (Kienzle et al., 2009). The aspect weaver of RAM allows specifying and handling dependencies between aspect models. Combemale et al. combined the Common Variability Language (CVL) (Haugen et al., 2013), which handles feature modeling and its resolution, with RAM (Combemale et al., 2012). Their approach uses direct mapping between

CVL choices and their aspect models containing RAMs to compose structural and behavioral models.

There are also approaches that offer generic aspect-oriented composers applicable on domain specific modeling languages (DSML) (Kienzle et al., 2009; Morin et al., 2008). Variability is achieved through aspects in these composers; however, variation within aspects themself are not straightforward to implement; it may require further integration with external tools.

There is research that introduces variability within aspects themselves. Lahire et al. argue that implementing variability to aspects results in models that are both more flexible and more reusable (Lahire et al., 2007). Lahire et al. implemented an aspect variability mechanism that augments aspects with composition protocols. These protocols state the required adaptation in the variable aspects before composing them to their target models. Their approach allows alternatives, options, and constraints within aspect models.

2.6.2 Synergy between Reuse, Separation of Concern and Variability

In the literature, variability models such as feature models are often mapped into components that are processed via compositional paradigms (mostly aspect models), but few approaches have expanded the variability scope into compositional techniques themselves. Introducing variability to a wide range of reusable elements (such as models and traits) and separation of concern constructs (including aspects and mixins) as well as to models containing pure code has not been explored. Hence, a variation of a software system can modeled as variations within models, as separate aspects, or as variable aspects (aspects having variations).

Although we cannot claim that our research is comprehensive, we have not come across approaches that propose to synergically combine reuse, separation of concerns, and variability. Hence, a combination of reuse and variability modeling seamlessly accommodates not only aspects but inclusive to various reusing capability such as mixins, traits, aspects, and language-specific modeling capabilities all together. Table 2 shows compositional paradigms that enable SPL and whether its SPL variability technique allows variability (within) and with other reuse techniques.

Table 2 : Separation of concern/reuse techniques and variability mechanisms in compositional paradigms.

Approach	Reuse Technique Support			SPL Variability	
	Aspects	Traits	Mixin	Technique	Variability within the technique
FOP **(Prehofer, 2006)**	⚐	⚐	⚑	Mixin	⚑
Heo et al. **(Heo and Choi, 2006)**	⚑	⚐	⚐	Aspect	⚐
Lahire et al. **(Lahire et al., 2007)**	⚑	⚐	⚐	Aspect	⚑
Noda and Kishi **(Noda and Kishi, 2008)**	⚑	⚐	⚐	Aspect	⚐
GeKo **(Morin et al., 2008)**	⚑	⚐	⚐	Aspect	⚐
Aspectual Module **(Apel et al., 2008)**	⚑	⚐	⚑	Aspect & Mixin	⚐
RAM **(Combemale et al., 2012)**	⚑	⚐	⚐	Aspect	⚐
FPTJ **(Bettini et al., 2015)**	⚐	⚑	⚐	Trait	⚐

2.7 *Software Product Lines*

Software product line engineering (SPLE) is based on the systematic reuse of shared assets in a planned manner to efficiently develop a family of software products tailored to a particular domain involving subtle variations (Clements and Northrop, 2002).

There are several sources for variation in software product lines. These include varying stakeholders' needs, and factors such as varying hardware resources and variants of dependent technology. Large organizations such as Boeing, Bosch Group, Philips, and Siemens have embraced SPL development to achieve quality improvement and to reduce development time in highly competitive markets (Bosch and Lee, 2010; Clements and Northrop, 2002).

SPLE brings major modeling challenges to traditional software development – specifically in variability modeling, variability implementation, and product derivation (El-Sharkawy et al., 2019). As in Figure 2, SPLE consists of two main processes: domain engineering and application engineering (Czarnecki and Ulrich, 2000). The former has the goal of building customizable

software platforms out of reusable components with well-defined commonality and variability (in requirements, design, realization, tests, etc.) while the later helps to derive products from a variability-aware platform.

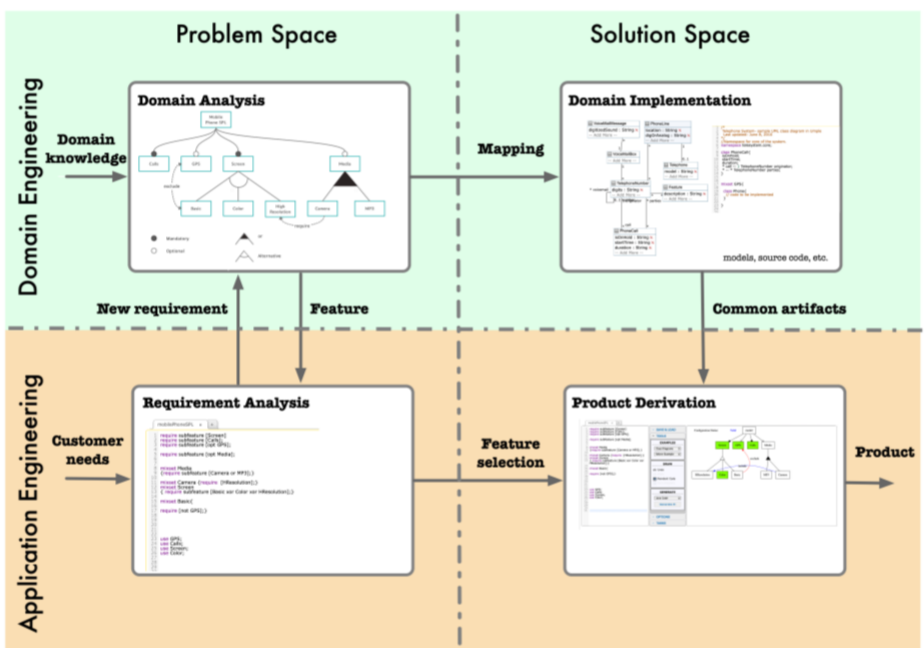

Figure 2: Overview of domain engineering and application engineering process in SPL adapted from Apel et al. (Apel et al., 2013, p. 20).

Problem space variability tackles extracting and describing variability across domain engineering and application engineering to produce customizable software systems that often involve producing variability models in the form of feature diagrams. Solution space variability aims to design and implement reusable SPL implementations (architectures), which involves establishing configurable infrastructures to reuse and represent a wide range of related components with ability to encompass their commonalities and differences. Product derivation utilizes such designs to construct software products based on a set of preferred configurations. Research to tackle issues related to variability modeling and variability management issues has been growing in both academia and industry (Chacón-Luna et al., 2020; Paulisch et al., 2018).

There are three strategies to adopt SPL: proactive, reactive, and extractive (Krueger, 2002). A software system follows a proactive approach when it has been preplanned to be a SPL from the beginning. On the other hand, a SPL follows a reactive approach when it is refactored from an

existing unitary software system into an SPL. An extractive approach refers to the case of extending a system to include further features.

2.7.1 Feature-oriented Software Development (FOSD)

Feature-oriented software development (FOSD) advances SPLE further with the arrangement of SPLs according to their features, which represent functionality increments (Apel and Kästner, 2009). Features in this approach are the primary units of reuse and they guide the design and implementation of the SPL. Software systems adhering to FOSD should generate variants automatically form a selection of features. The software architecture may have a small core, with each new capability being added as a feature. A feature might be represented as a set of software deltas (in a similar manner to deltas in a version-control system), using some kind of conditional-compilation preprocessor such as the C preprocessor (Ernst et al., 2002), as aspects that can be woven together with the core, as mixins, or using some other similar approach. FeatureIDE brings together many of these approaches into a usable tool (Thüm et al., 2014).

FOSD has similarities to Feature Driven Software Development (FDSD) as FDSD breaks down software projects to features by following the notion "plan by feature, design by feature and build by feature" (Hunt, 2006). However, FOSD centers on the agility of software product development with no account to the development of product lines.

2.7.2 Models Used in Problem Space Variability

Feature modeling (FM) and *decision modeling* (DM) are common approaches to capture variability in SPLs. Feature modeling is more popular in academia (Berger et al., 2013) and originated from the work of Kang et al. on Feature-Oriented Domain Analysis (FODA) (Kang et al., 1990). Figure 3 shows a typical example of a feature model with basic notations: "mandatory", "optional", "or", "alternative (xor)", "requires" and "excludes". Research in feature modeling mainly focuses on the extending the basic notations, cardinality, constraint precision and derivation of valid configurations. Although FM is mostly a diagramming approach, there are textual FM languages (e.g. TVL (Classen et al., 2011), and Clafer (Bąk et al., 2016)) that have the capability of being human-readable and offer rich syntax to handle feature modeling concerns such as cardinalities, feature attributes, and so on, in addition to making feature modeling more intuitive

and concise. However, there is no standard way of mapping FM to other software artifacts such as design, or source code (Czarnecki et al., 2012)

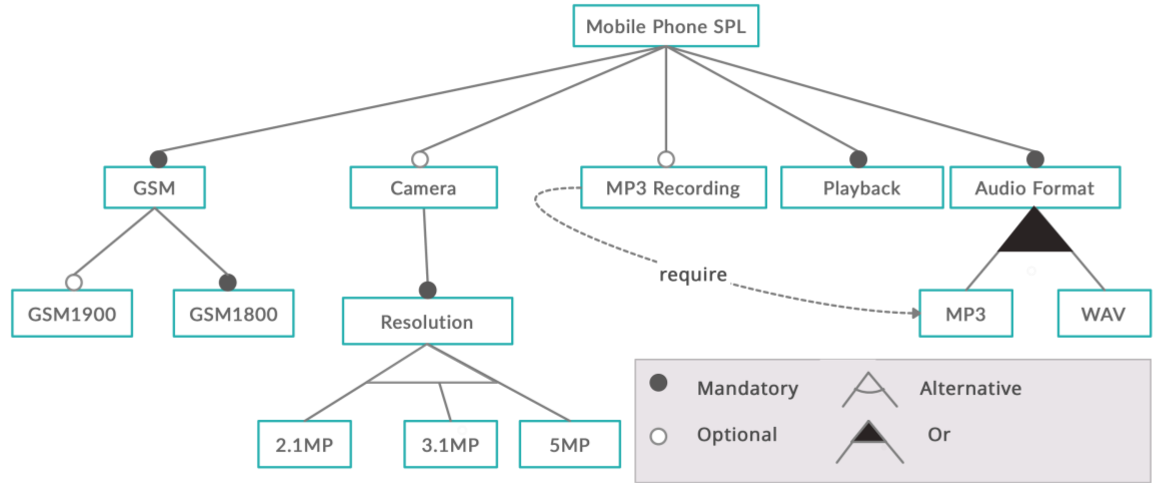

Figure 3: Example of feature model for a mobile phone product line taken from (Czarnecki et al., 2012)

DM approaches are influenced by the synthesis method, which exploits decision models; a decision model is a set of decisions that are adequate to differentiate family members of SPLs and they guide the derivation of a product in the application engineering process (VA, 1993). DM is represented by a decision table like Table 3 and hence, it is more toward product step-wise derivation of software than describing the product line domain.

Table 3: A decision model for the mobile phone SPL shown in Figure 3 (Schaefer et al., 2012).

ID	Question	Range	Cardinality	Constraints
GPS	Do you want GPS?	Yes/no	1	GPS.yes excludes Screen.Basic
ScreenType	Which type of screen do you want?	Basic| color | high resolution	1:1	Screen.Basic excludes GPS.yes
SupportedMedia	Which media shall be supported?	Camera, Mp3	0.2	SupportedMedia.Camera => ScreenType.High resolution

Automated reasoning to validate variability models or to calibrate their properties is achieved via propositional logic, constraint programming, or description logic (DL) (Jézéquel, 2012). For

example, the selection of optimal variants – with respect to some qualitive measures – is achieved in Clafer via feature qualitive attributes, which can take numerical values and arbitrary constraints (Murashkin et al., 2013). Addressing qualitive aspects in Clafer provides the means to visualize differences among variants with respect to their qualities, offers trade-offs, and help in selecting desirable variants. Figure 4 shows how Clafer visually assists in the selection of SPL variants based on predefined quality objectives.

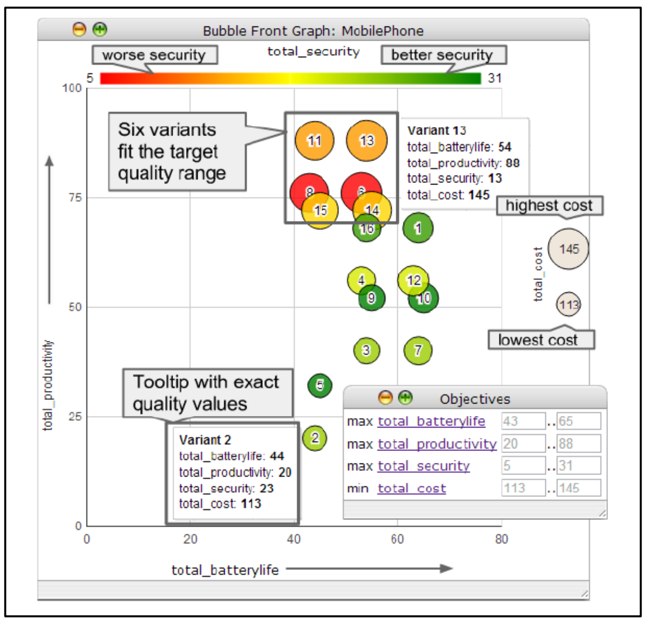

Figure 4: Assisting the variant selection based on quality objectives in Clafer (Murashkin et al., 2013).

2.7.3 Mechanisms for Solution Space Variability

Solution space variability facilitates the fulfillment of the problem space variability. Proactive SPLs anticipate and preplan this kind of variability in architectural elements, components, test cases, and design documentations. Nonetheless, capturing variability in solution space is more complex as it spreads across various artifacts in different levels including design models and source code.

While there are various ways to implement variability into software such as inheritance, design patterns, frameworks, runtime parameters and clone-and-own, the issue of these techniques is interwinding source code with SPL features whereas SPL variants are often required to be sliced out from large codebases (Zhang et al., 2016). Therefore, there are approaches that seek improved

management of SPL variant separation and explicit variability modeling. These approaches can be categorized as annotative and compositional variability techniques (Voelter and Groher, 2007).

Annotative approaches use a single model containing all variants of the SPL. Also, it has certain conditions (parameters) to enable variation. UML stereotypes (Gomaa, 2004) and presence conditions (e.g. Figure 5 (Czarnecki and Antkiewicz, 2005)) are two examples of annotative approaches at the model level. The venerable C preprocessor and Colored Featherweight Java (Kästner and Apel, 2008b) are examples of code-level annotative approaches.

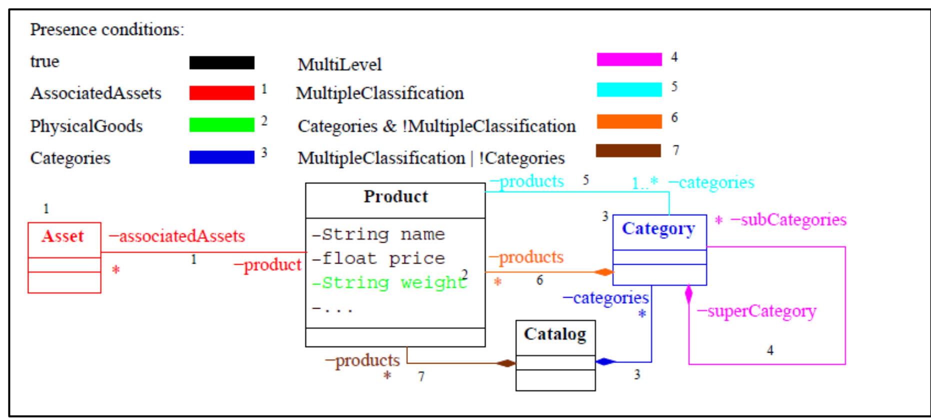

Figure 5: Annotated class diagram (Czarnecki and Antkiewicz, 2005).

Compositional approaches build upon reusable core models (base models), which often are not producing products themselves alone, but are subject to more refinements and are an integral part of SPL variants. Hence, a variant often adds new content, or fragments, to the original base models. Although there is a wide spectrum of compositional approaches, superimposition (Apel and Lengauer, 2008) is the more relevant approach to our research.

Delta modeling, a transformational approach, extends the composition power to delete members of a reused element. ABS applies delta programming and modeling concepts through different sub-languages called language layers to express feature modeling, delta modeling, configuration, and selection of variants (Clarke et al., 2011).

2.8 An Overview of Umple

Umple is a technology for developing software that incorporates both traditional code (organized as classes with methods) as well as modeling abstractions (Lethbridge, Forward, et al.,

2021). Modeling abstractions that Umple incorporates include class models with UML attributes (Badreddin, Forward, et al., 2014a), associations (Badreddin, Forward, et al., 2014b) and generalizations, as well as state machine models (Badreddin, Lethbridge, et al., 2014) containing events, hierarchical states, and transitions (Lethbridge et al., 2016). Umple generates code from the models and incorporates both generated code and user-supplied methods to build complete systems. It can incorporate and generate code written in multiple programming languages simultaneously.

Umple has been designed to be familiar-looking to developers, and to enable them to use it in conjunction with arbitrary text editors, version-control tools and build tools. Central to this is its textual form that looks like a C-family programming language, it has command-line, Eclipse plugin, and web-based tool support, so can be used in almost any toolchain. Its only dependency is that it runs on a Java virtual machine (v 8 or higher), and is not tied to any external metamodeling or IDE technology.

Features of particular relevance to this thesis are Umple's multiple approaches to separation of concerns and modularity that all work synergistically together: Classes (with inheritance), files, mixins, traits and aspects.

Like other object-oriented technologies, Umple organizes software primarily into classes and interfaces. Classes can contain methods as in Java and C++, but also attributes (like variables but with richer semantics), associations, state machines and various other entities.

In addition to classes and interfaces, Umple also has *traits* (Abdelzad and Lethbridge, 2017). Traits can be used to *pull* in pieces of common functionality (methods, variables, state machines, and so on) into a class without the need for multiple inheritance. By 'pull', we mean that an entity (here a class, or another trait) requests inclusion of common functionality found in a trait.

Furthermore, Umple also supports *aspects*. These are used to *push* functionality into multiple methods. By 'push', we mean that the instructions to inject functionality (in this context commonly called 'advice') has a mechanism (commonly called 'pointcuts') to include the injected functionality, via pattern matching, in various places among the existing functionality; the injection locations are commonly called 'join points'.

For simplicity we will refer to classes, interfaces and traits together as *classifiers*, following UML conventions.

A system is also divided into files, but a file can have multiple classifiers; and multiple files can contain multiple parts of the same classifiers: these multiple parts are *mixed* together (the parts are called *mixins*), to create complete classifiers. The developer can therefore organize the files however they want: Files could be organized so as to represent components, features, layers, or any other concern.

In Umple, each of the approaches to separation of concerns works with traditional code, with model entities, and with each other. This allows great flexibility regarding how developers can structure their system.

The following is a simple example of Umple code. The class diagram generated from this is in Figure 6.

```
1   class Bank {
2      1 -- * Account;
3   }
4
5   class Account {
6      owner; Integer number; Integer balance;
7   }
8
9   trait InterestBearingAccount {
10     Float interestRate;
11  }
12
13  class DepositAccount {
14     isA Account;
15  }
16
17  class LoanAccount {
18     isA Account, InterestBearingAccount;
19  }
```

- Line 2 describes an *association* between Bank and Account, with the '1' and '*' being the *multiplicities*.
- Lines 6 and 10 describe a total of 4 *attributes*. Umple uses String as the default type if it omitted.

- Lines 9-11 describe a *trait* that is incorporated into LoanAccount in line 18, essentially enabling a form of multiple inheritance.
- The ‘isA’ keyword is used for all forms of specialization, including specifying superclasses, implemented interfaces and used traits. Here there are isA statements on lines 14 and 18.

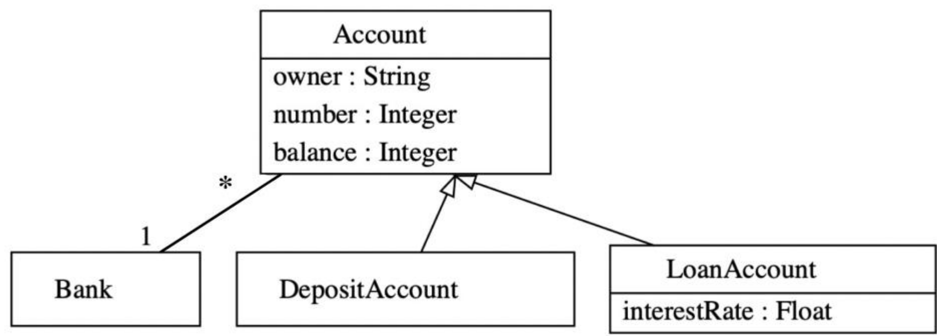

Figure 6: A class digram, genereted by Umple, for the bank class.

The Umple compiler works like other compilers in that it takes files in Umple syntax as input and generates various outputs including diagrams and code in target languages such as Java and C++. The compiler parses files (with suffix .ump) and builds an abstract syntax tree (AST); this is then analyzed to extract an internal model (conforming to Umple’s metamodel, available on its website (*Umple Metamodel*, 2021)) of the various classes and other model elements that are to be part of the system. Some of the items in the AST and model are chunks of target-language code such as method bodies that Umple does not parse but merely passes through to the generated target language code. To create a final executable system, a target-language compiler must then be applied to the output generated by Umple; however, in most toolchains this step would be invisible to the Umple user.

Umple’s incorporation of both models and code allow it to perform extensive analysis. The Umple compiler can detect many kinds of problems that could not be detected readily in technologies that just manage code or just manage models. During the Umple compiler’s analysis phase, it detects hundreds of potential types of problems in the model. The target-language code generated by Umple is hence free of numerous types of defects that might otherwise be present if a developer were programming in the target language directly. The generated code may still have

bugs in the chunks of passed-through target-language code, however Umple translates error messages raised by target-language compilers so they point at locations of such bugs in the original .ump files.

Code generated from Umple is not intended to be edited. It is supposed to be treated much like Java bytecode or machine code output from a typical C++ compiler. However, to raise the confidence of developers should they ever fear that Umple will disappear and to allow for inspection or auditing of code and models, the generated code in Umple is designed to be straightforward, self-contained and self-documenting. In particular, all comments present in the input Umple code are output in the generated code.

In addition to executable systems, Umple also generates various web-based diagram formats and documentation. These allow the reader to trace system elements back to their originating files (e.g. to places in files where traits, aspects and mixins are defined). This not only helps the developer understand the code, but also helps overcome one of the hazards of separation-of-concerns: Confusion and errors resulting from delocalization of related information.

The motivation for the work described in this thesis is to maintain and build on Umple's strengths outlined above. Although one can use previously-defined Umple separation-of-concerns mechanisms to build multiple variants of a system, we hypothesize that much can be gained by incorporating key ideas from existing technologies for product-line and feature modeling. Umple's new *mixset* concept (to be described in Chapter 3) allows mixins to be composed from multiple locations in a textual codebase, and constrained in a simple way. This mechanism enables Umple developers to apply many of the feature modeling or product-line modeling approaches reviewed in Sections 2.4.4, 2.5, and 2.6 .

Chapter 3 Introducing the Concept of Mixsets

In this chapter we introduce the concept of mixsets and show some of the power of applying mixsets to model variations. The aim is to show how mixsets are used to express annotative and compositional variability together, and to configure software product variants. We also discuss basic ideas about the use of mixsets for feature modeling and various issues related to checking the validity of the mixsets and their contents.

Some material, including examples, in this chapter are taken from a paper we published in the conference Isola 2018 (Lethbridge 2018).

3.1 Mixsets as Annotative Variability

Mixsets can be used as an annotative variability approach. Consider a bank SPL example, which describes generic software that can be used by various small banks. When a bank has multiple branches, the feature MultiBranch is required. Some banks may allow overdrafts on customer's deposit accounts and others will not.

The following is an Umple model showing the use of two mixsets. The third line of class Bank specifies that there will be an association with a Branch class only if the Multibranch mixset is included. Instances of the Account class are also linked to a particular branch only in the case of Multibranch, and the entire Branch class is present only if Multibranch is included. If OverdraftsAllowed is included, then the InterestBearingAccount trait is included in DepositAccount, as is overdraftLimit.

```
1   class Bank {
2     1 -- * Account;
3     mixset Multibranch
4     {
5       1 -- 1..* Branch;
6     }
7   }
8
9   mixset Multibranch {
10    class Branch {
11      Integer id; String address;
12    }
13  }
14
15  class Account {
```

```
16      owner; Integer number; Integer balance;
17      mixset Multibranch { * -- 1 Branch;}
18    }
19
20    trait InterestBearingAccount {
21      Float interestRate;
22    }
23
24    class DepositAccount {
25      isA Account;
26      mixset OverdraftsAllowed {
27        Integer overdraftLimit;
28        isA InterestBearingAccount;
29      }
30    }
31
32    class LoanAccount {
33      isA Account, InterestBearingAccount;
34    }
```

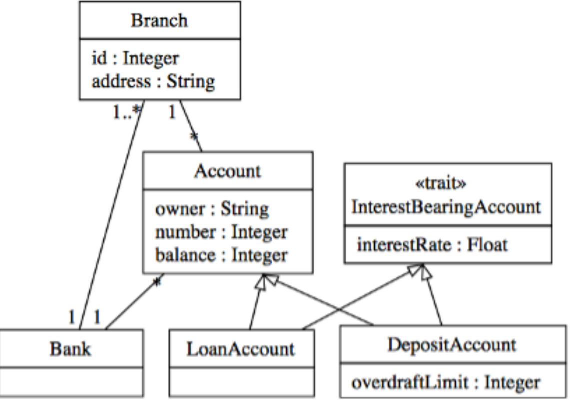

Figure 7: Bank system with MultiBranch and OverdraftsAllowed.

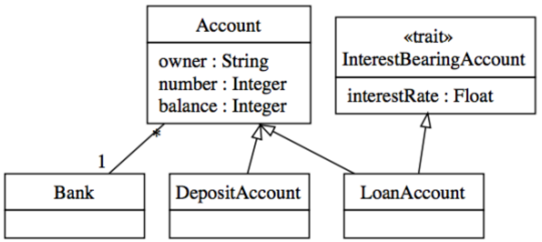

Figure 8: Bank system with neither optional feature.

When a mixset annotates certain part of a model or an entity, we will consider it an inline mixset. This definition applies to the four mixsets in the bank SPL code. Figure 7 (generated by Umple) shows the system with both mixsets included such as by compiling the system with the command line:

```
umple Bank.ump OverdraftsAllowed Multibranch
```

The second argument is the file to parse. The remaining arguments are considered to be mixsets because they do not end with ".ump" suffix. Figure 8 shows the system if neither of these mixsets are included.

3.2 *Mixsets as Compositional Variability*

The code/model presented in the annotative section distributed parts of the MultiBranch mixset among relevant entities. Some developers may instead prefer to group all the contents of the mixset in one place. Therefore, the base models, which represent the core models shared by all products, are separated from individual features. This alternative is shown below and illustrates the flexibility of the approach. The resulting system would be identical to that in the last section.

The base model of the bank SPL is store in a file called "BaseBank.ump":

```
1   class Bank {
2     1 -- * Account;
3   }
4
5   class Account {
6     owner; Integer number; Integer balance;
7   }
8
9   trait InterestBearingAccount {
10    Float interestRate;
11  }
12
13  class DepositAccount {
14    isA Account;
15  }
16
17  class LoanAccount {
18    isA Account, InterestBearingAccount;
19  }
20
21
```

The multibranch and overdraft models are stored in "BankFeatures.ump":

```
1   mixset OverdraftsAllowed
2   {
3     class DepositAccount {
4       Integer overdraftLimit;
5       isA InterestBearingAccount;
6     }
7   }
8
```

```
9   mixset Multibranch {
10    class Bank {1 -- 1..* Branch}
11    class Branch {Integer id; String address;}
12    class Account {* -- 1 Branch}
13  }
```

If the base model is stored in "BaseBank.ump" and the features stored in "BankFeatures.ump" are passed as arguments to the Umple parser, only the base models are parsed, because by default mixsets are switched off. To activate a mixset, it should be passed as an argument as well such as:

```
umple BaseBank.ump BankFeatures OverdraftsAllowed Multibranch
```

The two mixsets in this version of the bank SPL demonstrate compositional mixsets. A compositional mixset is a top entity mixset that contains an increment which is enclosed by the proper structure that identifies where to compose the increment. Compositional mixsets are unlike inline mixsets, they are free to be in separate files.

3.3 Unified Representation for Mixsets

Although mixsets offer flexibility to combine annotative variability and compositional variability, the underlining representation in the AST for both types is unified. At parsing time, annotative variability, which is manifested as inline mixsets, are transformed automatically into compositional mixsets. Therefore, inline mixsets can be considered another view of compositional mixsets. The opposite is possible; a preprocessor can direct the transformation towards a unified annotative representation in which all compositional mixsets are transformed into annotative ones. This option may suit languages that employ preprocessors while the first option is suitable for languages that have built-in mixin capability. We opted for annotative-to-composition option to implement mixsets since the host language supports mixins. The process of transforming annotative mixsets into their compositional counterparts is a straightforward rewriting process.

During the rewriting process, the parser pulls those inline mixsets that are embedded *inside* other entities (containers), out from their container, converts them into top-level entities, and then prepends their body with the container's signature. In terms of an abstract syntax tree, inline mixset rewriting transforms an inline mixset token to become a root token by deleting its link to

the parent token with slight adjustments to the body token. Figure 9 illustrates rewriting of the inline mixset “Multibranch” into a compositional form.

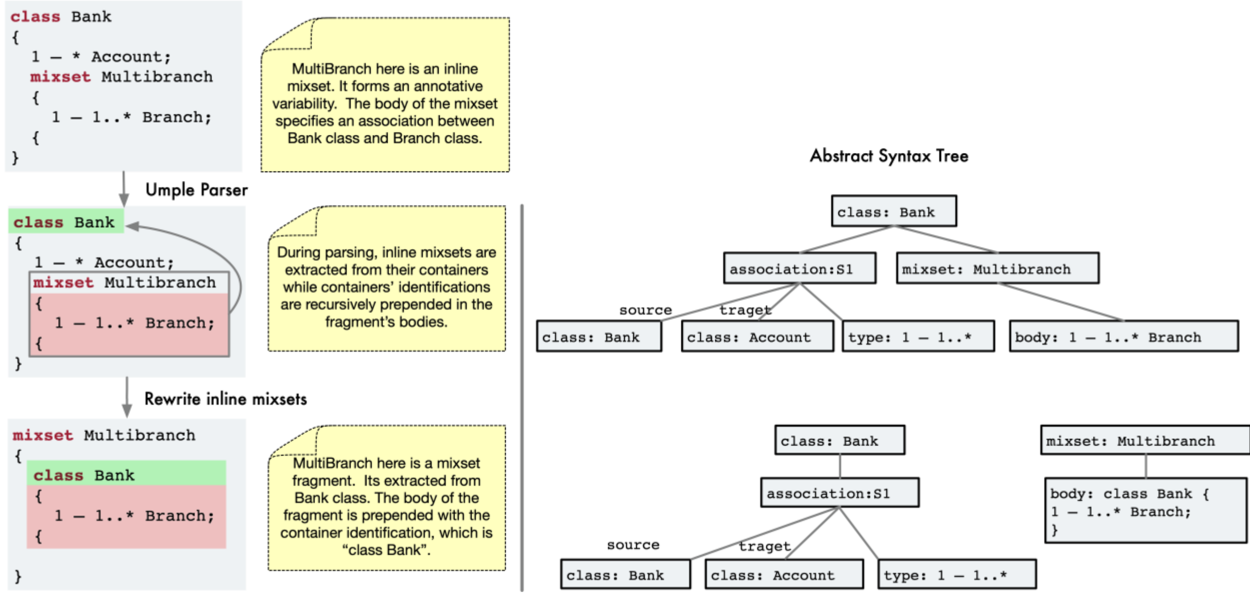

Figure 9: Rewriting the inline mixset “Multibranch” into a compositional form. On the right, the abstract syntax tree shows the modification at the token level.

When an inline mixset is placed in a container entity that is embedded in a hierarchy of entities, all signatures of the mixset’s parent entities are recursively prepended to its body. Figure 10 shows rewriting of the mixset “HalfOpenFeature”, which introduces to the GarageDoor an event inside “Opening” state and the “HalfOpen”” state. The rewritten mixsets at the bottom of the figure are prepended with the signatures “class GarageDoor { status { ” for the state “HalfOpen” in addition to “Opening {” for the event exiting the “Opening” state.

As inline mixsets’ bodies are directly prepended with container entities’ signatures, content particular to mixsets themselves such as feature modeling specifications, which will be introduced in Section 3.5, is not allowed in such cases. Otherwise, there would be a need for a filtering mechanism for the bodies of inline mixsets prior to the prepending of container signatures.

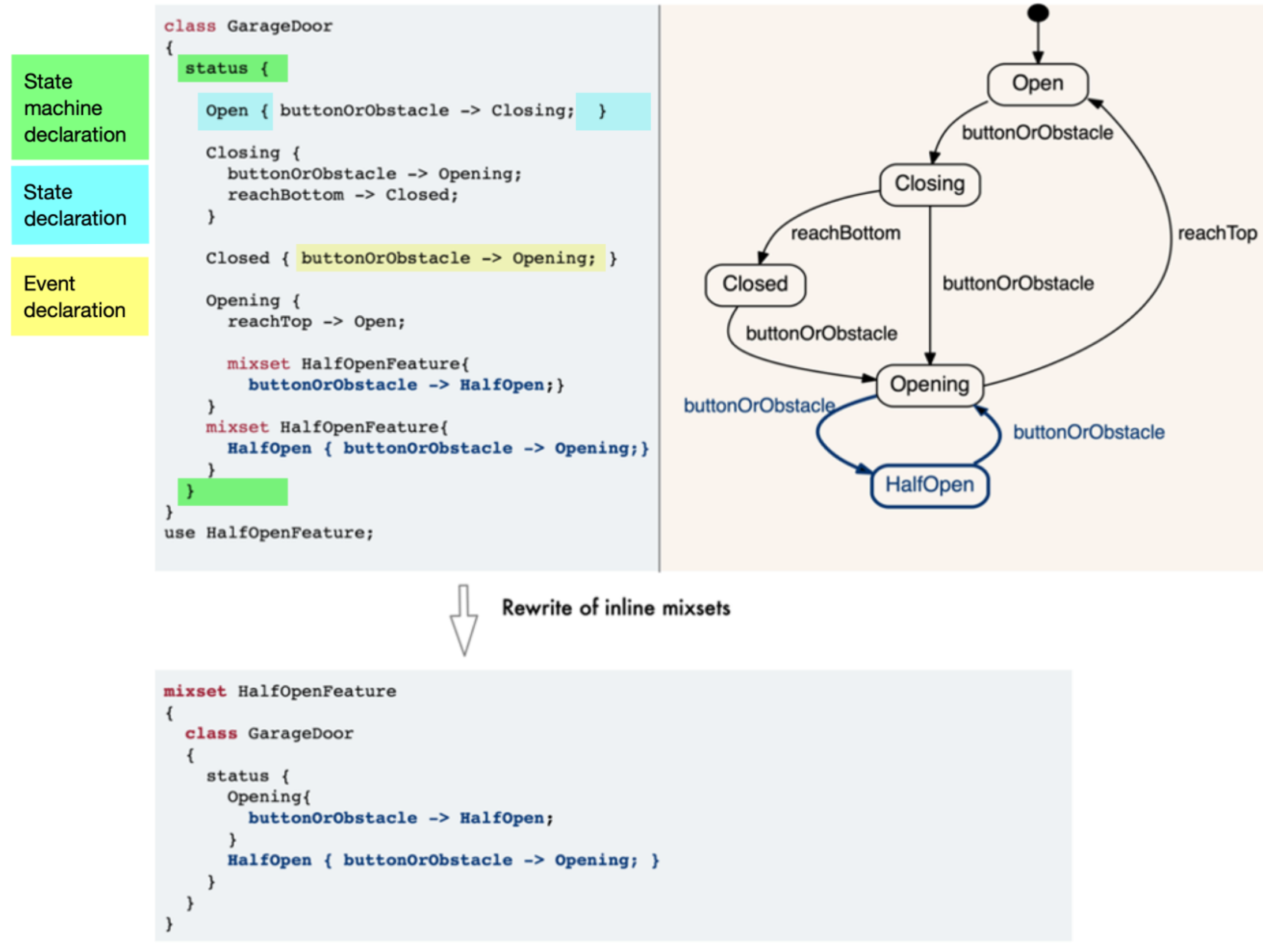

Figure 10: The inline mixset "HalfOpenFeature" is nested in two levels in the state machine model. The blue highlighted code belongs to HalfOpen feature in both the code and the diagram.

3.4 Refactoring Between Compositional and Annotative Mixsets

Since inline mixsets are transformed into compositional mixsets after parsing, it is possible to obtain a source code version in which inline mixsets are refactored into compositional mixsets. The opposite refactoring, which transforms compositional mixsets into inline mixsets, can be achieved through a generator that pushes compositional mixsets into their proper places in the abstract syntax tree. Figure 11 illustrates this duality. A typical use case for the composition-annotative refactoring would be a tool-based facility that shows a blending of all code segments for a feature selection when source code consists of compositional mixsets.

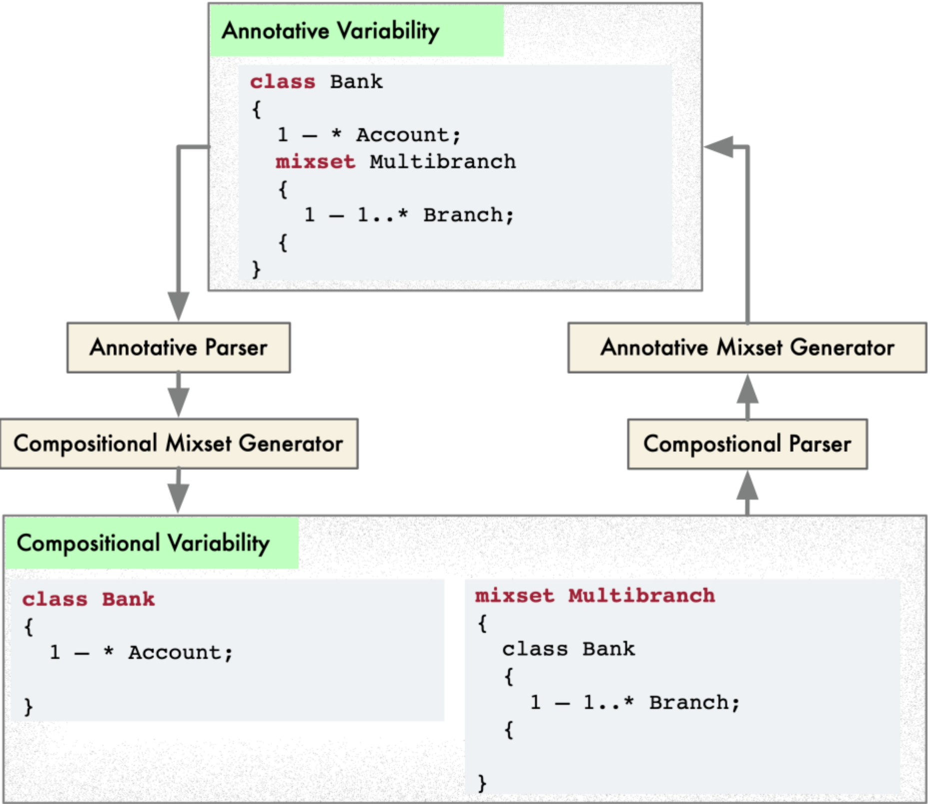

Figure 11: Two ways of refactoring of mixsets.

3.5 *Feature Modeling Using Mixsets*

Mixsets can be stand-alone entities that have their own attributes. In the context of software product lines, we extend the notion of mixsets with the *require* statements to model dependencies among mixsets. Hence a require statement contains a Boolean expression that specifies an exclusive or inclusive dependency among mixsets.

While mixsets offer conditional elements that help to implement features of software product lines, they can be used to assist with other aspects of software development. For example, mixsets can contain extra comments, documentation, or pieces of code that are used for special purposes such as debugging. To construct feature models out of mixsets, explicit specification is required to distinguish mixsets mapping to features.

In our approach, we have added the two keywords *isFeature* and *subfeature* to our language (along with the mixset and require keywords) to construct feature models. When a mixset acts as an optional feature; the keyword *isFeature* has to be included as an attribute in the mixset body.

A feature modeling segment, or a relationship can be specified using the *subfeature* keyword. This keyword forms a parent-child relationship between the source mixset and the target mixset and both will become features. The additional semantics that *subfeature* carries is the requirement to select the parent mixset when its child mixset is selected. In feature diagrams, parent-child relationships are transformed into a directed graph. To simplify the discussion, we will use the term *feature mixsets* for mixsets participating in feature modeling and regular mixsets or non-feature mixsets for technical mixsets.

The diagram of the feature model which was shown in Figure 3 is used as an example to illustrate how mixsets can be used to form a feature model. The Umple code for the actual feature model, constraining the possible configurations, is the following:

```
1   require subfeature [GSMProtocol opt Mp3Recording
2     and Playback and AudioFormat opt Camera];
3   mixset GSMProtocol {
4     require subfeature [GSM1800 opt GSM1900]; }
5   mixset AudioFormat {
6     require subfeature [1..2 of {Mp3,Wav}]; }
7   mixset Mp3Recording { require [Mp3]; }
8   mixset Camera { require subfeature [Resolution]; }
9   mixset Resolution{
10    require subfeature [1..1 of {Res21MP, Res31MP, Res50MP}];}
11  use GSMProtocol; use GSM1800; use Playback; use AudioFormat; use
12  Wav;
```

The first two lines represents a require statement with the keyword *subfeature*. This statement builds a feature model segment consisting of a parent feature (the current file as a base feature) and five children features. The feature diagram for this segment maps to Figure 12.

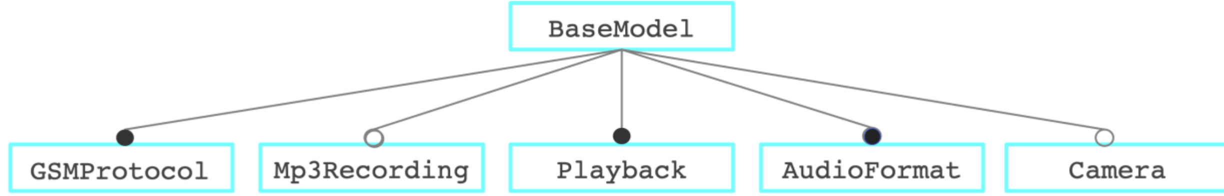

Figure 12: The corresponding feature diagram notation for the first require statement.

Line 7 has a require statement that specifies an inclusion relationship between the two features: Mp3Recording and Mp3. The final line of the code allows us to always meet the constraint associated with always-required features, to avoid having to specify these on the command line.

The above is as far as many feature modelling technologies go. However, mixsets easily allow modeling at a detailed level such that code can be generated. The following shows some details of one approach to structure the audio-related code for the above.

```
1   mixset AudioFormat {
2     class AudioCodec { /* other details omitted */}
3   }
4   mixset Mp3 class Mp3Codec {
5     isA AudioCodec; /* other details omitted */
6   }
7   mixset Wav class WavCodec {
8     isA AudioCodec {/* other details omitted */
9   }
```

Note that rather than using mixsets, files could have been used instead; the feature model (require statements) above would have been the same, except with '.ump' suffixes.

To build a system with some combination of features, the user could specify the following as the command line with its arguments, assuming Phone.ump is the top level file:

```
umple Phone.ump GSM1900 Wav Camera Res31MP
```

Alternatively a file called Config.ump could be created containing

```
1   /* A configuration file to activate the following
2      mixsets: GSM1900, Wav, Camera, and Res31MP.
3
4   */
5
6   use GSM1900, Wav, Camera, Res31MP;
```

and the command would then be:

```
umple Phone.ump Config.ump
```

Feature mixsets offer sufficient feature modeling capabilities for general specification of software product lines. Because feature mixsets can form fully fledged feature models, feature

modeling analysis and feature selection optimization can be implemented on feature mixsets. Implementing a large portion of feature modeling analysis methods mentioned in the literature (Benavides et al., 2010; Bhushan et al., 2020) would extend mixsets' usefulness for some domains, but we consider this matter out of the thesis scope.

3.6 Relationship and Nesting between Mixsets Types

Variability modeling tools helps to specify feature models at specification level. However, traceability between feature models and actual implementation often becomes difficult to handle as code evolves. Inconsistencies between variability at specification and its implementation is a major issue in SPL development (Jalote et al., 2014; D. M. Le et al., 2013; Tërnava and Collet, 2017). To solve this issue, there are several mining techniques proposed to extract evolving features from code (Jalote et al., 2014; Liao et al., 2018; Michelon et al., 2020).

Instead of extracting feature models from source code and then validating the extracted feature model against the one in the specification, mixsets seek explicit specification of the mapping and the relationships between features (represented as feature mixsets) and implementation-specific variability that is not represented in variability models (represented as non-feature mixsets).

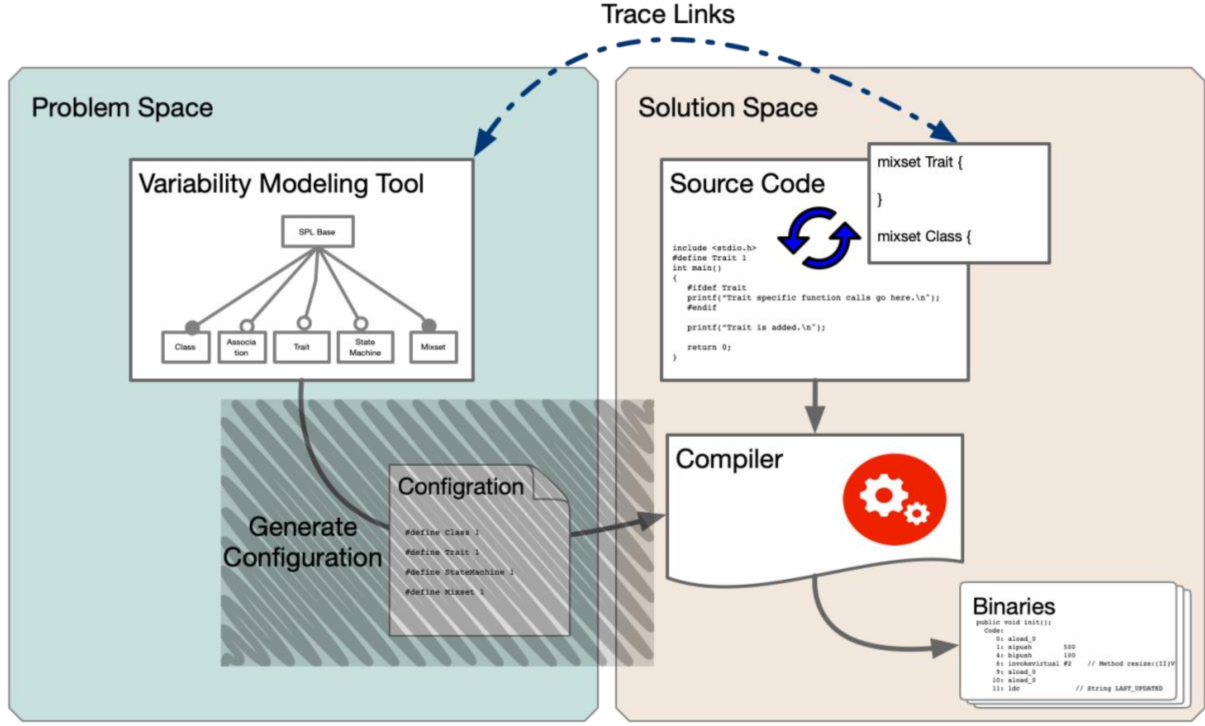

Figure 13: Feature models will be represented in the solution space as mixsets instead of being configuration parameters (dark scribbled rectangle).

Figure 13 illustrates the approach to maintain feature models through mixsets in the solution space. Duplication of feature models in the solution space avoids generation of configuration files, which contain parameters to produce a certain variant. Configuration files are often processed as input with other source code files with no means to detect inconsistencies between variability specified at specification and at implementation.

To further control the mapping between feature models and other conditional elements in the solution space, we propose three modes to control the inclusion of regular mixsets, feature mixsets, and their dependencies. The first mode is FeatureOnly mode, which allows only mixsets forming feature fragments. Hence, regular mixsets are not allowed in this case and each feature in the feature model will map to a corresponding feature mixset as in Figure 14.

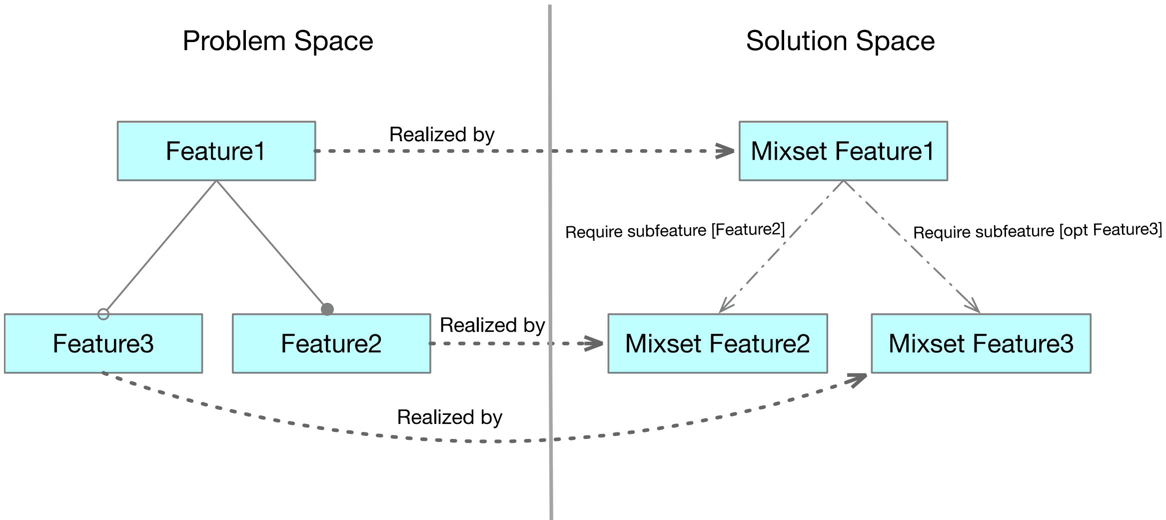

Figure 14: Direct mapping between of SPL features in the problem space and feature mixset (right).

Properties of FeatureOnly mode:

- Each feature maps to unique feature mixset.
- Dependencies among feature mixsets are modelled via *require* statements, which should conform to the feature model specification.
- Non-feature mixsets are not allowed in this mode.
- Feature mixsets should not activate (*use*) other feature mixsets via *use* statements. The instantiation of SPL variants should occur outside a feature mixset.

The second mode is Strict mode, which relaxes the FeatureOnly mode by allowing regular mixsets to be present in source code and to be used by feature mixsets without being part of the feature model. For instance, a group of non-feature mixsets may have their own dependencies separate from feature mixsets (which construct feature segments). Figure 15 illustrates the Strict mode.

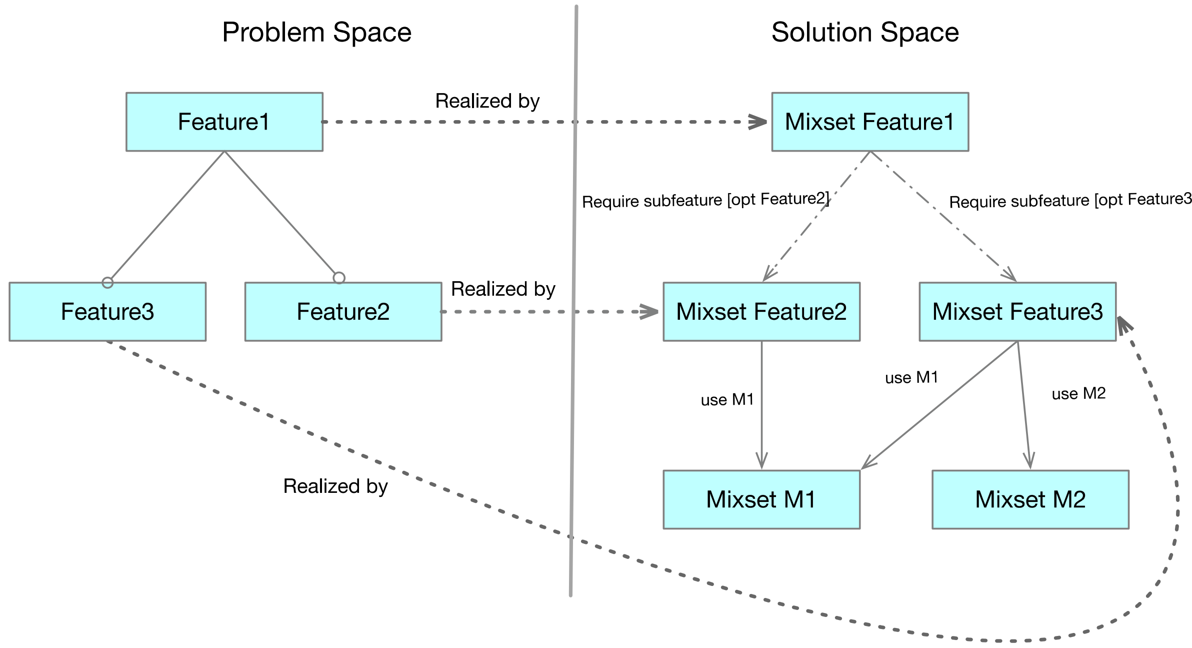

Figure 15: A Feature may map to multiple mixsets in the Strict mode.

Properties of Strict mode:

- A feature may map to more than one non-feature mixset.
- Dependencies among features in the feature model are modelled through *require* statements, which express dependencies among feature mixsets.
- A feature mixset can *require/use* non-feature mixsets. *Use* makes features included by default whereas *require* offers a set of mixset selections.
- If a feature mixset only *uses* a non-feature mixset, the mixset becomes required for that feature.
- A feature mixset is not allowed to *use* any feature mixsets.
- Non-feature mixsets is not allowed to *require*/*use* feature mixsets.
- Non feature mixsets can build their own dependencies but they are not allowed to cause conflicts with feature mixsets' dependencies.

The third mode is Default mode, which has no restriction on mixset types and dependencies. This mode is flexible but has the potential to contain all types of inconsistencies when used to model features for an SPL. Figure 16 depicts these three modes in a notation which is expressed in the white box of the first mode.

Embedded variability is another factor that complicates feature mapping and their relationship in implementation (D. M. Le et al., 2013). Since variable units may embed other variable units in uncontrolled manner, mixset modes govern the embedding of mixsets. Table 4 below summarizes the cases in which the two types of mixsets may nest each other.

Table 4: Nesting mixsets.

Mixset Nesting	Both (M1 and M2) are Feature Mixsets	Both (M1 and M2) are Non-Feature Mixsets	A Feature Mixset and A Non-Feature Mixset
Mixset M1 is totally separate from mixset M2. (Diagram: Mixset M1, Mixset M2)	Always allowed. This is an ideal case for feature mixsets. They are ofren separate.	Always allowed.	Allowed if feature mode permits both feature mixsets and non-feature mixsets.
Mixset M1 completely embeds mixset M2. (Diagram: Mixset M1, Mixset M2)	Allowed only when feature mixset M1 is a parent feature of feature mixset M2.	Always allowed. In this case, M2 implicitly requires M1.	Non-feature mixsets should not totally embed feature mixsets. The opposite is allowed if the feature mode permits.
Mixset M1 partially embeds mixset M2. (Diagram: Mixset M1, Mixset M2)	Always allowed. This type of nesting could be used to manage feature interaction code.	Always allowed.	Allowed if the feature mode permits.

Mixset modes help to manage *imperfect modular variability* which describes the mismatch between the nature of SPL features as modular concepts in specifications while their implementation maps to sets of variation points and variants (Tërnava 2017). The FeatureOnly model allows 1-to-1 mapping between features in specification and variable units in implementation. The Strict and default modes permit n-to-m mapping but in a manageable way. Therefore, it is possible for multiple features to reuse some shared mixsets. In addition, a feature mixset can be restricted to choose from a set of regular mixsets (with dependencies and constraints

among them) while these mixsets are implemented in diverse ways such as inheritance, generic types, overriding and design pattern.

Mixset modes control implementation variability scope and explicitly distinguish between variability attributed to SPL features (occurs in specification) and variability emerges in implementation due to component dependencies or development purpose. A practical use of such modes can be allowing the default mixset mode during the SPL development process while ensuring accurate feature mapping with the Strict mode and the FeatureOnly mode before software delivery. Another use case for the default mode is explicitly knowing features that are bound to technical constraints such as features that depend on certain package versions or external systems. In mobile application development, there are several reasons for internal variability such as OS version, screen size and package version dependencies. This kind of variability is not essential to SPL features, but has great impact on the software development and has to be separate from SPL variability.

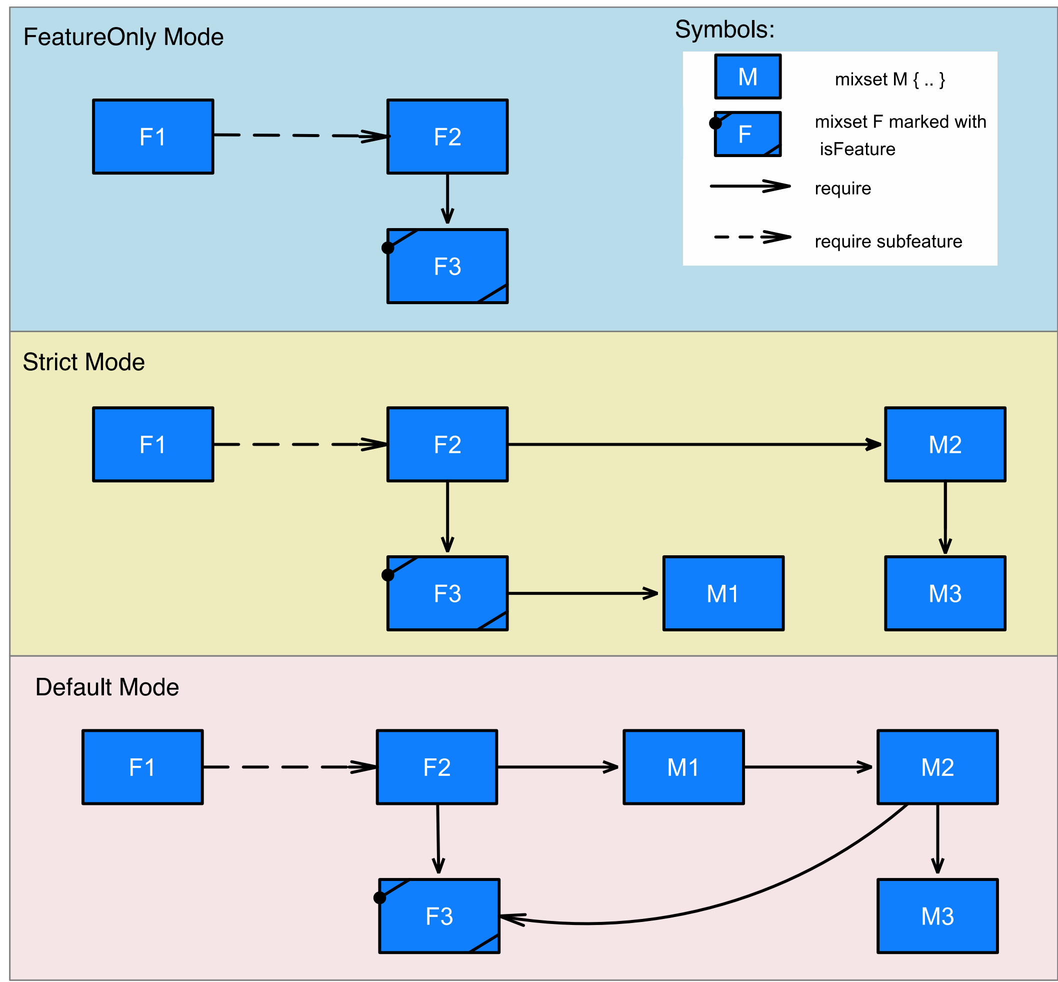

Figure 16: Three modes to include mixsets in source code. Flexibility increases downward.

3.7 Mixset Metamodel

A generic metamodel of mixsets is shown in Figure 17. The body of CompositionalMixset can contain fragments of the language's top entities to a certain granularity. The level of granularity is determined by the mixin composer. For example, a mixin composer may not allow expression-level granularity such as introducing parameters to methods. To specify which concepts should allow inline mixsets, each concept of the host language is required to have an association with the InlineMixset concept.

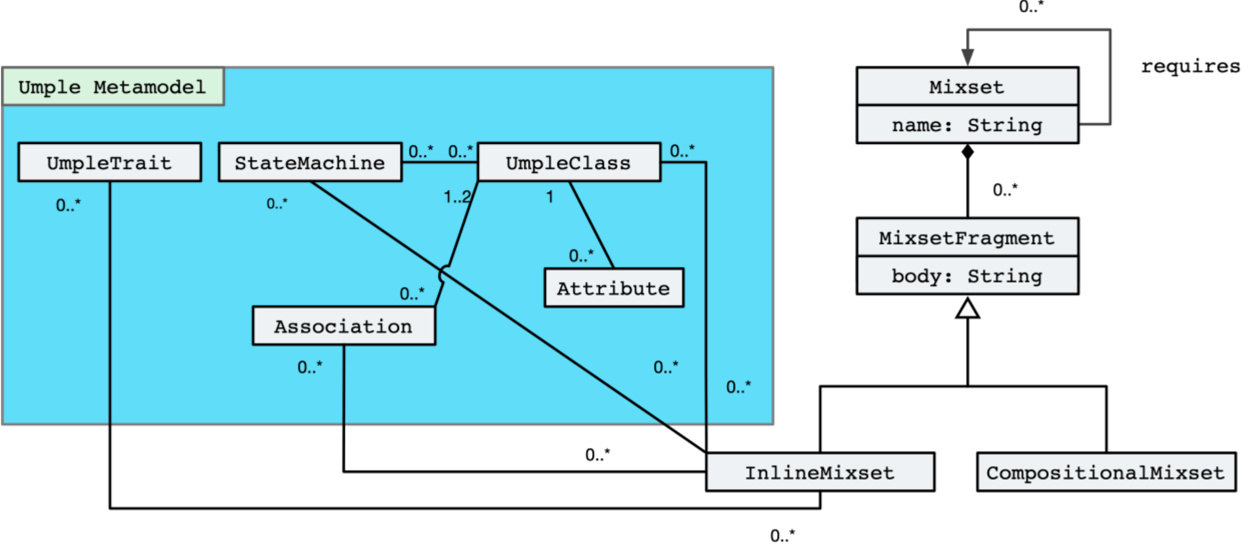

Figure 17: Mixsets metamodel augmented with a simplified fragment of the original Umple metamodel (highlighted in the blue box).

3.8 *Syntax Checking for Mixsets*

SPL variability brings challenges for any language's static analysis such as syntax checking; this applies equally to modeling or programming languages. Code annotation and code composition are two sides of the same coin when it comes to SPL type checking although compositional approaches have the advantage of being able to apply syntax checking for feature code in isolation (Kästner et al., 2012). For instance, parsing all elements at once without considering SPL variability in the following Umple code (stated in line 12) will result in a conflicting association error:

```
1   mixset IPhone8{
2     class IPhone {
3       1 -- 1 BackCamera; // has a single 12MP camera
4   }
5
6   mixset IPhone11{
7     class IPhone {
8       1 -- 2 BackCamera; // has dual 12MP camera
9   }
10  class Iphone{ }
11  class BackCamera{ }
12  require [Iphone11 xor Iphone8];
```

The SPL syntactic checking has to be addressed natively via a variability-aware parser; the lexer tokenizes variable fragments and their *presense conditions* (optional, mandatory, or alternative) as *conditional tokens* (lexemes) to the abstract syntax tree (Kästner et al., 2011). The major drawback of such variability-aware parsers is the massive change to stable and widely used compilers. In addition, variability-aware parsers may have scalability issues for large SPLs (Braz et al., 2016). The alternative for handling variability in parsers is to hide it from parsers. Mixsets follow the alternative scenario and introduce SPL primarily to languages which lack this capability.

Apart from variability-aware parsing, there are three ways to parse bodies of mixsets. First, mixsets' bodies can be considered as plain text that is ignored while parsing unless these mixsets are used. This option eases mixsets' integration into the host language. Hence, compositional and inline mixsets will contain only plain strings while inline mixsets can be applied to some elements of the host language. The negative implication of this option is the blindness to detect syntactic errors inside mixsets that are not used.

The second option elevates the syntactic checking by checking local bodies of unused mixsets (whether they are inline or compositional mixsets) in isolation of other mixsets and the rest of the language elements. For example, an inline mixset that is inserted inside a class may allow attributes, associations, methods and elements that a class permits. This is a middle-ground alternative which improves the syntactic correctness while requiring more integration overhead at the metamodel level.

Third, formal methods can be applied to achieve a higher degree of SPL syntactic checking. At the modeling level, Czarnecki and Pietroszek's UML SPL is checked according to well-formedness constraints, which are expressed in the Object-Constraint Language (OCL) (Warmer and Kleppe, 2003), such as absence of dangling associations (Czarnecki and Pietroszek, 2006). At the code level, formal methods can be used to rigorously prove type safety. For example, Colored Featherweight Java (CFJ) ensures type checking formally for an annotative SPL in Java source code (Kästner et al., 2012). CFJ bases its formalism on Featherweight Java (FJ), which is a compact and minimal calculus to rigorously prove type safety in Java programs (Igarashi et al., 2001).

MDD technologies that allow language-specific code to be embedded within models such as Umple add further challenges for SPL type checking. Figure 18 shows Java code generation from a state machine model named GarageDoor in Umple. GarageDoor's model is represented using state machines, and it incorporates pure Java code.

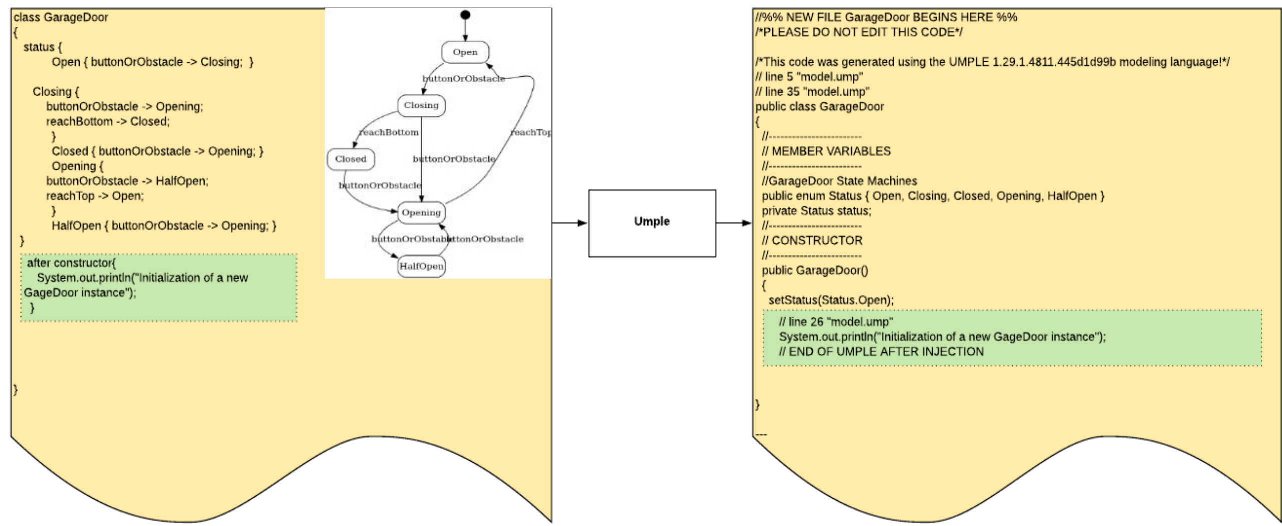

Figure 18: GarageDoor state machine model on left. Green shaded area contains pure Java code added to the model.

A desirable approach is to build SPL-aware compilers that have a full support for SPL compilation with SPL type checking for both models and embedded code. The possible solutions for checking syntax and type checking for a model driven SPL containing embedded code are:

- Entirely transform the source code including feature models and feature code into the host programming language. This would transfer SPL variability management into the host programming language to process feature models, verify syntax and type systems, and produce variants. The drawback of this option is maintaining a set of tools, error traceability, and its unfeasibility for the case of multiple embedded code.
- Level up embedded code to the model level. This would require manual, or novel automated, reasoning to understand the relationships between the embedded code and the models.
- Partial checking for syntax and type system of models while embedded pure code will be unchecked. The major issue with this option is the fact that the embedded code would creep into the generated code and may cause errors that are not detected.

The solutions above are not exclusive; a MDD tool can employ an option and then gradually move to more rigorous solution. From practical perspective and for quick adoption of SPL, the third option is more reasonable.

3.9 Summary

This chapter demonstrates two key features of the mixset approach. First, it shows how a superimposition composer can be used to unify the underlining representation for inline and compositional mixsets via the language's AST. Hence, mixsets offer a generic approach to offer annotative, compositional, or combined variability. Second, we discussed the mechanism of mixset modes to control the mapping between feature models and other conditional elements in the solution space through explicit specification of regular mixsets, feature mixsets, and their dependencies. We proposed three mixset modes (FeatureOnly, Strict, and Default mode) and we expressed their characteristics in terms of relationships (use and require statement) and nesting. The end of this chapter introduced the metamodel of mixsets and expressed a few issues related to the syntax checking of mixsets. Of particular importance, we draw the attention to the complexity to apply syntactic checking for code embedded with models when MDD embraces SPL.

Chapter 4 Umple Extensions to Incorporate Mixsets

In this chapter, we introduce the enhancements we have made to Umple to implement the approach of mixset. Specifically, we justify implementing mixsets in Umple. Next, we describe the mixin composer as a superimposition technique to merge matching entities. Then, the definition and use of mixsets to specify fragments of Umple are described including their grammar, the algorithms for processing them, and the constraints that can be imposed on them. The addition we made in this chapter is integrated into the master repository of Umple (Lethbridge, Abdulaziz, et al., 2021).

4.1 Why Umple?

The previous chapters discussed the mixset concept itself as independent of Umple. The reasons why it was implemented in Umple in this thesis are:

a) The CRuiSE lab at the University of Ottawa is developing Umple, so it made sense to extend Umple, in which we have deep expertise, rather than extending unfamiliar technology.

b) We wanted to demonstrate and explore the synergies that might arise from being able to use several different separation-of-concerns mechanisms together (traits, mixins, aspects) along with both models and traditional code; we were not aware of any tool other than Umple that has this sort of capability.

c) Umple is open-source software, available since 2008. The core functionality of Umple has emerged from academic research. Furthermore, large numbers of students (mostly undergraduate) across the world use Umple for tasks related to software modeling. We count on this exposure for enhancement, constructive criticism and integration with other tools.

d) When mixsets are enabled, it becomes possible to use Umple as a case study for a typical software product line. Since Umple is written in itself, mixsets can be used to transform Umple into an SPL.

e) Umple, in addition to being a command line application and an Eclipse plugin, is also an online tool that is accessible and easy to explore. Documentation and examples are well-integrated with the design and the implementation in Umple. For example, the grammar documentation for each capability is extracted from its actual grammar file. This helps encourage use, which will enable us to potentially find people willing to participate in empirical studies.

One could implement mixsets in a programming language or a different modeling language. Generically, mixsets can be added to a language by:

- First providing a mixin capability to allow language elements to be created in parts that are separately defined.
- Creating a syntax that encapsulates named code or model fragments, where the name can be repeated, and each repetition of the same name designates a member of the mixset.
- Creating a syntax to use (i.e. activate, or include, or import) a mixset, enabling it to be parsed and its members hence *mixed in* to the system. In this thesis, we use the term ‘use’, since that is the keyword adopted by Umple.
- Creating a parsing mechanism that accounts for various special cases such as whether a mixset is a) defined before an instruction to use it is encountered, b) defined after such an instruction, or c) partially defined before such an instruction and partially afterwards.
- Creating a mechanism to describe logical constraints among mixsets and analyze them to verify that their compilations are valid.

4.2 Mixins: The Foundation for Mixsets

Key top-level entities in any Umple file prior to the work reported in this thesis were defined using the syntax:

```
class name {…}

interface name {…}

trait name {…}.
```

The three dots (…) represent internal contents such as attributes, associations, state machines and methods; but we do not need to consider the details of these.

Any repeated occurrence of a top-level entity will add content to the entity (i.e. *mixes* it in). Thus:

```
class A {a;}

class A {b;}
```

results in:

```
class A {a; b;}.
```

The redefinitions can be in the same file, or in separate files. When in separate files, the possibility of selective inclusion of mixins has always been available in Umple, hence allowing production of different system variants.

Umple traits (Abdelzad and Lethbridge, 2017) have additional semantics: they can be incorporated wholly or in part in other traits and classes with potential renaming of internal elements, and enforcement of the presence of various required (dependent) elements. Traits can be built from various parts using mixins, just like classes and interfaces can. Figure 19 shows how the Sensor class parametrizes the Dashboard class for the Subject observer trait.

```
class Dashboard{
  void update (Sensor sensor){ /* code */ }
}
class Sensor{
  isA Subject< Observer = Dashboard >;
}
trait Subject <Observer>{
  0..1  -> * Observer;
  void notifyObservers() { /* code */ }
}
```


Figure 19: A trait implementing an observer design pattern.

4.3 Mixset Basics

Mixsets, the core concept of this thesis, provide variability modeling for both structural and behavioral models in Umple. Mixsets are a new type of top-level entity in Umple. Umple's other top-level entities include classes, interfaces, traits, standalone state machines, use statements, and enums.

The basic syntax for introducing a mixset is:

```
mixset mixsetName {…}
```

A compositional mixset contains a set of Umple top-level entities. But since top level entities can be specified in pieces, or mixins, that are composed to form the final entity, each mixset statement need only describe one of these mixins. A mixset is therefore a *set of mixins*.

Members of the set can be fragments of any of Umple's valid top-level entities including other mixsets. Mixsets allow these members and their contents to be combined to build a potentially optional feature or part of a system. Different combinations of mixsets would give rise to different variants perhaps targeted for a particular customer group or hardware platform.

Mixsets can enable structural model variability by adding new classes, attributes or associations to a model. They can provide behavioral model variability by adding alternative implementations of methods, or alternative states, transitions, and actions in state machines.

The contents declared in the curly brackets of a mixset will only ever be incorporated in the variant if the Umple compiler encounters the syntax:

```
use mixsetName;
```

This is the same as the syntax used for incorporating separate Umple files, so a filename ending in .ump is actually a special case of a mixset name. Naming a mixset as a command line argument has the same effect as specifying a use statement, in fact, internally command line arguments are converted to such statements.

Since mixins are top-level entities, subject to being composed using Umple's mixin capability, parts of a given mixset can be found in different files or other mixsets.

4.4 Basic Mixset Cases

The following are the four simplest cases for the definition and use of mixsets to specify fragments of Umple classes. Wherever a class definition is shown in these cases, several classes could be given, and elements such as interfaces and traits can also appear. Specifying fragments of other entities such as interfaces and traits works the same way.

Case 1: Basic mixset definition and inclusion.

```
mixset M1 {class A {a;}}
```

The above means include the class A with string attribute "a" only if somewhere else the mixset use statement below appears or if M1 appears in a command argument to compile the system:

```
use M1;
```

From now on, by *including* a mixset we mean by either via a use statement or a command argument. For syntactic simplicity, if only a single entity is contained in the mixset definition, the outer braces can be omitted, so the above can be simplified to:

```
mixset M1 class A {a;}
```

Case 2: A class defined partly in a mixset.

```
mixset M2 class A{a1;}

class A {a2;}
```

The above results in class A always being present with attribute a2, but the class A will also have attribute a1 *only* if M2 is included.

Case 3: Parts of a mixset defined separately.

```
mixset M3 {class A{}}
mixset M3 {class B{}}
```

The above has the same effect as the following:

```
mixset M3 {class A{} class B{}}
```

In other words, the two declarations of M3 are mixed together using Umple's standard mixin approach.

Case 4: Mixset definition containing elements internal to classes, traits and interfaces. Such elements (e.g. methods, attributes) can be wrapped in mixset notation, as can blocks of code, wherever code is allowed in Umple (e.g. state machine actions, and so on).

```
class X { a; mixset M4 b;}
```

The above syntax is equivalent to:

```
class X {a;}
mixset M4 class X{b;}
```

This will result in class X having attribute b only if the following use statement is encountered:

```
use M4;
```

Case 4 items will be transformed to Case 3 as described in Chapter 3 .

4.5 Grammar for Mixsets

The following is an extract of the Umple grammar rules describing mixsets. This is taken directly from the Umple user manual grammar page (*Umple Grammar*, 2021), which is generated in 'pretty-printed' form from Umple's source code (as written by the author of this thesis). In the Umple EBNF grammar notation, terminal constant symbols are shown in red; placeholders for alphanumeric identifiers are show in green with single square brackets; nonterminal rule names are in blue, and references to rules are in double square brackets. Three dots means that a rule that existed prior to the work of this thesis has been modified to add the item, with irrelevant details represented by the three dots left out.

Rule 1 indicates that require and mixset statements as well as specializing feature mixsets are newly introduced as top-level entities. Rule 2 shows that they can also be added within classes (similar rules showing they can be added to interfaces, states, and state machines are omitted for brevity).

Rules 4 and 5 show how the 'extra code' comprising each mixset fragment is specified. Lines 6 through 12 define the require statement grammar. Note that the reference to multiplicity in Rule 6 refers to an existing Umple rule. The content of mixsetInnerContent (Rule 4) may include Rule 13, which indicates a feature mixset.

```
1    entity- : …  [[requireStatement]] | [[mixsetDefinition]] | [[mixsetIsFeature]] …
2    classContent- : …  | [[mixsetDefinition]] …
3    mixsetDefinition : mixset [mixsetName] ( [[mixsetInnerContent]] | [[mixsetInlineDefinition]] )
4    mixsetInnerContent- : { [[extraCode]] }
5    mixsetInlineDefinition- : ( [entityType] [entityName] ( [[mixsetInnerContent]] )
6    requireStatement : require ( [=subfeature] )? ( [[requireBody]]
       | ( [[multiplicity]] of { [[requireTerminal]] [[requireMultiplicityList]] } ) )
7    requireBody- : [(([[requireLinkingOptNot]])? [[requireTerminal]] [[requireList]])]
8    requireList- : ([[requireLinkingOp]] [[requireTerminal]])*
9    requireMultiplicityList- : (, [[requireTerminal]])*
10   requireLinkingOp : ([[requireLinkingOptNot]] | [=and:&|&&|and|,] | [=or:(| | || |or|;)] |
     [=xor:xor|XOR] )
11   requireLinkingOptNot : (opt | not)
12   requireTerminal : [targetMixsetName]
13   mixsetIsFeature: isFeature
```

4.6 General Algorithm for Processing Mixsets

To understand how mixsets work, it is helpful to understand how they are processed when Umple parses its source text.

Upon encountering a 'use' statement, the *earlier* version of Umple prior to this work would assume the identifier following the 'use' keyword was a file name, and would attempt to load and parse the file, with an error being raised if the file were not found. There would be no effect of repeated use statements referring to the same file. Specifying command-line arguments containing file names had the same effect as specifying multiple use statements.

With the introduction of mixsets, any identifier in a use statement or command line argument that does not end in '.ump' is assumed to be a mixset, and not a file.

Essentially, mixsets are treated as dynamically-generated virtual files – the Umple compiler only holds them in memory, rather than reading them from the file system. The compiler adds the contents between the mixset definition's braces (which we refer to as a fragment) to the relevant virtual file, without parsing it (other than to balance braces, quotes and comments), but also recording its location in the original .ump file to allow for debugging and to help solve the delocalization problem (i.e. to enable tools to help the user search for the fragments of a mixset, or the various mixins contributing to another entity such as a class).

Use statements work essentially the same with mixset virtual files as with real files. Virtual files containing mixset contents are processed when a matching use statement is encountered as though the file was real.

The pseudocode below (Algorithms A-D) describes the process that occurs when mixset and use statements are encountered. The nuances of this process that differ from processing of real files are:

1. If the Umple compiler encounters a use statement of a mixset *before* encountering a matching mixset declaration, Umple creates an empty virtual file for that mixset, by recording that a reference to the mixset name has been encountered (Lines C6-C7).
2. There is the possibility of (additional) fragments of a mixset being created *after* a matching use statement. In that case, a (new) fragment is added to the mixset virtual file and is processed immediately (Line A5 below). In other words, upon encountering such a mixset fragment, Umple first looks to see whether a use statement for it has already been encountered; if so, it processes that fragment immediately.
3. When mixset content is found *inside* a top-level entity, it is rewritten to the Case 3 form, as shown in Case 4 above, before being processed (Algorithm B below).
4. Detection and reporting of a *missing* mixset (used without a matching definition, Lines D4-5 below) or other constraint violations (associated with require statements, Lines D8-9 below) can only occur at the end of parsing.
5. Reporting of errors and warnings refers to the original location in the .ump file, rather than to the location in the virtual file (Line A7 below).

```
A1   Algorithm A handleBasicMixsetStatement(
             mixsetName, codeBlock)
A2     -- Invoked when Umple parser encounters
A3     --   mixset mixsetName {codeBlock}
A4     If (use statement for mixsetName has been encountered)
A5        Parse codeBlock in the same way as a new Umple file
A6     Else
A7       Record with codeBlock the line in the .ump file
              where the mixset statement was found
A8       Add codeBlock to the ordered set of fragments
              associated with mixsetName
A9     End if
A10  End

B1   Algorithm B handleMixsetInsideEntity(
                 mixsetName, codeBlock, entity, entityName)
B2     -- Invoked when Umple parser encounters
B3     -- entity entityName { … mixset mixsetName {codeBlock}}
B4     Add 'entity  entityName {' to the start of codeBlock
B5     Add '}' to end of codeBlock
B6     Process modified codeBlock
           using handleBasicMixsetStatement
B7   End

C1   Algorithm C handleUseStatement(mixsetName)
C2     -- Invoked when Umple parser encounters
C3     -- use mixsetName;
C4     -- and also if mixsetName is a command line argument
C5     If(a use statement for mixsetName
               has NOT already been encountered)
C6        Record that a use statement was encountered
              for mixsetName
C7        If (no fragments are associated with mixsetName)
C8            Do nothing
C9        Else
C10         Parse any fragments associated with mixsetName
               in the same way as an Umple file
C11       End if
C12    End if
C13  End

D1   Algorithm D endOfParsingHandling()
D2     -- Invoked after all parsing is complete
D3     For each (mixset referred to in a use statement)
D4        If (no fragment has been encountered for it)
D5          Emit a warning
D6        End if
D7     End for
D8     If(the logical condition of any require statement
                        evaluates to false)
D9        Emit a warning
D10    End if
D11  End
```

4.7 Nesting of Mixsets

Mixsets can be nested as follows:

```
mixset P {
  class M{};
  mixset Q {
    class N{}
  }
}
```

If P is included only, then class M will exist. If Q is included only, then neither class N or M will be created by this code because the system only 'sees' Q if P is included. If both P and Q are included, then both classes M and N will appear.

Combinations among mixsets can be specified in the following ways, where M1 … Mn are mixsets:

```
use M5, M6;
```

means include both M5 and M6. If either doesn't exist it is an error.

```
use M5; use M6;
```

is same as the above, just in two separate statements.

```
mixset M7 {use M8, M9;}
```

means that if M7 is incorporated, then incorporate M8 and M9 as well. In other words, M7 requires both M8 and M9. This is just a simple combination of a regular mixset statement and regular use statements.

4.8 Dependencies among Mixsets

The *require* statement constrains a feature of a product family, or a dependency between non-feature mixsets. The following gives an example of or dependency:

```
require [M10 or M11];
```

This means that somewhere there *must* be a use statement for M10 or M11, or both. Standard Boolean logic is allowed with the keyword `xor` meaning at most one, `and` and `not` having the usual meanings, and parentheses being used for grouping.

The `of` operator is also available in the require statement. This can be used to require a certain number of mixsets selected from a set specified by a multiplicity, such as 1..* (meaning one or more). An example would be:

```
require [1..2 of {M12, M13, … }];
```

meaning that only one or two of the listed mixsets is allowed, The notation

```
require [opt M14];
```

embedded in an expression is syntactic sugar for

```
require [0..1 of {M14}];
```

in other words marking a mixset as optional.

Compound constraints follows Boolean-like operator precedence if braces are not explicitly stated in the require statement. The precedence order sequences from (higher to lower): not', 'and', 'or', 'opt', then 'xor'. The following statement:

```
require [M14 and M15 not M16 xor M17 opt M18 and M19];
```

would translate into the conjunctive clause, where AND is capitalized for clarity:

```
require [(M14 and M15) AND ((not M16) xor M17) AND (opt (M18
and M19))];
```

Note that 'not', 'opt', and the multiplicity operator are unary operators whereas 'and', 'or', and 'xor' are binary. The multiplicity operator has the priority of "xor " operator precedence.

Taken together the above allows a rich way of embedding feature models within arbitrary Umple code. A feature model can be described all in one place in a single file, or the dependencies and inclusions can be distributed among multiple files. This flexibility is one of the hallmarks of Umple. However, Umple also prevents delocalization confusion by allowing automatic drawing of standard feature model diagrams.

4.9 *Feature Diagrams*

We enhanced Umple with the capability to generate feature diagrams out of feature mixsets in UmpleOnline. Figure 20 shows a feature diagram which is generated from the MobileSPL in UmpleOnline. Also, mixset dependencies can be shown in UmpleOnline.

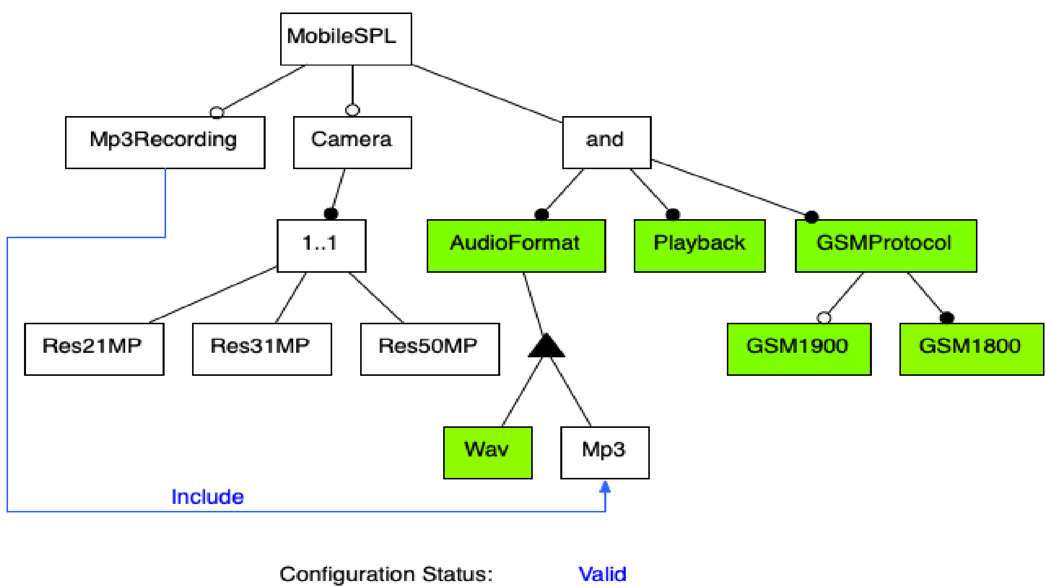

Figure 20: The feature diagram generated from the mobile SPL code mentioned in Section 3.5 .

When feature diagrams become large and dependencies among mixsets get complicated, we dedicate a mixset-dependencies diagram to visualize their relationships. Figure 21 shows an example of feature dependencies of the Berkeley BD JE, which will be discussed in Chapter 6 .

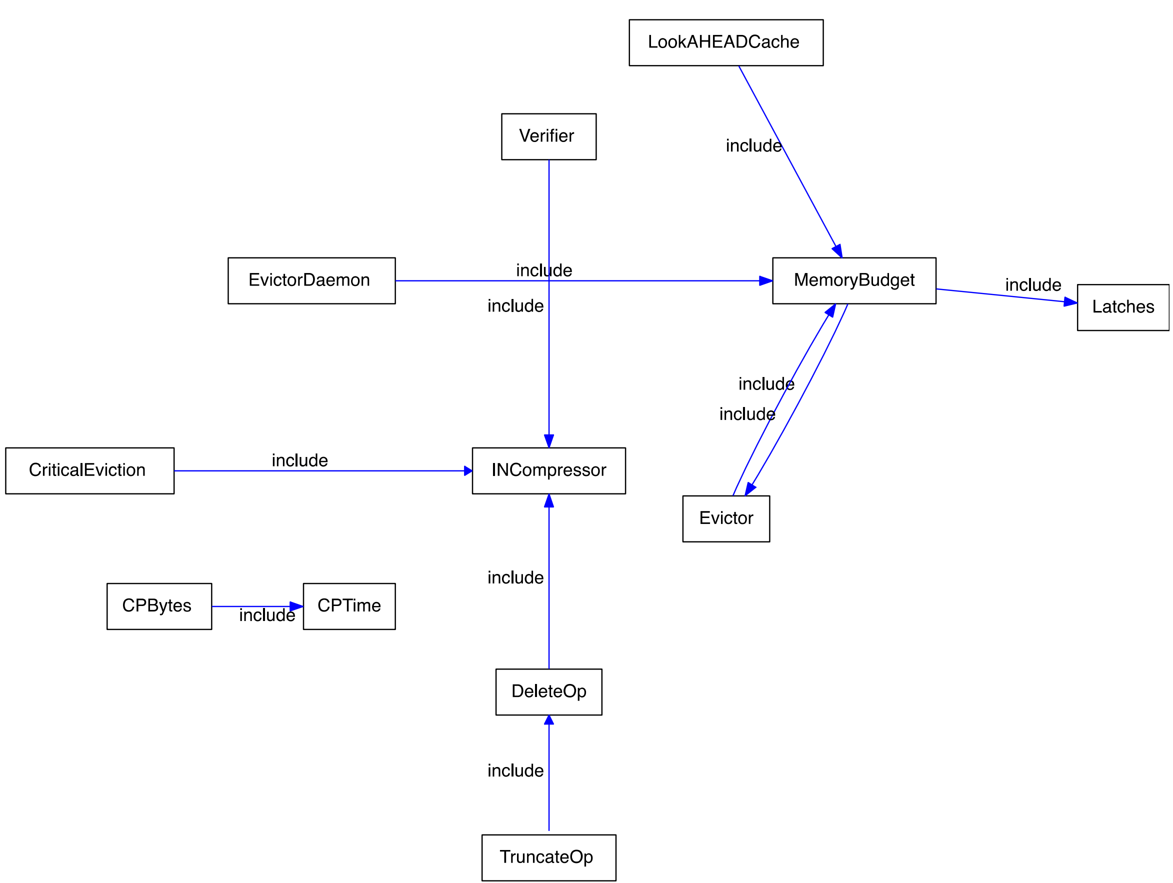

Figure 21: Dependencies among mixsets is shown in a separate diagram using UmpleOnline.

Chapter 5 Mixsets for Fine-Grained Feature-Oriented Software Product Lines

An obstacle to the practicality of existing SPL composition techniques is the coarse-grained granularity in that an SPL base is merged to feature increments limited to adding new classes, or new fields and methods to existing classes with little help to modify code inside methods. In this chapter, we present a solution that allows SPL composition to be performed at the statement level, which is considered *fine-grained* granularity (Kästner, Apel, and Kuhlemann, 2008). To achieve this, we extend the concept of mixsets to allow compositional mixsets to inject their code at specific positions *inside* methods. We use code labels as marking mechanism (variation points) to identify locations within the code of methods. Code labeling has long been supported by important programming languages.

The background in the next section discusses SPL composition as a technique enabling feature-oriented SPLs. Then, we point to the obstacles that prevent compositional SPL approaches from applying their statement level mechanisms when handling fine-grained variability. After that, the mechanisms to overcome fine-grained variability for compositional SPL are discussed. In Section 5.4 we motivate and present our work. Finally, Section 5.5 reviews limitations of our approach.

We will discuss the application of our method to the Oracle Berkeley Database in Chapter 6 . The study shows promising improvement in code quality as compared to alternative compositional approaches; in particular, it allows elimination of many complex hook methods.

5.1 Background

Code composition techniques are based on separation of concerns. In feature-oriented software development (FOSD), an SPL is assembled from a set of features and the code of each feature is placed in physically distinct feature modules (Apel et al., 2013). Hence, a feature consists of code units that are separated from the base code and often stored in a separate directory. To obtain a product, or a configuration, in an SPL, the base code and the code of the selected features are merged according to a specific composition technique. The composition can take place at

compile time, deployment time or runtime. Although SPL composition offers high feature traceability, it faces the challenges of fine-grained variability and concern delocalization.

Feature traceability is an intrinsic property of composition-based approaches. The term traceability refers to, "the link describing a relationship or dependency between two artifacts developed during the various phases of software engineering" (Berg et al., 2005). There is a clear mapping between features and their code units in FOSD, hence, feature traceability is explicit. This mapping does not imply one-to-one relationship; it might be n-m whereby a unit of code that belongs to one feature can be reused by other features. Detaching SPL features in separate places lifts the burden of updating variation traceability information as the system evolves. The traceability has an important role in SPL development and influences change impact analysis and software maintenance.

The granularity of variable code required to implement SPL features ranges from file inclusion and progresses down to code fragments at the statement level till expression and token levels. The addition of statements in specific positions inside methods or extending a list of parameters of a method signature is considered fine-grained granularity. Inclusion of code fragments such as new classes, fields, and methods are considered coarse-grained granularity and are readily supported by composition-based techniques. Prior to the research, code composition techniques for SPLs do not inject new lines of code at arbitrary places inside methods (Kästner and Apel, 2008a). The limitation of composition-based approaches to coarse-grained changes hinders their practical use (Apel et al., 2013; Kästner, Apel, and Kuhlemann, 2008; Kruger et al., 2018).

Another challenge for compositional approaches is the delocalization of features as they constitute isolated modules; these modules may lead to local and partial understanding of features' effect on the whole system (Letovsky and Soloway, 1986). Tool support is often employed to overcome feature delocalization (Kästner, Apel, Trujillo, et al., 2008).

5.1.1 Compositional SPL Techniques

This subsection reviews main techniques to realize SPL features through composition: aspect-oriented programming (AOP), feature oriented programming (FOP), aspectual feature modules (AFM), and delta-oriented programing (DOP).

Aspect-Oriented Programming (AOP)

Aspect-oriented programming (AOP) introduced aspects to reuse common code that crosscuts multiple classes and cannot be captured by traditional programming methodologies (Kiczales et al., 1997). A *concern* is a broad term involves all conceptual units that have impact on design and maintenance of program's components (Robillard and Murphy, 2007). An *aspect* is a reusable piece of code which mainly consists of *advices* and *pointcuts*. An *advice* describes a crosscutting concern that should be injected at specific join points specified by a pattern called a *pointcut*. In addition to pointcuts and advices, *inter-type declarations* can introduce fields, methods, and constructors to classes as well as implementing interfaces and extending classes. An *aspect weaver* parses aspects to identify the location of injection in original modules and then combines advice to corresponding modules. AOP works well to encapsulate homogeneous crosscuttings containing blocks of code are tangled and replicated over different locations (Apel et al., 2013). Security and logging requirements are typical cross-cutting concerns that illustrate AOP elegance (Sullivan et al., 2005).

There are two important characteristics of AOP languages: quantification and obliviousness (Filman and Friedman, 2000). The quantification characteristic describes the ability to inject a single advice to multiple locations in a program by using wildcards. Obliviousness describes the state of the (base) source code as not being able to directly reference aspects. Hence, source code has no knowledge of aspects' presence.

Although AOP is a generic reuse methodology that is supported by some programming languages through external libraries, AOP has gained more widespread within the Java realm. For example, the Spring framework, a popular Java web framework, offers native support for AOP (Gutierrez, 2014). The Spring framework has a subset of AspectJ, which is an aspect weaver (Kiczales et al., 1997) that is introduced in next section.

AspectJ

AspectJ is a mature aspect weaver for the Java programming (Kiczales et al., 2001; Przybylek, 2018). AspectJ modularizes aspects through dynamically incorporated join point models expressing aspectual pointcuts, advice, and inter-type declarations to Java code. Pointcuts and

advice modify program flow at runtime while inter-type declarations allow static adjustments to classes such as introducing class members and changing class hierarchy.

A pointcut may match a call, or a returning of method call. Also, it can match another pointcuts using the “cflow(..)” identifier. Advice can be executed before, after, or around a pointcut. The below code shows examples of a pointcut and the three ways to inject advice in AspectJ.

```
public aspect AccountAspect {
 private int Account.MAX_WITHDRAW_LIMIT = 1000; // Example of an inter-type declaration
 pointcut withdrawPointcut(int withdrawAmount, Account acc) :
 call(boolean Account.withdraw(int)) && args(withdrawAmount, acc) &&
 target(acc);

  before(int withdrawAmount, Account acc) : withdrawPointcut (withdrawAmount, acc)
  {
    if (withdrawAmount > MAX_WITHDRAW_LIMIT) {
      return false;
    }
  // This advice is executed before calling the pointcut withdrawPointcut.
  }

  boolean around(int am withdrawAmount ount, Account acc): withdrawPointcut
  (withdrawAmount, acc)
  {
    if (acc.balance == 0 || acc.balance < withdrawAmount) {
      return false;
    }
  return proceed(amount, acc);
  }

  after(int withdrawAmount, Account balance) : withdrawPointcut (withdrawAmount,
balance)
  {
  // This is advice is executed after calling the pointcut withdrawPointcut.
  }
}
```

The aspect above should be stored as a AspectJ file ending with ".aj" extension. It also can be written as a regular Java class augmented with AspectJ's special annotations such as (@aspect, @pointcut, @before, etc.). Annotated AspectJ often is flavored to utilize Java tools and IDE support, but it does not support many inter-type declaration capabilities.

```
import org.aspectj.lang.*;
@aspect
public class AccountAspect {
 //Inter-type declarations are challenging in an annotation style.
 @pointcut("call(boolean Account.withdraw(int)) && args(withdrawAmount, acc) &&
 target(acc);")
 public void withdrawPointcut(int withdrawAmount, Account acc) {
 }
  …
 @before("withdrawPointcut(int withdrawAmount, Account acc)")
 public void beforeAdvice(int withdrawAmount){
   if (withdrawAmount > 1000) {
     return false;
    }
  }
}
```

AspectJ aspects can be woven into Java source code, or bytecode at compile time, or load time, and runtime. Bytecode weaving weaves aspects when there is no access to source code for classes or aspects (Laddad, 2003). Runtime weaving binds aspects to classes while they are loaded into a running Java Virtual Machine (JVM).

Feature Oriented Programming (FOP)

Feature Oriented Programming (FOP) is a compositional software engineering approach which seeks flexible composition of classes in order to form SPL features (Prehofer, 2006). FOP lays out features as refinement layers consisting of related classes (class collaborations) and introduces the concept of *lifters* to handle interaction between feature pairs. Features in FOP can be parametrized and then are instantiated with specific classes like generic classes in Java. Despite FOP flexibility to express SPL features, it only extends object-oriented programming (OOP) and cannot be generalized for different types of languages and models.

Algebraic Hierarchical Equations for Application Design (AHEAD) is a FOP architectural model in which the SPL base artifacts are constants whereas feature refinements are functions applied on these constants (Batory et al., 2004). Hence, SPLs can be expressed as hierarchical mathematical equations. A refinement represents addition of new elements or modification to existing fragments of a feature or the SPL's base code. The core idea of AHEAD is to map SPL features to modular implementations called feature modules. Thus, feature modules can incrementally refine existed components in a stepwise manner. AHEAD composition is a polymorphism operation in that each artifact has a specific composer which is responsible to apply the composition.

AHEAD refinements in Java can be implemented using Jak, which is a domain-specific language (DSL) allowing feature refinements (Batory et al., 2004). Jak refines interfaces and classes in Java by inserting the modifier "refine" before declaration of interfaces, or classes. The keyword "original()" is another keyword in Jak that refers to a preceded class or interface in a refinement chain. Composition of Jak refinements is achieved by a composer called Jampack, which layers all refinements to their single targets.

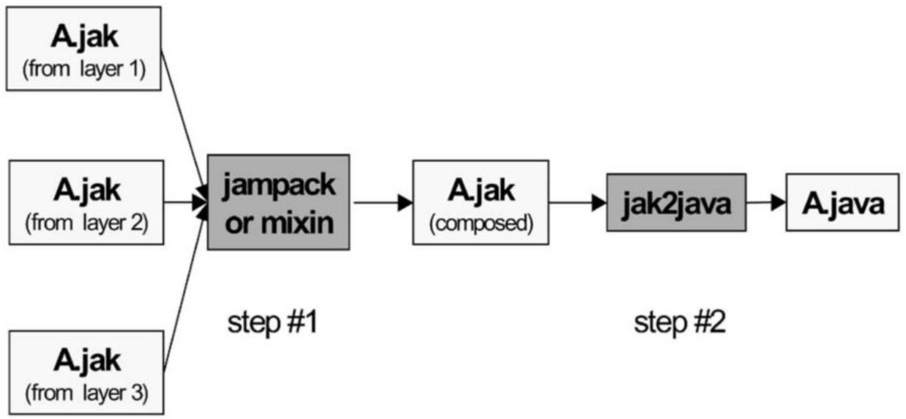

Figure 22: Jak composition to form Java files (Batory et al., 2004).

FeatureHouse is an SPL tool that can compose pieces of code written in different programming languages (Apel, Kästner, et al., 2009). Its builds on the work of AHEAD and utilizes a generic

structure of software artifacts (called the feature structure tree (FST)) to perform feature composition. Figure 23 shows an FST model for a Java program.

Figure 23: Java code and its corresponding FST (Apel, Kästner, et al., 2009) .

Aspectual Feature Modules (AFM)

Aspectual Feature Modules (AFM) combine the two concepts of AOP and FOP as complementary techniques (Apel et al., 2008). Therefore, FOP strengths overcome AOP weaknesses and vice versa. AFM utilizes FOP to offer large-scale composition and feature modularity while AOP can be used to modularize repetitive crosscutting concerns.

There are three criteria that influence the decision of what technique to use to implement fragments of features in AFM: homogeneity of the fragment, whether its static or dynamic, and its relationship to other fragments within the feature (Apel et al., 2008). First, AOP suits homogeneous crosscuts that affect multiple places with repetitive piece of code. On the other side, FOP is tailored for heterogeneous crosscuts that changes several places with distinctive additions for each place. Second, dynamic modifications are well-supported by AOP whereas FOP has no support for dynamic alterations, but it is more expressive to handle static changes. Third, a single module of FOP can encapsulate implementation of a complex feature – especially when its fragments express high cohesion – that requires complicated solution to be encapsulated in AOP aspects.

The major issue of AFM is the requirement to employ two different compositional technologies to implement SPL features. This issue would decrease adoption, increase the learning

curve, and lead to synthesis complexity between the two technologies (when there is a change that is specified using in one technology and requires manipulation of the other one).

Delta-Oriented Programming (DOP)

The ability to delete code fragments from a module to maximize reuse is an additional capability of the compositional approach called Delta-Oriented Programming (DOP) (Schaefer et al., 2010). SPLs according to DOP are comprised of core models (base models) and a set of delta models that add new code to core modes like FOP features. However, deltas may also modify or delete code units from models found in the base, or other included features.

DeltaJava was introduced to achieve DOP in Java (Koscielny et al., 2014). Deltas in DeltaJava allows specifying conditions and execution ordering for deltas. Deltas can add, rename, and delete classes and interfaces. In addition, fields and methods of core models can be added, renamed, and deleted. The code below shows a delta named “DAdd”, which is activated when Add feature is selected (Koscielny et al., 2014).

```
delta DAdd when Add {
  adds class Add implements Exp {
    Exp expr1; Exp expr2;
    Add(Exp a, Exp b) { expr1=a; expr2=b; }
    }
  }
```

5.2 The Complexity of Handling Fine-grained Variability

Compositional approaches achieve variability through introducing new classes, methods and class fields. Still, they don’t offer satisfactory solutions to deal with finer-grained variability that is commonly needed (Kästner, Apel, and Kuhlemann, 2008). Below, we summarize five main reasons that compel software developers to employ fine-grained variability in compositional approaches. The first three reasons are discussed in (Kästner et al., 2007) and (Kästner, 2007) in much detail.

5.2.1 Statement Extension Problem

A feature may insert some code to an existing method. Aspect composition and method refinement (Batory et al., 2004) in FOP are able to handle such addition of code in a similar way. Here we assume that the added code does not interact with (neither depends on nor introduces) local variables in a method, but it can interact with instance variables or method parameters. For method code addition, there are three main scenarios:

The first scenario is when the new code occurs at the beginning or end of a method; aspect composition and method refinement can handle this type of addition through wrappers.

The second scenario occurs when the code implementing a feature is tangled inside some sequence of statements of the SPL base, or feature, code. If the feature's code should be placed before or after a method call within the sequence of the original method, capabilities of AOP such as in AspectJ allow specification of the required join points in certain cases, such as when there is only one call to a given method.

The third scenario occurs if the desired location of the addition is neither preceded nor followed by any method call. It is too complex to implement such an addition using current AOP languages.

Method refinement for FOP is far from AspectJ's capabilities; specifically, it does not offer a straightforward solution for second and third scenarios.

5.2.2 Local Variables Access Problem

In composition approaches, there is no direct access to local variables that do not appear in the parameter list of a method. Although some composition approaches are able to inject code at the end of methods as in the first scenario, the injected code may depend on some local variables that are not exposed.

AOP and FOP use two main workarounds to access methods' local variables. The first is to create hook methods that receive the local variables as parameters. Hence, a key purpose of hook methods is to expose these local variables. The second workaround is to change the scope of local variables to be an instance variable. This workaround decreases code quality and may also lead to inappropriate use of these instance variables.

5.2.3 Local Variable Modification Problem

This issue is related to the case in which the added code requires modifying multiple variables of a method. For example, a feature may change values of some variables at the beginning of a method. In this case, the modification can be implemented via multiple hook methods; one hook method per modified variable. A second alternative would be a single hook method that returns an object of a custom class (often called a method object) containing a list of the changed variables. This alternative requires either a global method object that would contain all needed variables in a class or an object per method. Both approaches add complexity. The first choice requires a sophisticated mechanisms to overcome name clashes, synchronization, and accidental changing of variables. The second choice increases the number of required classes for the SPL.

5.2.4 Scope-Sensitive Statement Problem

Techniques of feature composition are inconvenient to handle increments that involve scope-sensitive statements, which are bound by method context such as “this”, “switch”, “break” and “return” (Benduhn et al., 2016). Although a feature may consist of a one-line scope-sensitive statement, the context in which the statement appears cannot be easily preserved through workarounds. Aspect composition is better than FOP in this regard, as the former offers limited capability to inset new code in certain places within a sequence of the code, however, the addition of “this” and “return” is still problematic and requires complex workarounds. For example, there are 1254 return statements annotated with #ifdef that belong to some features in the C version of the Berkeley DB (Benduhn et al., 2016). Implementing these return statements with a compositional approach poses great challenges.

5.2.5 Exception Handling Problem

Exception handling that occurs in original caller methods is not preserved in hook methods. Therefore, compositional approaches require extra manipulation to incorporate fine-grained increments placed at positions that may raise exceptions, such as places surrounded by try-catch statements. Often, the same exception handling code has to be repeated in each hook method. While AspectJ is considered as an ideal candidate to deal with exceptions since it is flexible to route exceptions raised during method execution, AspectJ may require rewriting of all catch

statements appearing in an original method. The situation becomes worse when the catch statement in the original method uses a scope-sensitive statement.

5.3 Handling Fine-grained Variability in Compositional Approaches

Method overriding, FOP method refinements, and aspect advice are techniques used in compositional approaches to allow extensions to the behavior of methods. These techniques have limited freedom to introduce arbitrary statements inside methods as explained in the previous section. To enable granularity to be even finer, software developers can employ either hook methods, or alternatively they can combine annotation with composition. Here we will discuss these two ways.

5.3.1 Hook Methods

A hook method is a method which has an empty body and is placed in the base of the SPL code and does nothing by default. It acts as a possible point of extension. The frequent need for fine-grained variability may result in extensive, and sometimes excessive, use of hook methods. The large number of hook methods affects code quality by reducing its readability and maintainability (Kästner, Apel, and Kuhlemann, 2008; Murphy et al., 2001). Hook methods can also be complex: although they allow something additional to be done in a variant, that additional work may require passing many arguments. Additionally, for certain kinds of algorithm variation, the multiple implementations of the hook method might need to be very similar, resulting in code cloning.

5.3.2 Combining Annotation with Composition

Annotative approaches in general are capable of adding new code at a fine level of granularity, for instance, at the statement level. It is possible to add extra parameters to a method for certain features, or even to extend expressions such as changing a condition inside an if statement. Therefore, annotations can be used to deal with fine-grained variability cases which in turn will increase the adoption of compositional approaches in practice (Kästner and Apel, 2008a).

FeatureC++ is a combined approach; it is a compositional approach that supports FOP, AOP and incorporates the annotative approach of CPP (Apel et al., 2005). Another combined approach is FeatureCoPP which is an extended preprocessor that allows code composition (Krüger et al., 2016). It uses comments (with a special convention) to mark desired places in the code similar to

the #ifdef mechanism. The distinguishing capability of FeatureCoPP is the ability for the annotations to conditionally call hook methods outside the scope of the methods in which annotations were placed. This is different from what happens with hook methods, which are always called. Methods that are not called in annotations can be deleted from the codebase.

Combined, or integrated, approaches offer software developers easy migration to code composition to implement SPL as they can continue using annotations while incrementally adopting compositional approaches. They seek to balance two competing factors: modularity and granularity. However, integrated approaches lead to decreased traceability and mixing of separate concerns. The fine level of granularity comes with a cost: It damages the modularity of the code (Feigenspan et al., 2013; D. Le et al., 2011) and makes it too hard for features to reuse another features' code in an annotative approach.

5.4 Motivation and Idea

Here we explain how to achieve the goal of fine-grained variability in a compositional approach followed by an illustrative example. To help the reader to get more technical insight about statement-level injections in Umple, we also provide the grammar used to describe labeled aspects.

5.4.1 Code Labels to Support Fine Grained Variability

As we have discussed in the previous section, there are several reasons to supply hook methods in compositional approaches to obtain a high level of variability. Aspect composition is the best in this regard, as it offers limited support for statement-level variability. However, there are some pitfalls of aspect composition such as increased code size, aspect scaling, and the "third-person perspective" (Kästner et al., 2007). The third-person perspective describes referencing the extended class as an external entity without being able to access its local context using the "this" keyword (as first-person perspective) within the advice.

The reason behind the limited support for statement-level variability is the lack of identifiable structural representations to lay out statements inside methods. This omission makes it hard to merge separate modules containing only statements into methods in compositional approaches. This is not the case for the first-class entities such as classes, fields, and methods. They all have

unique identifiers associated with their structures, for example, classes are identified by their names.

In our approach we (re)use code labels within methods to allow finer variability at the statement level. Therefore, code labels act as variability identifiers within statements inside methods. This counters the need for addition of hook methods in compositional approaches. Therefore, code belonging to SPL features can extend the base code in specific places without requiring workarounds. In our approach we impose a condition on using multiple labels: labels inside the method body must be unique; i.e. a label cannot be used more than once in the same method. This condition is already imposed in some programming languages such as C. Figure 24 (a) shows how labels are used in Umple to provide variability inside a method. This example is taken from the work of Krüger et al. (Krüger et al., 2016).

Alternatively, code comments could alternatively be used to identify desirable places for extensions inside methods instead of code labels. Code commenting is considered a good practice and often offers valuable documentation. However, we argue that code labels are more effective than code comments for the purpose of being extension points due to their low frequency of use and their semantics. Code comments are usually present in countless places in the source code. From a semantics perspective, labels are processed by compilers to ensure uniqueness and they carry connotation of informing the reader about the specific purpose of a block of code because they were originally designed to be destinations of control-flow statements. For example, Java labels are used to direct the execution after encountering break statements in nested loops.

```
1     class Main {
2      public static void main(String[] args) {
3       Hello_label:
4       Beautiful_label:
5       Wonderful_label:
6       mixset World {
7        System.out.print (" world !");
8       }
9      }
10    }
11
12    class Main {
13     mixset Hello {
14      after Hello_label: main(String ) {
15       System.out.print(" Hello ");
16      }
17     }
18     mixset Beautiful {
19      after Beautiful_label:
20        main(String) {
21       System.out.print(" beautiful ");
22      }
23     }
24     mixset Wonderful {
25      after Wonderful_label:
26        main(String) {
27       System.out.print (" wonderful ");
28      }
29     }
30    }
31
32    use Hello;
33    use Wonderful;
34    use World;
35
36    require subfeature [ opt Hello];
37    require subfeature [ opt Beautiful];
38    require subfeature [ opt Wonderful];
39    require subfeature [ opt World];
```

a) Implementation of HelloWorld SPL in Umple.

```
1     public class Main {
2      public static void main(String[] args) {
3       Hello_label:
4        System.out.print (" Hello ");
5       Wonderful_label:
6        System.out.print ("wonderful ");
7       System.out.print (" world !");
8      }
9     }
```

b) A Java variant generated from (a) containing the Hello, Wonderful, and World features, but not Beautiful

Figure 24: Umple code in (a) and the generated Java code in (b), showing activation of features with fine-grained variability.

5.4.2 Code Injection Providing Direct Access to Method Context

The issues mentioned in the background section demonstrate that compositional approaches are burdensome when implementing SPL projects containing massive numbers of statement-level changes, and SPLs with resource-limited contexts.

The domain of operating systems is one of the best examples for the first category. From our observations, it is extremely hard to implement the Linux kernel using a pure compositional

approach due to the number of required workarounds. This observation is supported by Reynolds et. al's study in applying AOP to some configurable options of Linux kernel's code (Reynolds et al., 2008).

An SPL implemented with OOP is not applicable for highly-constrained devices that have limited hardware capabilities (CPU, screen size, RAM, battery and user interface) (Pleumann et al., 2002). Although compositional approaches offer capabilities beyond OOP, they heavily rely on polymorphism or dynamic calls that make them not applicable for highly constrained devices. In Java for instance, there will be performance and space overhead due to JVM creation (and then destruction) of stack frames, which include method information and local variables for each method call (Oh et al., 2012). The emergence of open-source preprocessors targeting Java such as Antenna (Pleumann et al., 2002) shows the need for practical SPL capability tailored for mobile devices with a limited memory (up to 30 KB). These small devices often cannot load large JAR files containing all variations of an SPL.

Mixsets reduce the need for programming workarounds when creating variants by allowing plain code to be injected into methods with full access to their local context. Therefore, the injected code can access local variables defined before the place of injection, update values of local variables, and use scope-sensitive keywords in feature modules.

5.4.3 HelloWorld SPL Example

An example HelloWorld SPL is shown in Figure 24. The SPL has four features captured in the following mixsets: Hello, Beautiful, Wonderful, and World.

Lines 1-10 of part a) contain the base code of the SPL except the inline mixset "World". The main method has three explicit points of extension represented by the labels: Hello_label, Beautiful_label and Wonderful_label.

Lines 6-8 show an inline mixset called World. This shows how Umple can accommodate annotative conditional compilation in addition to the compositional approach that is the focus of this chapter. This block could be refactored into a separate file to make the approach purely compositional.

Lines 12-30 contain feature fragments. These fragments could be scattered across the code or imported from different files. All of them are in class Main. Each fragment is in a mixset, with the fragments starting at lines 13, 18 and 24. Each fragment injects different code at specific labels in the main method.

Lines 32-34 consist of three use statements which act as a configuration of an SPL variant. The variant is generated as Java code in Figure 24 (b). Note that 'Beautiful' is not selected.

Lines 36-39 consist of require statements which define the feature model of the SPL. The feature diagram is shown in Figure 25. These require statements are simply specifying that the four mixsets are all optional.

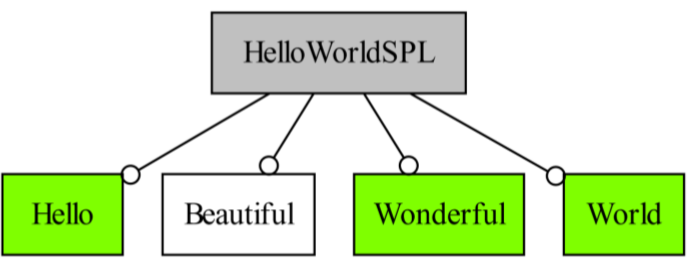

Figure 25: Feature diagram generated by Umple for the SPL according to lines 36-38 of the code. The SPL base is the default-selected feature (gray color), while selected features are in green.

One should bear in mind that while code labels may facilitate the integration of features, they are not part of any feature. Thus, a possible optimization to reduce code size might be achieved by reusing code labels for multiple features. In the case of HelloWorld SPL, only one label could be used for all three injected features since they all are injected at the same logical place.

The behavior of variants in an SPL may change according to composition order of features. This property is called order flexibility (Lopez-Herrejon et al., 2005). When reusing one label for multiple mixsets, the order of the use-statements can impose the desired order for feature composition. Enhancing the mixsets with a declarative composition language such as Marot and Wuyts' DSL is still possible (Marot and Wuyts, 2008). In this particular example, the use-statements have no effect on the sequence of feature composition because each feature has a unique place for its code. The flexibility in the way in which features could be integrated is an advantage

for composition approaches over their counterparts since this flexibility cannot be achieved easily in annotative approaches.

5.4.4 Code Injection Grammar

The grammar excerpt shown below is an extract of the full Umple grammar describing the utilization of aspects to enable insertion of code segments inside methods. The complete grammar of Umple can be found in Umple's online manual (*Umple Grammar*, 2021).

Mixsets were introduced in an earlier in Chapter 3 and we will not reproduce the complete grammar for them here.

The Umple EBNF grammar notation follows these conventions:

- Terminal constant symbols are shown outside the scope of any brackets and are highlighted in red in the documentation (e.g. 'after' in rule 6).
- Terminals that are parsed using special expressions are shown with single square brackets (e.g. **code on lines on lines 3 and 7 transfers control to a parser for an embedded programming language such as Java; !codeLabel on line 7 transfers control to a regular expression parser.) These are highlighted in green.
- Nonterminal rule names are in blue.
- References to rules are in double square brackets.

For compactness and clarity, three dots in the extract below means omitted rules because they are not related to the work presented in this thesis.

Rule 1 specifies the classContent rule which is referenced in the classDefiniftion rule (not shown in this extract of the grammar). Rule 1 says that a class can have a concreteMethodDeclaration (Rule 2) or a softwarePattern (Rule 4).

Rule 2 (concreteMethodDeclaration) specifies rules required to define a method in a class. The rule includes methodBody (rule 3), which captures method body code as plain text in [**code]. The body may also contain labels. Therefore, labels are identified and processed by an embedded programming language parser.

Rule 4 (softwarePattern) includes rules for aspect injection (rule 5) in addition to other rules. Rule 5 includes two common types of aspect injections (before and after).

Rule 6 is the after-injection rule; the rule for before injections is not shown and is similar.

The last rule (7) is the aspectBody rule. It has an optional codeLabel, followed by a colon, and indicates a method, and code to be injected. The only grammar change for the work reported in this chapter is the codeLabel, that allows specifying where in the methodBody to inject the added code fragment.

```
1  classContent- :  … | [[concreteMethodDeclaration]] |
   [[softwarePattern]] | ...

2  concreteMethodDeclaration :  ... [[methodBody]]  ...

3  methodBody- :  { ( [[precondition]] | [[postcondition]] ) [**code] }

4  softwarePattern- :  [[codeInjection]] | ...

5  codeInjection- :  [[beforeCode]] | [[afterCode]]

6  afterCode :  after [[aspectBody]]

7  aspectBody- : ([!codeLabel:\S+:])? [[injectionOperation]] { [**code] }
```

5.5 Limitations

This section highlights limitations of mixsets as a compositional approach offering fine-grained variability. We will discuss expression-level variability, code label support in programming languages, the requirement of AOP to apply statement level composition, code comprehension, and type checking.

5.5.1 Expression-level Variability

Granularity of mixsets as a compositional approach is elevated to a higher degree via code labels but still mixsets are not capable of handling expression-level variability. Augmentation of method signatures is an example of a change at the expression-level which would require a

workaround such as completely separate method declarations in our approach. Although the undisputed champions when it comes to fine-grained variability are annotative approaches, particularly CPP, the lack of expression-level variability in mixsets should not hinder it practically. Because there are case studies that showed the rarity of expression-level variability in software projects (Couto et al., 2011; Liebig et al., 2010); we argue that a few workarounds – especially in large software projects – can be justified.

5.5.2 Code Label Support in Programming Languages

The mixset approach relies on code labels to inject code inside methods. However, code labels are not supported by some mainstream programming languages. In this case, it may be best to instead specify locations inside methods for code injection using code comments, such as the mechanism of FeatureCoPP; an alternative would be to use labels that the Umple compiler then removes before passing to the target language compiler. When code comments are adopted because of the absence of code labels in a certain programming language, comments that indicate method extension points need to be distinguished using a special syntactic comment form.

5.5.3 AOP to Apply Statement Composition

Statement level injection in the implementation of mixsets assumes AOP availability in the host language. However, this is not a condition to enable mixsets occurring at the statement level. In the absence of AOP, composition of statement level fragments alternatively could be modeled as merely mixin fragments. For example, the code 12-17 of Figure 24 (a) could be modeled as the following mixin fragment:

```
Class Main {
  Mixset Hello{
     main (String ) {
       Label Hello_label {
         System.out.println(" Hello ");
    }
  }
}
```

The above mixin fragment will be composed into the Main class – specifically into the main method that has a string argument. We have not implemented this in Umple, since Umple uses the AOP approach; we simply present the above as a potential alternative.

5.5.4 Code Comprehension

When developing SPLs using mixsets, developers need to add new labels whenever they want to inject statement-level changes. At the same time there is a reduced need to introduce hook methods to manage expression-based variability. As Figure 26 demonstrates, showing all labels (in the original files) as potential injection points, which may or may not be used, might reduce the code quality. This potential is magnified if they use arbitrary names for the labels. Therefore, meaningful names should be used to improve code quality. Tool-based assistance could reduce the number of code labels if there is a chance to reuse a single label for multiple features. Developers might also be advised to consider adding a comment before labels, such as "// Additional code will be injected here if the X feature is activated"

```
//File: umpleSPL/src/generators/Generator_CodeUmple.ump
private void initializeParser()
  {
    if(parser != null) { // Parser is shared data, we need only generate once.
      return;
    }
    parser = new cruise.umple.parser.analysis.RuleBasedParser();
    parser.addGrammarFile("/umple_core.grammar");
    Label_Class_7:;
    Label_Trait_8:;
    Label_FIXML_9:;
    parser.addGrammarFile("/umple_patterns.grammar");
    Label_StateMachine_10:;
    Label_Trace_11:;
    Label_Template_12:;
    Label_Structure_13:;
    parser.addGrammarFile("/constraint/umple_constraints.grammar");  // TODO Under development
    Label_Layout_14:;
    parser.addGrammarFile("/umple_exceptions.grammar");
    parser.addGrammarFile("/use.grammar");
    Label_Filter_15:;
    Label_Mixset_16:;
    parser.setupRules(true);    // Compute the Umple grammar rules
    rootRuleToken = parser.getRootToken();
    if(rootRuleToken == null) {
      throw new UmpleCompilerException("rootRuleToken retrieved from parser is NULL.", null);
    }
    analyzer = parser.getAnalyzer(); // Add this method to the RuleBasedParser
    rootRule = analyzer.getRules().get("$ROOT$");
    if(rootRule == null) {
      throw new UmpleCompilerException("Parser's rootRuleToken is null", null);
    }
  }
```

Figure 26: Example of multiple labels found in a method taken from the compositional UmpleSPL.

Because of the issue of feature delocalization, feature fragments may not be easily understood unless the developer knows the full context where these fragments are placed. To reduce the effect of delocalization, a tool-based solution should be implemented (in our case, UmpleOnline and other Umple tools). The basic idea is to offer the ability to view the code resulting from merging one or more features or with the SPL base code. That said, if a person comes across a label name either in the base code where the code will be injected, or in the feature code containing the code to be injected, a simple search using any search tool (such as grep) should enable bringing the two together. This should be easier than what might happen in AOP where the set of join points matching a pointcut may be hard to determine.

5.5.5 Dynamic Variability

AOP has the capability to extend methods' behaviour at run-time. This is applicable to AFM as it includes AOP, which supports dynamic variability for SPLs. Dynamic variability is required

in some domains in which software systems may behave differently according to certain contexts. Mixsets currently do not support dynamic variability. We envision the possibility to extend mixsets' capability for run-time variability via changing the generation of code from inline mixsets designated as 'dynamic' to if conditions (with parameters) that can be set at run-time. Still, this solution is not comparable to the sophisticated facilities of AOP in routing dynamic calls in different situations (before call, during execution, and after termination of method calls).

5.5.6 Type Checking

As Umple allows mixing design modeling with native code (of programing languages), only mixsets which are in-use are checked against the metamodel of Umple. Mixsets that are not part of the selected variant are not checked. Also, code blocks of native languages are not always checked. This poses a limitation for mixsets and requires further investigation.

A possible solution would be offering plug-ins for type checking tailored to a particular target language. A type system can be activated during SPL development to check type conformance to a particular programming language. Since Umple is language-independent and offers capability to merge code written in different languages, the automatic detection of code language in certain parts and then applying the right type checking is another capability for which we are investigating its usefulness in practical SPLs.

5.6 Summary

This chapter discusses the weaknesses of current compositional approaches in accessing local contexts and in identifying variation points within the sequence of methods statements. The implication of this is hook method overuse to deal with fine-grained variability. The mixset approach resolves this matter through two techniques. First, it uses code labels to indicate variation points within methods' statements. Code labels are favored over code comments to mark locations of variability due to their sematic connotation and their support by important programming languages. Second, mixsets allow aspect code to directly access method bodies. The direct access to method context avoids the five problems listed in Section 5.2.

Chapter 6 Berkeley BD JE Compositional Case Study

As we were designing and implementing mixsets in Umple, we studied SPL case studies that fall into the category of object-oriented design, which is fundamental to most of current software projects. The aim is to help us to assess mixsets' strengths and weaknesses compared primarily to other compositional approaches but also to annotative approaches. Initially, we evaluated mixsets with small-scale case studies found mainly in the SPL2go website (*SPL2go*, 2011), and FeatureIDE examples (Thüm et al., 2014).

After we obtained confidence in the capabilities and scalability of mixsets, we looked for larger case studies. A relevant case study that we found practical is the SPL (Java version) of the legacy code of Berkeley DB (Seltzer, 2007). It is a typical medium-scale open-source project that is used commercially and has been used in the literature to evaluate several other SPL approaches (Kästner et al., 2007; *SPL2go*, 2011).

We introduce the Berkeley DB JE in the next section. In subsequent sections we describe the refactoring procedure, and then we analyse and discuss the result of the case study.

6.1 Berkeley DB Java Edition

Berkeley DB, which became Oracle Berkeley DB in 2006, is an embedded high-performance database system that was first released in 1991 and is available as open source (Olson et al., 1999). It was written originally in C but was later implemented for various programming languages including Java.

The Java Edition (JE) of Berkeley DB is a mid-sized application with 300 classes, around 84,000 LOC, and detailed documentation for usage and interfaces (Kästner, 2007). Because it is designed for embedded use, some features of conventional relational database management systems are omitted. It forms an excellent software product line with a long list of features including high concurrency, automatic recovery, indexes, an in-memory cache, full ACID (atomicity, consistency, isolation, durability) transactions, a high-performance no-rewrite storage system, and numerous background threads. Berkeley DB JE is designed and implemented in a state-of-the-art object-oriented manner (Kästner, 2007).

6.2 Procedure

Berkeley DB JE was originally refactored by Kästner et al. into a feature-oriented SPL using CIDE (Kästner, Apel, and Kuhlemann, 2008), FeatureHouse (*SPL2go*, 2011), and AspectJ (Kästner et al., 2007). Figure 27 depicts the feature diagram of Berkeley DB JE. The refactoring process resulted in 38 features scattered in more than 1100 extensions. Table 5 shows how they have categorized these extensions.

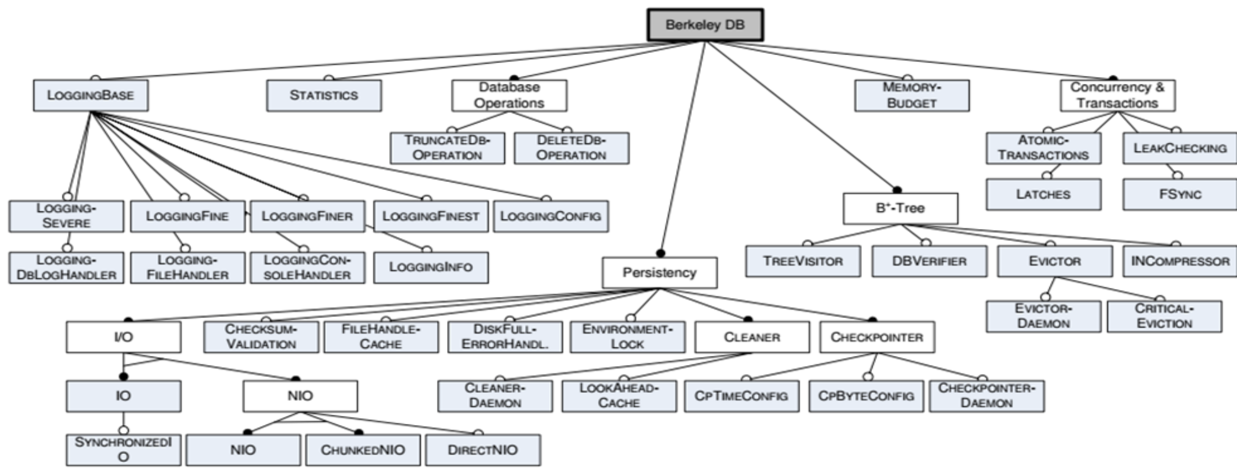

Figure 27: Feature diagram for Berkeley DB JE SPL as illustrated in (Kästner et al., 2007).

The next four sections explain the tasks followed in this procedure. Then Section 6.2.5 shows an example comparing our mixsets implementation against the FeatureHouse implementation. Finally, the validation of the mixsets implementation is discussed in the last section.

Table 5: Extension types resulting from refactoring Berkeley DB JE into FOSD.

Category	# of Extensions	%	Description
A: entity addition	640	56%	Introducing new interfaces, classes, methods, or fields.
B: in-method	214	19%	Extending behavior of methods using before, after, or around.
C: fine in-method	261	23%	Fine-grained insertion of code in the middle of a method.
D: expression	24	2%	Expression-level modifications.

6.2.1 Umplification of Berkeley DB JE

To obtain better understanding of FOP approaches, we decided to transform the compositional SPL version of Berkeley DB JE written in Java/FeatureHouse, into Umple, with mixsets to

represent SPL features. The version we started with is available in the FeatureIDE and the CIDE repository (Kästner, 2012). Initial translation to Umple was accomplished by a tool called the Umplificator (Garzón et al., 2014). Then we transformed each feature's fragments written in FeatureHouse code into mixset fragments. Although it is possible to refactor the pure Java version into Umple, the FeatureHouse version has the advantage of separating the SPL features into dedicated folders. Our implementation is available in a GitHub repository called "BerkeleyDbUmple" (Algablan, 2020).

6.2.2 Retain Original Bodies Out from Chains of Hook Methods

Calls of hook methods found in the FeatureHouse implementation were deleted and replaced with labels in almost all cases. Chains of hook methods were refactored by moving bodies of hook methods back to their original positions in the base code since mixsets allow statement insertion in methods in specific locations. Hook methods in FeatureHouse are named as "hook" followed by a number. We followed a similar convention for variation points inside methods although we encourage better naming convention when using labels as extension points. We added labels named "label" followed by the same number used in the matched hook methods as much as possible.

6.2.3 Remove Static Inner Classes

An important issue we encountered is that to address fine-grained granularity at the statement level, FeatureHouse introduced 219 static inner classes in the case study. The purpose of these inner classes is to expose the local variables of methods by making them instance variables of the inner classes. In each case, the original method instantiates a new object of an inner class and then it makes a call to a specific method (called "execute()") which resembles the body of the original method. Considering 38 core features, there are almost 6 methods in each feature that required refactoring with inner static classes. Large features often have more inner static classes. For example, there are 20 static inner classes in the medium-large scale feature called Latches, scattered in 13 original classes.

6.2.4 Erase Customized Throw Statements

Another issue worth remarking on is the introduction in the FeatureHouse SPL of custom-made throw statements such as "throw new ReturnBoolean(false);" and "throw new

ReturnObject(...);". These act as a workaround to the Scope-Sensitive Statement Problem in which a hook (the overriding) method throws an exception with a wrapping object that needs to be caught either in the base, or the overridden, method. The wrapping object could be ReturnInt, ReturnObject, ReturnBoolean, or ReturnVoid. For example, a customized ReturnObject exception with null value can be thrown to interrupt the execution of a hook method. This requires the error type to be caught in the overridden base method to terminate or to return a specific value. There are 87 throw statements of this type. The code below shows the code of the ReturnInt class:

```
public class ReturnInt extends RuntimeException {
    public int value;
    public ReturnInt(int v) {
      value = v;
    }
}
```

6.2.5 Example

As a direct and understandable example, we present some code from the BIN class. Figure 28 shows an excerpt of the original implementation (before refactoring) of the BIN class. The blue-shaded area inside isValidForDelete() is the part that is refactored as a fragment belonging to the Latches feature of the SPL. The green-shaded area is a part of the code that should remain in the base code after refactoring to SPL. To make the example easier to read in this thesis, we removed certain unimportant lines and replace them here with "...". We selected this example purposely because it is direct and understandable.

Table 6 demonstrates the required change in the base code (top) and in the Latches feature (bottom) using FeatureHouse (left) and mixsets (right). We use a red-bold font to highlight mechanism-specific syntax.

The FeatureHouse implementation works as follows:

• The keyword "original(...)" in lines 81 and 94 of the FeatureHouse code is used to call the matched methods in the superclass.

• The fragment between the lines 872-899 of the original code has been refactored into the method hook607() between lines 525 and 548.

• Method hook607() in turn calls hook608(...) which appears on line 549 in the refactored base code, and does nothing.

• Method hook608 is overridden at line 89 to include the condition coming from line 869 if the Latches feature is selected.

• To enable returning value of hook607() in the original method, Featurehouse throws exceptions called ReturnBoolean. Various different Return objects are reused through the SPL in FeatureHouse.

In the second column of Table 6, the same code is refactored using mixsets. The amount of change is very low while the understandability is not damaged as in the case of FeatureHouse.

```
37     public class BIN … {
…      …
859      boolean isValidForDelete() throws
             DatabaseException {
…          …
867        boolean needToLatch = !isLatchOwner();
868        try {
869          if (needToLatch) {
870            latch();
871          }
872          for (int i = 0; i < getNEntries(); i++) {
…              …
877          }
878          if (numValidEntries > 1) {
879            return false;
880          } else {
881            if (nCursors() > 0) {
882              return false;
883            }
884            if (numValidEntries == 1) {
885              Node child = fetchTarget(validIndex);
886              if (child == null) {
887                return false;
888              }
889              child.latchShared();
890              boolean ret = child.isValidForDelete();
891              child.releaseLatch();
892              return ret;
893            } else {
894              return true;
895            }
896          }
897        }
898        finally
899        {
900          if (needToLatch && isLatchOwner()) {
901            releaseLatch();
902          }
803        }
904      }
```

```
...
```

Figure 28: Excerpt of original implementation of the BIN class.

An alternative refactoring, which keeps the calls to implementation hook methods, may be proposed. Hook methods can act as labels in which mixsets will inject feature code. This alternative has the advantage of easing the refactoring process, however, it still carries all the problems of hook methods that we are trying to resolve.

6.2.6 Validation of Mixsets' Implementation

To validate the syntactic correctness of our implementation, we generated SPL variants for the base, all features, and various combinations of features. Checking the syntax correctness of *all* SPL variants of Berkeley DB is a complex problem since there are more than 3000 valid feature combinations. To validate the semantics, we run some examples provided in the original source code package, but we encountered several bugs related to the Latches feature. We realized that the FeatureHouse implementation contains these bugs too – and we had just inherited those bugs. Therefore, we decided to refactor the Latches feature based on the original source code. After that, we were able to run the Latches unit tests and the examples.

Table 6: FeatureHouse v.s. Mixset implementation of IsValidDelete() in Berkeley Latch feature.

SPL Base — FeatureHouse

```
24   public class BIN …{
…    …
436    boolean isValidForDelete() throws
           DatabaseException {
437      try {
…          …
440        boolean needToLatch = false;
441        this.hook607(validIndex,
     numValidEntries,
442          needToLatch);
443        throw ReturnHack.returnBoolean;
444      }
445      catch (ReturnBoolean r) {
446        return r.value;
447      }
…      }
525
       protected void hook607(int validIndex, int
           numValidEntries, boolean needToLatch)
526        throws DatabaseException {
527      this.hook608(needToLatch);
…        for (int i = 0; i < getNEntries(); i++)
532  {
533        …
534      }
53-      if (numValidEntries > 1) {
537        throw new ReturnBoolean(false);
538      } else {
539        if (nCursors() > 0) {
540          throw new ReturnBoolean(false);
541        }
542        if (numValidEntries == 1) {
             Node child =
54-  fetchTarget(validIndex);
545          throw new ReturnBoolean(child !=
546  null
547              && child.isValidForDelete());
548        } else {
549          throw new ReturnBoolean(true);
           }
         }
       }
       protected void hook608(boolean
     needToLatch)
           throws DatabaseException { }
```

SPL Base — Mixsets

```
2    class BIN {
…      …
430    boolean isValidForDelete() throws
           DatabaseException {
…        …
433      boolean needToLatch = false;
434      try {
435        Label607:  // extension point.
436        label608:  // extension point.
437        for (int i = 0; i <
     getNEntries();
…              i++) {
441          …
442        }
443        if (numValidEntries > 1) {
44-          return false;
446        } else {
447          if (nCursors() > 0) {
448            return false;
449          }
450          if (numValidEntries == 1) {
               Node child =
451              fetchTarget(validIndex);
               boolean ret =
452              child.isValidForDelete();
45-            child.releaseLatch();
455            return ret;
456          } else {
457            return true;
458          }
459        }
460      }
461      finally {
462      Label607_finally: // extension
…    point
         }
       }
```

Latch Feature — FeatureHouse

```
2    public class BIN {
…    …
78     protected void hook607(int validIndex, int
           numValidEntries, boolean needToLatch)
           throws DatabaseException {
79       needToLatch = !isLatchOwner();
80       try {
81         original(validIndex, numValidEntries,
             needToLatch);
8-       } finally {
84         if (needToLatch && isLatchOwner()) {
85           releaseLatch();
86         }
87       }
88     }
89     protected void hook608(boolean needToLatch)
90         throws DatabaseException {
91       if (needToLatch) {
92         latch();
93       }
94       original(needToLatch); }
95   }
96
```

Latch Feature — Mixsets

```
2    class BIN {
…    …
75   after Label607: isValidForDelete() {
76       needToLatch = !isLatchOwner();
77     }
78
79     after Label608: isValidForDelete() {
80       if (needToLatch) {
81         latch();
82       }
83   }
84
85     after  Label607_ finally:
86       isValidForDelete() {
87       if (needToLatch && isLatchOwner())
88   {
89         releaseLatch();
90     }
…    }
     …
```

6.3 Analysis and Discussion

The differences between the implementation of mixsets and FeatureHouse are highlighted in the next two sections. Furthermore, we provide answers to potential questions related to AOP as an alternative, result generalization, and the possibility to reduce hook methods of the FeatureHouse implementation via refactoring.

6.3.1 Labels vs. Hook Methods as Variability Mechanism

The FeatureHouse implementation of Berkeley DB JE contains 861 invocations of hook methods. An invocation of a hook method can be considered as an extension point, or variability point, in the SPL. These hook calls are mapped to 1736 code fragments which are either method definitions, or method overriding of existing methods. Some hook methods themselves call further hook methods. We call the latter *cascading extensions*. There are 123 cascading extensions in the Berkeley DB JE implemented with FeatureHouse.

Labels in mixsets are used to indicate extension points. Contrary to a hook method, which has an explicit beginning and ending through which aspects can be injected, a single label can only indicate a unique point of variability in code. Therefore, when there is a need to create an extension that surrounds a sequence of statements inside a method, this implies that an extra end label has to be added in the code. This explains why our mixset-based SPL has more extension points (903) in the implementation of Berkeley DB JE than the FeatureHouse version, which had 861 calls for hook methods.

The majority of labels are placed in the positions where FeatureHouse called hook methods, and are used to indicate a starting point of variability. A few labels are introduced to indicate the end of injection position; 21 labels introduced as closing labels, one label introduced as a starting label and one already reused two labels of the base code. Figure 29 shows the distribution of the injection types used in the mixset approach. The figure shows that after-injection is heavily used in our implementation. In fact, we used after-injection as a convention because we think it is more meaningful to add code after a label, however before-injection can be used in a similar way. A cascading extension occurs when a feature fragment introduces a label that did not exist in the base code. There are 11 cases of label introduction into the base code.

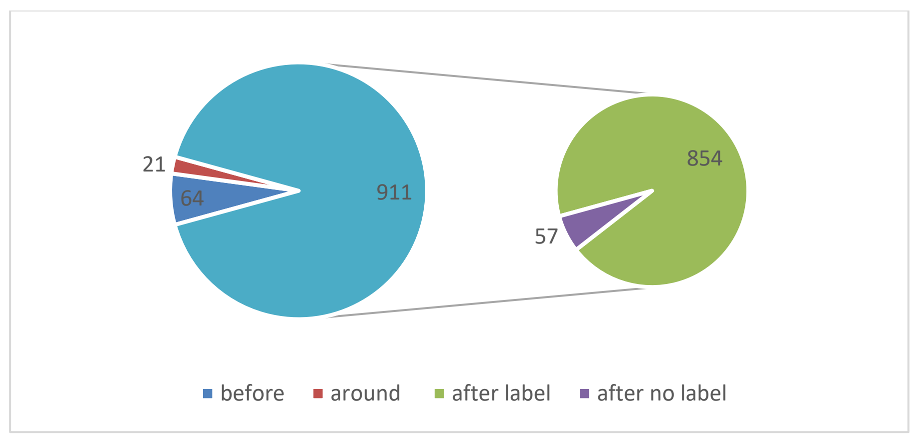

Figure 29:Types of injection used in mixset approach in the implementation of Berkeley DB JE. The right circle breaks out the 911 after cases.

6.3.2 Comparison between Mixsets and FeatureHouse

In this section, we compare mixsets against FeatureHouse in terms of number of extension points, fragments, and cascading extensions; higher numbers negatively affect SPL code quality – particularly code size, understandability, and maintainability. Table 7 highlights differences between mixsets and FeatureHouse in the three criteria. The table shows that mixsets have a slightly higher number of extension points in the implementation of Berkeley DB JE case study. However, mixsets considerably outperform FeatureHouse by needing fewer code fragments and cascading extensions. Mixsets decrease the number of code fragments nearly to half (43%). Moreover, more than 85% of the cascading extensions were eliminated in the mixset approach.

Table 7: Mixset v.s. FeatureHouse implementation of Berkeley DB JE.

Approach	Extension Points	Fragments	Cascading Extensions
Mixset	903 Labels	996	11
FeatureHouse	861 hook method invocations	1736	123

Excessive method overriding used for hook methods increases code size and reduces running performance. We have encountered numerous empty hook methods in the FeatureHouse implementation. For example, the SPL base contains 719 method definitions for hook methods whereas majority of them have empty bodies.

Cascading extensions reduce the overall understandability because software developers will spend more time navigating through several code blocks, which may be placed in different locations. Similarly, SPL maintainability becomes harder with more extension points, fragments, and cascading extensions.

6.3.3 Why Not AOP?

The massive volume of code injection which occurs in our mixsets may raise the question of why not to embrace AOP. In fact, AspectJ has been used to implement this same case study (Kästner et al., 2007). The result was not promising as the authors concluded after implementing around 10% of the SPL that AspectJ is not suitable for this system. The authors pointed to limitations such as method-access modifiers, complexity and length to specify pointcut designators, and indirect access to the extended objects. These limitations made it not feasible to implement the case study with AspectJ.

6.3.4 Generalization of Result

Mixsets as a compositional approach reduce the code size and would increase understandability (and hence maintainability) for the Berkeley DB JE case study. To arrive at a conclusion based on only this case study would not be fair; however, the magnitude of the Berkeley DB JE and the way in which it was designed (as a legacy system that was not originally designed as an SPL) allows us to argue that mixsets will achieve similar results for mid-sized and large-size SPLs. For small-scale SPLs we have observed that mixsets provide less of an improvement due to the limited need of fine-grained variability extensions.

6.3.5 Mixsets Against CIDE

Comparing mixset to the annotative approach of CIDE in terms of the number of extension points; both approaches require the same number. However, CIDE and annotative approaches in general generate fewer code fragments compared with composition approaches since the later add extra boilerplate code for each code fragment. Cascading extensions also require boilerplate code in composition approaches. The slightly more code in compositional approaches pays off in feature modularity and feature tractability.

6.3.6 Possible Refactoring to Improve the FeatureHouse Implementation

Lastly, one may validly argue that a better refactoring might have yielded fewer hook methods and fewer cascading extensions in the FeatureHouse implementation of Berkeley DB JE. We believe the claim is valid for the mixset approach too since Umple has additional abstraction mechanisms which can be utilized to improve code quality. For example, a trait in Umple can provide a method template which can be customized to several feature fragments. These abstraction mechanisms were not employed for the work presented in this chapter. We tried to limit our work in this case study to only mimic changes required in annotative approaches. The goal is to reduce refactoring effort and to be able to compare with FeatureHouse.

6.4 Summary

This chapter presents the application of mixsets to implement a feature-based SPL of the Oracle Berkeley Database JE. The database SPL was umplifed and then refactored into FOSD. The implementation of FeatureHouse, as a typical compositional SPL approach, employs a large number of hook methods, abundance of inner static classes access methods' local variables, and countless of try-catch blocks, which are notorious users of resources. The case study shows that the implementation of mixsets eliminates the need for massive numbers of hook methods, inner static classes, and try-catch blocks. It shows promising improvement in the SPL code quality – specifically code size, performance, and understandability.

Chapter 7 Refactoring Umple into a Feature-Oriented SPL

This chapter documents our experience and the challenges we faced during refactoring a significant portion of Umple from a single software system into a feature-oriented SPL using the approach of mixsets. This case study is significant in that it shows how variability can be applied in model-driven development (MDD) and in the development of a language and of code generators, areas where it had not been fully explored. Therefore, this case study contributes to the SPL community with insights on refactoring a modeling tool into a feature-oriented SPL.

The contributions of this case study are twofold. First, we have extracted a featured-oriented SPL including ten features from an MDD tool; the variability spans both models and native embedded code, an issue not previously investigated in the literature. Our study is publicly available and can be further used to evaluate compositional techniques such as AOP and FOP. Secondly, we have characterized the granularity of variability for the features considered in our study. The results of this characterization can be used to understand the features, their interaction, and to shed light in the practicality of a fully compositional SPL.

7.1 Refactoring Objectives

The decision to refactor Umple into an SPL was taken due to the following reasons:

- Umple is becoming larger as time passes, hence, a separation mechanism to underline interconnected concerns and to achieve better understandability of the code becomes necessary.
- There are few case studies which explore SPL in the domain of MDD techniques and their tools. The contribution of a case study focused on a feature-oriented SPL in a model-driven tool will be a useful asset for the SPL community. Umple is suitable candidate since it has been implemented using MDD techniques as well as it can embed Java code.
- We would like to apply and evaluate our experience with mixsets to refactor a single software product to a software product line. Although we have applied mixsets to the Berkeley DB case study, we relied on the work of Kästner et al. (Kästner et al., 2007) to identify Berkeley DB features. The Umple case study has the objective to enrich SPL

literature by documenting our whole experience with refactoring a single software system into a software product line.

- To provide a case study to be used to compare SPL techniques in Java. While Umple has its own syntax, generated Java code can be traced back to corresponding features using Git commands, which can show differences between Java files when compiling and activating Umple features.
- To allow slicing Umple based on its features. The feature-based separation helps to derive variants of Umple that focus on certain domains. A potential use case is deriving a variant that deals with modeling of state machines only. Another potential variant is a version of Umple that contains mixsets only, which could be used as a generic SPL tool.
- To apply mixsets on a domain (Umple) in which the thesis' author and his supervisor have deep understanding and technical expertise.

7.2 Overview of the Umple Codebase

Since its inception and prior to this refactoring, Umple was implemented as single software system without being represented as a product line. Through the years, numerous capabilities have been incorporated into Umple, some of them as entire thesis topics. Associations (Badreddin, Forward, et al., 2014b), state machines (Adesina et al., 2018), traits (Abdelzad and Lethbridge, 2017), tracing (Aljamaan and Lethbridge, 2012), test generation (Almaghthawi, 2020), and mixsets modeling fall in this category.

Contributors to Umple follow agile practices including test-driven development and continuous integration (CI). Every increment or bug-fix should be complemented with a test case.

The code base of the Umple compiler is a set of Umple files, since Umple is written in itself. Naming conventions are used distinguish between key subsystems such as metamodel, parsing, and generator files.

7.2.1 Umple Architecture

Although Umple has extensive documentation in the code repository, here are few highlights to help the reader to understand the Umple architecture for our case study:

1. Umple Parser

    The UmpleParser directory contains a parser library, written in Umple, to allow parsing in any application; in the Umple compiler it is used to parse Umple files. The parser tokenizes source input according to specification expressed through grammar filles (with suffix ".grammar"). Parsed tokens are then analyzed in the UmpleInternalParser class (found in the cruise.umple/src directory, below). Currently, building Umple from its source code entails parsing and analyzing .ump files to form an internal representation of the Umple language, and then invoking a Java code generator to form Java code, which is finally deployed in the "umple.jar" executable.

2. Modeling Capabilities

    The cruise.umple/src directory has the necessary source code for all model-oriented development technologies in Umple. A modeling capability (e.g. for state machines, mixsets, etc.) generally consists of three components: a grammar file, code to process tokens after parsing (in UmpleInternalParser or in a parser for the specialized part of Umple), and a set of Umple files that specify relevant components of the metamodel (e.g. for state machines it would specify classes for states, transitions, guards, and so on).

3. Umple Generators

    Code generators in Umple receive parsed and analyzed models and are invoked to generate code for languages like C++, Ruby, Java, Php, as well as for diagrams and other outputs. For example, the UmpleToJava directory contains Umple template code that are used for generating Java code.

4. Build tools

    Build automation tool is an essential element of the software deployment process. Each build tool compiles source code, packages it, and automates unit testing. The Umple build directory has several Ant scripts that assist software developers to compile, run tests, and to update the local version of UmpleOnline before contributing to the mainstream Umple branch in Github. There are also instructions for building in Gradle, to which Umple is being transitioned.

When building, the Umple Parser is compiled first. Then a parallel compilation occurs for master files in cruise.umple/src including Umple generators. The master file coordinates the order of compilation of all files within a directory.

5. Umple Tools

   There are several directories in the Umple repository that facilitate using Umple in different environments including an Eclipse plugin, the website UmpleOnline, and Docker images.

7.3 The Refactoring Process

We began the refactoring process by crafting a shared outline document that expresses and discusses the main subjects of the intended refactoring. The document includes refactoring rationale, what constitutes a feature (feature allocation), what to refactor first, discussion of difficulties and available solutions. The document shapes some sections in this chapter.

7.3.1 Scope of Refactoring

In this case study, the scope of refactoring covers modeling capabilities and Umple generators. In addition, there are a few modifications required in the build files because of the parallel compilation for the master files in "cruise.umple" and for master files of each generator directory. We have elected to limit the coverage by not currently refactoring UmpleOnline, the Eclipse plug-in, and other elements that are not associated with the core compiler. For future enhancements, deactivating a certain feature in the compiler should make it unavailable in UmpleOnline.

7.3.2 Procedure Followed for Refactoring

This section outlines the procedure that was followed to refactor Umple into a feature-based SPL:

1- Identify potential features.

This procedure aims to extract potential features from Umple. Section 7.3.3 discusses feature allocation and feature modeling in further detail.

2- Select features to refactor.

We pick the easiest features first, and then work towards the more complex ones. We started with state machines as a potential first feature to refactor, but it proved too complex

for our first effort. We switched to refactor Umple generators (e.g. Java, PhP, Graphviz class diagrams) as they are self-contained and easier to slice. Then after slicing all generators as features, we began extracting features at the modeling level, including state machines.

3- **Perform feature-based refactoring**

For a selected feature, the first task is to split large files into smaller ones with use statements. Each use statement brings refactored content back together during compilation. A directory is created to contain each feature's refactored files. A mixset is then created to contain code related to each feature with inline mixsets also added to enclose cases of fine-grained variability. Mixsets with large-grained variability such as the results of class-level refactoring are moved into corresponding feature files. For a few cases, inline mixsets are refactored into compositional mixsets. Figure 30 demonstrates an example taken from the filter feature. Finally, a use statement is introduced in the master Umple file for each newly created feature mixset.

File: umple/cruise.umple/src/UmpleInternalParser_Code.ump

```
class UmpleInternalParser
{
…
public UmpleInternalParser()
{
  this("UmpleInternalParser", new UmpleModel(null), new
RuleBasedParser());
}

private void init()
{

  if(model.getUmpleFile() != null) {
    setFilename(model.getUmpleFile().getFileName());
    setRootToken(reset()); // Makes sure the root token position has
the filename
  }
  …
  // Set up needed parser actions
   parser.addParserAction("useStatement", new
UseStatementParserAction() );
  parser.setLinkedFileHandler( new UmpleLinkedFileHandler() );
  parser.setAnalyzerGenerator( new
UmpleAnalyzerGeneratorHandler() );
  parser.addGrammarFile("/filter/umple_filter.grammar");
  parser.addGrammarFile("/umple_exceptions.grammar");
  parser.addGrammarFile("/use.grammar");
  }
…
}
```

File: umple/cruise.umple/src/UmpleInternalParser_Code.ump

```
class UmpleInternalParser
{
…
public UmpleInternalParser()
{
  this("UmpleInternalParser", new UmpleModel(null), new
RuleBasedParser());
}
private void init()
{

  if(model.getUmpleFile() != null) {
    setFilename(model.getUmpleFile().getFileName());
    setRootToken(reset()); // Makes sure the root token position has
the filename
  }
  …
  // Set up needed parser actions
  parser.addParserAction("useStatement", new
UseStatementParserAction() );
  parser.setLinkedFileHandler( new UmpleLinkedFileHandler() );
  parser.setAnalyzerGenerator( new
UmpleAnalyzerGeneratorHandler() );
  mixset Filter {
    parser.addGrammarFile("/filter/umple_filter.grammar");
  }
  parser.addGrammarFile("/umple_exceptions.grammar");
  parser.addGrammarFile("/use.grammar");
  }
…
}
```

File: umple/cruise.umple/src/UmpleInternalParser_Code.ump

```
class UmpleInternalParser
{
  …
}

mixset Filter {
  use filter/UmpleInternalParser_CodeFilter.ump;
}
```

File: umple/cruise.umple/src/filter/UmpleInternalParser_CodeFilter.ump

```
class UmpleInternalParser
{
  after init {
    parser.addGrammarFile("/filter/umple_filter.grammar");
  }
    …
}
```

Figure 30: The blue highlighted code of the top-left rectangle is refactored into an inline mixset and then into a compositional mixset.

4- **Test the refactored code**

Three kinds of tests are considered to ensure that refactored code results in an SPL that avoids errors. First, the refactored code must pass all Umple tests (6000+) when all features are included. Second, each feature is syntactically checked alone (without including other features unless there are feature dependencies), and then each pair of features, when used together, is also checked. This to verify the features and pairs do not have type errors and method reference errors. Third, the unit tests that are related to each feature must run with no failures.

5- **Iterate over the above steps.**

During each step, we documented our rationale, experience, and challenges. At the beginning, we spent around two weeks to lay out the general procedure, address challenges, and to perform quick evaluation of alternative approaches. Then the refactoring tasks became smoother since many problems were anticipated and their solutions had been already discussed.

7.3.3 Feature Identification and Modeling

Feature diagrams are often documented in SPL requirement documents. However, because Umple was not initially an SPL, the question of how to refactor the current version of Umple into feature-based SPL necessitates a feature list somewhere in Umple documentation. This was not true for our case; therefore, it was vital to construct a feature list, or more precisely, a feature model for Umple. We relied on three main sources to identify potential features in Umple:

1. **Exploring the Umple website**

   Specifically, the user manual page has rich content describing Umple. As Figure 31 demonstrates, most menu headings can map to features.

Figure 31: Umple menu headings.

2. **Considering students' theses**

   Members of the research group who completed PhD or master theses usually implement practical aspects of their work in Umple. These theses often result in large features such as state machines, traits and tracing.

3. **Mining the source code**

   We relied on the Umple names of directories, files, classes, associations and methods to help identifying some features.

We used the three sources accompanied with brainstorming and reflection on the results to form a feature model in an iterative manner. Figure 32 shows the final version of the Umple feature model. The generators can be classified into generic (displayed in yellow) and specialized generators (displayed as light blue). The Java generator is an example for a generic generator as it generates code for all models. The Event sequence generator is a specialized generator as it generates output only from a specific part of the model: state machine events. Complete dependencies among generators and model level features are not shown in the diagram but can be found in the umpleSPL repository (Algablan, 2021).

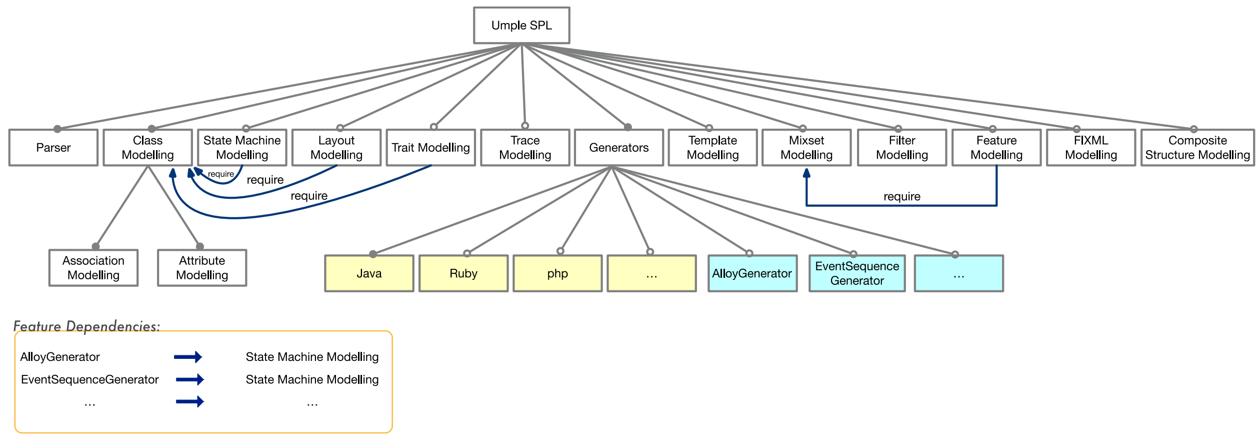

Figure 32: Umple feature model.

7.3.4 Feature Catalogue

The table below shows a description for the features appearing in Figure 32.

Table 8: Description of refactored features in Umple.

Feature	Description
Parser	The parser of Umple is the base, or root, of the Umple SPL. All features depend on the parser feature.
Class Modeling	Modeling of UML classes in Umple. Class modeling includes the Association and Attribute features. These two features can be changed to be optional, but they are treated as required features to simplify the refactoring process.
State Machine Modeling	State machine modeling in Umple includes rich functionality such as transition actions, guards, nested states, timed transitions, queued states, and pooled states. Each of these could be considered as a sub feature. State machines are one of the core features of Umple but could be left out if it was desired to create a very simply tool for code generation from classes.
Layout Modeling	Layout modeling helps in tweaking the layout of diagrams textually. Layout modeling elements are injected into classes and heavily used in UmpleOnline. For simply compiling code, layout modeling is not needed as a feature.
Trait Modeling	A trait models a group of class content items that serves as building blocks for classes. Mainly, through methods, state machines, and other modeling elements used for modeling behavior. A useful and slightly simpler compiler could be created without traits, but the compiler itself uses traits so Umple would not be able to compile itself.
Trace Modeling	Trace modeling allows developers to specify tracing at the model level. For example, a class attribute, or a state in a state machine, can be traced without adding print statements or some other special debug statement to code. This feature is strictly optional.
Generators	There are many generators in Umple. They can be classified into generic and specialized generators, with the specialized ones generating output only for certain other features.
Template Modeling	Umple templates assist in generating string output (e.g. specifying languages).
Filter Modeling	A filter directive in Umple selects only a certain part of a model and ignore the rest. It allows creation of different diagrams from the same model. It is one of Umple's mechanisms for separation of concerns.
FIXML Modeling	FIXML modeling accepts financial data representation expressed in FIXML and generates object-oriented code. It is a domain-specific language that was added to Umple to improve its capabilities.

Composite Structure Modeling	Composite structure models the communication of various instances of classes, called parts, at run time. It is a strictly optional feature.
Mixset Modeling	Mixset modeling contains models and code which are required to implement Mixset in Umple. It is an optional feature.
Feature Modeling	Feature modeling is an optional feature that empowers mixsets as modeling constructs with feature modeling. It is an optional feature but requires the presence of the mixset modeling feature.

7.4 Outcomes

Although it is possible to achieve fully compositional representation of features in the Umple SPL, as we will illustrate in section 7.4.3, the variability of the current master branch of Umple is achieved using combination of compositional and annotative approaches. Based on our assessment, transforming some annotative fragments in Umple SPL into fully compositional counterparts would impair comprehensibility – especially when feature fragments are nested in a non-trivial way. Strüber et al. pointed to similar observation for tasks that require overview over all variants (Strüber et al., 2020). Therefore, we adapted a combined variability approach for the master branch so it raises code understandability although it does not fulfill full feature separation.

After refactoring Umple, there are ten new directories in the "cruise.umple/src" directory, which contains all model level capabilities in Umple. In terms of mixsets, Table 9 shows the number of fragments of each mixset at the modeling level.

Table 9: Total number of fragments of each feature mixset.

n	Mixset Name	Fragment Count
1	StateMachine	75
2	Trait	49
3	Mixset	15
4	Trace	14
5	Structure	6
6	FeatureModel	5
7	Template	6
8	Layout	6
9	Filter	5
10	FIXML	3

These mixsets reflect features that are standalone and can form up to 2^n different combinations, or SPL variants. The table indicates that StateMachine and Trait have higher counts

of fragments. This is due to the magnitude of state machines and traits as features in Umple and their interaction with other features.

There are mixsets which have not been completely separated as features but identify a concern or are used to manage feature interaction code such as the Template_Structure and Structure_StateMachine mixsets. They contain some shared code that are necessary for the two features when they are selected together. If one or all features involved in forming feature interaction mixsets are not selected, interaction mixsets should not be available for selection. Table 10 shows mixsets that have not been separated as features.

Table 10: Total number of fragments of each non-feature mixset.

n	Mixset Name	Fragment Count
1	Class	6
2	Association	46
3	AspectInjection	7
4	Constraint	5
5	Template_Structure	1
6	Structure_StateMachine	2

The code blow shows the content of Structure_StateMachine mixset.

```
mixset Structure_StateMachine {
  class ConstraintPort
  {
    String getName()
    {
     return port==null?null:port.getName();
    }
    public String getType(){ return "port"; }
  }
}
```

7.4.1 Granularity of Feature Mixsets

Fragments of mixsets forming features can be classified into coarse-grained variability and fine-grained variability. Coarse-grained variability includes definition of classes, fields, methods, state machines and associations as well as file inclusion. Fine-grained variability incorporates mixsets inside methods, and mixset fragments which contain aspect injection. Table 11 classifies mixsets fragments according to their granularity.

Table 11: Fragment type of feature mixsets.

n	Mixset Name	Coarse grained variability	Fine-grained variability
1	StateMachine	36	39
2	Trait	7	42
3	Mixset	13	2
4	Trace	6	8
5	Structure	2	4
6	FeatureModel	3	3
7	Template	3	3
8	Layout	1	5
9	Filter	2	3
10	FIXML	1	2

7.4.2 A Deep Look at Fragments

Table 12 shows more details about the content of mixset fragments. Although, a mixset fragment may contain different types of variability such as class definitions, file inclusion (use statements), and associations, each fragment has only single type of variability in this case study. Therefore, each fragment increments only a column's count.

Table 12: Fragment details

n	Mixset Name	Coarse grained variability				Fine grained variability				Total
		C1: # of file inclusion	C2: # of class def.	C3: # of association def.	C4: # of method def.	F1: # of Statement change – start with if	F2: # of Statement change – contains only method call	F3: # of Statement change not containing F1 and F2	F4: # of aspect injection	
1	StateMachine	6	10	4	12	18	7	13	2	75
2	Trait	2	1	2	3	25	3	12	1	49
3	Mixset	2	0	0	0	5	7	1	0	15
4	Trace	3	1	0	2	5	0	2	1	14
5	Structure	2	0	0	0	1	2	1	0	6
6	FeatureModel	2	0	0	0	0	3	0	0	5
7	Template	2	1	0	0	2	0	1	0	6
8	Layout	1	0	0	0	1	4	0	0	6
9	Filter	2	0	0	0	0	3	0	0	5
10	FIXML	1	0	0	0	0	2	0	0	3
Total		23	13	6	17	57	31	30	4	184
						118				

The following points describe the variability column headings:

- **C1: # of file inclusion:** number of mixset fragments which only contain use statements that includes files. Such as the following Trait fragment:

```
mixset Trait {
  use trait/Umple_Code_Trait.ump;
}
```

- **C2: # of class def. :** number of mixset fragments that introduce new classes.
- **C3: # of association def. :** number of mixset fragments that introduce new associations between classes. For example:

```
class UmpleModel {
  …

  mixset Trait {
    1 -> * UmpleTrait;
  }
  …
```

- **C4: # of method def.:** number of mixset fragments that introduce new methods for classes.
- **F1: # of Statement change – start with if:** number of mixset fragments that introduce statement level change starting with if statement. The if statement usually verifies an entity type, which controls run time variability. The code below shows an example:

```
class  UmpleInternalParser {
  …
  private void analyzedReferencedStateMachine(Token
stateMachineToken, UmpleClassifier uClassifier)
  {
   …
    StateMachine sm = new StateMachine(name);
    mixset Trait {
      if (uClassifier instanceof UmpleTrait){
        sm.setUmpleTrait((UmpleTrait)uClassifier);
      }
    }
  …
}
```

- **F2: # of Statement change – contains only method call:** number of mixset fragments that have only method calls and nothing else. This type of statement could be qualified for AOP or FOP refactoring if they do not evolve in problems specified in Section 5.2. The code below illustrates the idea:

```
void addExtendsTo(Token classifierToken, UmpleClassifier aClassifier,
Map <UmpleClassifier,List <String>> unlinkedExtends, Map
<UmpleClassifier, List<Token>> unlinkedExtendsTokens)
  {
    …
    else if (extendsToken.getValue("gTemplateParameter") !=null ){
        mixset Trait {
          processGTemplateParameterAssignment(extendsToken,
aClassifier,
          extendName);
        }
      }
    …
    }
```

- **F3: # of Statement change:** number of mixset fragments that introduce statement level change not starting with an if statement and not only a method call, but multiple statements.
- **F4: # of aspect injection:** number of mixset fragments that introduce statement using Umple aspect injection. These injections do not contain labels. They utilize before or after keywords.

7.4.3 The Argument for Fully Compositional Fragments

To obtain fully compositional fragments using the mixset approach for the Umple SPL, there is additional work to be done. Mainly, moving coarse grained variability fragments (C1, …, C4) to their corresponding features. All fragments of C1, which denotes file inclusion fragments, may be better placed in a master file. Fine-grained variability fragments (F1, F2, and F3) can be extracted out as aspects with labels, whereas the labels point to arbitrary locations within some method statements. This work originally had to be done manually.

To automate the process for refactoring Umple into FOSP, we implemented a specific generator called AnnotativeToComposition to accomplish these tasks. The generator moves all annotative fragments into proper compositional fragments. For coarse grained variability fragments, the generator gathers all fragments of each mixset and then produces compositional counterparts. The generator automatically generates labeled aspects for all fine-grained variability fragments. Fully automating the refactoring from annotative to compositional mixsets has not been achieved yet, since the generator does not remove mixsets occurring in the original Umple files. We created a forked repository called UmpleSPL which contains the fully compositional feature-oriented SPL of Umple (Algablan, 2021). The master branch in Umple repository, however, consists of combined variability as mentioned earlier.

7.4.4 Feature Model Specification

To form feature models using mixsets, a set of require statements is needed to specify relationships (parent, child, include, etc.) among mixsets. The assumption of compiling all source files through a single master file, through which fragment of mixsets can be grouped together to form a feature model is applicable for certain software systems. However, the current Umple implementation has several master files that are compiled concurrently; each acts as a starting point to include several Umple files.

Because feature modeling fragments in Umple are spread over multiple files that are compiled separately, placing the relationships of feature mixsets in a common configuration file is viable option. Hence, all master files need to access the configuration file (through file inclusion in each master file). The advantage of this option is to delegate activation of mixsets (to select a feature) to any parsed file.

Alternatively, feature selection can be specified as arguments in the build management tool during generation of Umple SPL variants. This option is more appealing because it separates feature modeling specification in a distinct concern with no need to access it from master files. We created a file named "featureModelingConf.ump" as a configuration file to check the validity of the product configuration in the Ant or Gradle build tools. If the configuration is not valid, the build tool will terminate with an error message. The feature selection can be specified in the Ant build file "build.xml" or it can be passed as argument of the Ant command.

7.5 Observation and Challenges

This section discusses the observation and the challenges that we have faced during the process of refactoring Umple into a feature-based SPL.

7.5.1 Feature Identification and Localization

Identification and localization of features in Umple was not a complicated task despite there being no formal feature descriptions from which we could start the SPL refactoring. This is attributed to our knowledge about the domain and the tool, which may not be available in other contexts. The feature localization was mostly based on static analysis: usually looking at the source code (class names, method names, control statements, etc.), comments as code documentation, and grammar and user manual. Therefore, our procedure to feature identification and localization cannot be fully generalized when refactoring other software systems.

Our challenge during feature identification was the diversity of functionality in Umple that can be split as features and whether to split a feature into multiple subfeatures. For example, state machine modeling has many things that could be considered as subfeatures such as guards, nested states, and so on. We decided to limit or work to high level features that are visible to most users of Umple.

7.5.2 Grammar Files

Grammar files in Umple have specific syntax (they are represented in their own separate domain specific language, parsed by the Umple Parser) and they end with the ".grammar" extension. The syntaxes of all Umple modeling features are introduced through grammar files. Elements of the Umple documentation are also generated automatically based on included

grammar files. Umple grammar files contain great overlap through rule referencing. For example, the mixset definition rule is defined in the file “mixset.grammar” and is referenced in multiple grammar files such as those for state machines and associations. When deactivating a feature, the corresponding grammar file will not be parsed and therefore, references to rules in that file are left dangling with no definitions. There are four ways to handle dangling rules that belong to a deactivated feature:

1. The concept of mixsets could be extended to handle Umple grammar files. Therefore, a composition technique is required to apply the merge to grammar files. Deactivation a feature removes all its rules and cleans up their references in other grammar files.
2. The rules related to feature interaction could be hard coded in some “.ump” files instead of being grammar files. In this case, the grammar files would not include rules that relate to feature interaction. Instead, the rule would to be added as pure code through aspect injection.
3. The grammar files could be split into multiple files. This would move the feature interacting rules to files that are called based on feature selection.
4. Leave dangling rules in place even if there is no code to process the resulting tokens, which would just be ignored. This option carries a bad code smell and negatively impacts the design quality. It may cause confusion for developers since Umple will automatically include all grammar files including full syntaxes of deactivated features. So, for example, if the state machine feature was deactivated, a state machine would be tokenized but otherwise totally ignored.

Option 4 has the least implementation effort but reduces design quality and leads to incorrect documentation. Option 3 maintains the documentation integrity by including only grammars of selected SPL features. For this case study, we selected option 4 to speed up and ease the implementation although option 3 remains highly practical.

7.5.3 Test Units

Umple has a very large number of tests that are executed on any commit to the repository, and must pass before changes are applied to the master branch in GitHub. Ant, which is the main build management tool used in Umple, compiles multiple files including all test cases. When everything

passes, it packages the final executable called “umple.jar”. When a feature is deactivated to generate a product from Umple SPL, the build will raise an error due to some failing tests and the packaged jar will not be created.

Therefore, there is a need to either combine features with their tests, or to separate the testing from the build of the Umple executable. The first option entails refactoring a massive number of test files. The second option is more practical and requires a separate build file that allows to pass arguments to select a set of features and then only include tests based on user selection. Option 2 is followed for the compositional feature-oriented SPL of Umple (Algablan, 2021)

7.5.4 Refactoring during Continuous Integration

One of the challenges that we faced is to refactor while Umple is continually changing. Therefore, on some occasions we were forced to carefully schedule changes because other contributors were working on the same set of files. For software following continuous integration, this challenge hinders the refactoring process and may lead to update conflicts that prolong the refactoring duration. The exploratory study of Saidani et al. confirms our finding and pointed to the negative impact on the refactoring frequency due to CI adoption (Saidani et al., 2021).

7.5.5 GitHub Logs to Identify Features

Particular to Umple, specific contributors are often responsible to implement certain features. Since Umple is hosted in Github, the log command offered by the git CLI can be used to show all modifications written by a certain user. We found that the git command with the path argument “git log -p --author=<author_name>” is very useful to list all additions that a GitHub user has contributed to a certain branch. The log command shows detailed information such as all commits, deleted code, date, and so on. To narrow the search result, the grep command (which is installed on most Mac and Linux computers by default while Windows has other equivalent software), is used to filter the text by including only results that begin with “+++” (which refers to the file name) and with “+” (which contains the newly modified code). Figure 33 demonstrates a fragment of all additions that opeyemiAdesina has contributed to the master branch of Umple. The log result can be used to assist the SPL refactoring because most of opeyemiAdesina’s contributions will map into the Alloy and NuXmV generators.

```
abdulaziz@Abdulazizs-MacBook-Pro umple % git log -p --author=opeyemiAdesina | grep "+++\|+"
+++ b/umpleonline/testsuite/spec/load_examples_helper.rb
@@ -75,7 +75,8 @@ STATE_EXAMPLES = {
+  "TrafficLightsB.ump" => ["clust6", "node30", "edge24"],
+  "HomeHeater.ump" => ["edge23", "node37"]
+++ b/cruise.umple/src/Generator_CodeGvStateDiagram.ump
@@ -27,8 +27,8 @@ class GvStateDiagramGenerator
+  Boolean showTransitionLabels = false;
+  Boolean showGuardLabels = false;
@@ -71,8 +71,8 @@ digraph "<<=filename>>" {
+    showTransitionLabels = hasSuboption("showtransitionlabels");
+    showGuardLabels = hasSuboption("showguardlabels");
@@ -388,8 +388,8 @@ node [shape = circle, fixedsize = true, width=.3];
+        //showTransitionLabels = true;
+        if ( !showTransitionLabels ) {
@@ -433,8 +433,8 @@ node [shape = circle, fixedsize = true, width=.3];
+          //showGuardLabels = true;
+          if ( !showGuardLabels ) {
+++ b/umpleonline/scripts/umple_action.js
@@ -178,6 +178,14 @@ Action.clicked = function(event)
+  else if (action == "ToggleTransitionLabels")
+  {
+    Action.toggleTransitionLabels();
+  }
+  else if (action == "ToggleGuardLabels")
+  {
+    Action.toggleGuardLabels();
+  }
@@ -1505,6 +1513,8 @@ Action.updateUmpleDiagramForce = function(forceUpdate)
     if(!Page.showActions) language=language+".hideactions";
+    if(Page.showTransitionLabels) language=language+".showtransitionlabels";
+    if(Page.showGuardLabels) language=language+".showguardlabels";
@@ -1724,6 +1734,18 @@ Action.toggleActions = function()
+Action.toggleTransitionLabels = function()
+{
+  Page.showTransitionLabels = !Page.showTransitionLabels;
+  Action.redrawDiagram()
+}
+
+Action.toggleGuardLabels = function()
+{
+  Page.showGuardLabels = !Page.showGuardLabels;
+  Action.redrawDiagram()
+}
```

Figure 33:Git log for the user "opeyemiAdesina", who implemented some functionality related to alloy and NuXmV, generator.

The log result above displays only the added code but require extra work to identify the exact location where the new code is added within a method or a class. As future work, we would like to explore the visibility of using GitHub logs to help to identify SPL features based on users' contributions.

7.6 Summary

In this chapter, we have extracted a featured-oriented SPL consisting of ten main features from the Umple MDD tool. The granularity of variability for the features considered spans both models and native embedded code. We offer two SPL versions: one version using combined variability and other using strictly compositional variability. In our case study, the overhead to accomplish fully compositional SPL was high and required several modifications in parsers, languages, and tools. In addition, the comprehensibility of the code was reduced in a few places due to feature deallocation, since some feature fragments, which are involved in non-trivial nesting, are difficult to understand by looking solely at their code. On the other hand, a fully compositional Umple SPL

offers the ability to search for all aspects of a feature by looking for the relevant mixsets, better organization of the code for ongoing maintenance, direct way to factor out features that are in alpha or beta development, the ability to deliver certain features standalone, and less transformation effort is required when features are transferred from an existing SPL into other SPLs.

Chapter 8 Applying Mixsets to the Rover Control Challenge Problem

In this chapter we present a case study of the use of mixsets in Umple. This case study was published at the MDETools 2018 workshop (Lethbridge and Algablan, 2018a). The key idea in this case study is the use of mixsets to switch different control algorithms as related files, and their dependencies on or off. It demonstrates the viability of mixsets to represent SPL variants whether they contain pure code or they extend modeling constructs of a common SPL base.

The problem, as specified by the MDETools challenge problem statement (*MDETools 2018 Challenge Problem*, 2018), involves arranging for a rover to follow a randomly moving leader rover in a simulated environment. The challenge is to create a model that can generate code whereby the follower stays neither too far (15 simulation distance units by default) from the leader nor too near (12 units) to it, and the follower turns appropriately to follow the leader. Follower speed and direction primarily has to be controlled by independently varying power to left- or right-side sets of wheels; braking is also available.

Our solution has a class model consisting of the two types of rovers with various attributes and methods. There is also a separate controller class. We approached the dynamic part of the problem – autonomous following – with two distinct models, each of which can be activated using a different mixset.

The first we call 'frequency modulation'. It involves applying pulses of 100% forward thrust to both sides, as well as turning pulses (one side 100% forward, the other side 100% reverse). The pulses vary in duration (and hence frequency), with longer pulses being needed when greater corrective action is needed. This mimics the way a human might manually operate the following rover; indeed, it was inspired by spending time using a manual interface provided by the workshop organizers.

The second approach we call 'amplitude modulation'. It uses state machine models to adjusts the power of left side and right side wheels at regular intervals. The amount of power applied is

varied depending on the need for corrective action. Similarly, the greater the need for turning, the greater the difference between power applied to left and right wheels.

While it was easy to tune the system to give good following results for the frequency modulation approach, we were not able to effectively tune the amplitude modulation approach.

8.1 Method Used to Develop the Solutions

We followed an agile approach to developing our solutions. Each step is outlined in this section. We have documented these steps to help others understand experiences in using Umple, and to allow comparison of tools. The resulting code, at various stages has been placed in a Github repository (Lethbridge, 2018). Releases of the jars have been created so the reader can check out and run the simulation at any stage.

8.2 Version 1: Converting to Umple and Getting a Basic Controller Working

We created the first version of our system in the following three sub-steps:

Operationalizing the Provided Manual Program

First, we ensured we could run the provided (manual-controller) version of the challenge problem (*MDETools 2018 Challenge Problem*, 2018). This includes three executables: 1) the rover simulator, a visual environment made with Unity (*Unity*, 2021), 2) the Observer environment, a Java application; and 3) the provided RoverController that allows the user to manually use keyboard keys to give commands to the follower. These three communicate using TCP/IP. The challenge problem requires replacement of only the third component with an autonomous controller.

The provided application also comes with a configuration file allowing changing various parameters, such as how erratic the leader should be in terms of speed and changes of direction. Although we experimented informally with adjusting the parameters, we elected to leave them exactly as they were provided for the remainder of our work. This will allow other models to be compared to ours more easily.

After getting the manual controller working, we spent approximately 30 minutes controlling the follower manually in order to better understand the behavior of the leader and strategize about how we would model an autonomous follower.

We committed the original version of the code to our repository (Lethbridge, 2018).

Umplification of the Provided Manual Controller

Umplification (Lethbridge et al., 2010) is a key process recommended for many adopters of Umple that are faced with an existing code base. It involves taking code written in a language such as Java and converting it into Umple. The idea is to allow developers of legacy software to gradually and painlessly convert their code so that it becomes model-based, all the while making sure tests pass at every stage.

The provided manual controller code consists of three files. SocketCommunicator.java, DriveCommands.java and RoverController.java. We elected to initially only umplify the third of these and leave the others as external libraries. The umplification of RoverController.java was a one-minute process, since the Umple compiler accepts Java syntax almost as-is. The result was a file called RoverController.ump. We simply had to convert Java 'import' statements into Umple 'depend' statements and declare the other two files as external Java classes using Umple's 'external' keyword. We then verified that the resulting program (compiled now by Umple) behaved identically to the provided program.

It is worth noting that we used the command-line version of the Umple compiler for this work, as well as UmpleOnline for working with diagrammatic views of the models.

Stripping the Manual Control and Adding Dumb Controller Logic

The next step in our agile conversion was to strip all the manual control logic, including the user interface that accepted keystrokes, from the originally-provided system. Control of the rover was replaced by a single short method called 'follow()' that simply drives straight, makes a turn and drives straight again. In other words, it does not actually pay any attention to the leader, but was written to ensure that the system still could be compiled and executed.

It is worth noting at this stage that although the RoverController class is written in Umple, it is still not at this stage using any of Umple's modeling capabilities. The follow() method is just plain Java. One of the key features of Umple is that plain Java can be embedded in Umple. However, a diagram can nonetheless be drawn of the system by Umple, and this appears in Figure 34.

The end-result of this step was committed to the repository (Lethbridge, 2018) and tagged as V1. A release of the V1 jar is also available.

8.3 Version 2: The 'Frequency Modulation' Solution

Our next step was to start altering the Umple model to enable autonomous control. In the RoverController.ump file. We defined an abstract Rover class that has attributes and methods common to the leader and follower. We added subclasses for the latter two. Implementations of methods allow instances of these classes to be updated with current information about their position by querying the connected servers.

Of particular interest is a method that computes the bearing from the follower to the leader, enabling it to know what course it must optimally follow. This is line 151 in RoverController.ump and is as follows:

```
1    bearingToLeader = Math.toDegrees(
2      Math.atan2(
3        leader.getXPosQuick() - getXPosQuick(),
4        leader.getYPosQuick() - getYPosQuick()
5      )
6    );
```

Central to the following approaches is turning the rover such that the bearingToLeader is within 5 degrees of the follower's compass heading.

A class diagram generated by Umple is presented as Figure 35. The following is a sample of Umple text describing the Rover class.

```
1    class Rover {
2      abstract;
3      String nameForCommanding;
4      SocketCommunicator querySocket;
5      long lastUpdateTime = 0L;
6      * -> 1 RoverController controller; // association
7      Double xPos = 0.0;
8      before getXPos {updatePosition();} // get from server
9      Double xPosQuick = {xPos}; // get from cache
10     Double yPos = 0.0;
11     before getYPos {updatePosition();}
12     Double yPosQuick = {yPos};
13   }
```

Some features of the above code are worth pointing out:

- Line 2 has the keyword 'abstract' this is Umple's syntax to declare a class as abstract. We call such keywords 'stereotypes' following UML conventions.
- Lines 3,4,5,7 and 10 define various attributes. They are like variable declarations but have more sophisticated semantics, aligning with UML conventions, but also allowing 'aspects' to systematically adjust their generated code.
- Lines 8 and 11 are simple aspects that ensure that before accessing the data, it is updated from the server.
- Lines 9 and 12 allow bypassing this update by obtaining the cached position values, if the developer knows it has recently been performed.

After some fine tuning, we were able to ensure that follower successfully follows the leader in all circumstances, but it errs on the side of following too far (following too close can lead to crashes). Table 13 shows that version 2 of the follower on average manages to stay in the required zone 53.2 percent of the time when tested according to the provided testing module UnityObserver.jar. Version 2 of the RoverController jar is available in our repository.

DriveCommands

SocketCommunicator

RoverController

main(String[]) : void
+follow() : void
sleepms(int) : void
readFile(String) : java.util.List<String>

Figure 34: Umple-generated diagram of version 1 of the solution to the challenge problem with a basic dumb follow method

Table 13. Test results for FM versions 2 and 2a (10 tries each)

	Version 2, 500ms pulses min				Version 2a, 250ms pulses min		
	Percent in zone	Percent too close	Percent too far		Percent in zone	Percent too close	Percent too far
Mean	53.2	6.6	40.3		65.5	6.3	28.1
Std. Dev.	21.0	4.1	20.0		14.4	4.0	14.4
Max	82.9	13.8	67.1		85.8	10.4	57.9
Min	27.5	0.0	14.6		42.1	0.0	10.0

8.4 Version 2a: Improving Model Quality and Exploring Parameter Variation.

Our next step was to refactor and improve the Umple model. Unlike Java and many other languages, Umple does not force the user to keep one class per file. In fact, a typical Umple model would have a several-to-several relationship between files and class fragments. Files in Umple should be seen as units of human-understandable functionality. Umple's ability to distribute class fragments among multiple files might hypothetically make it hard to find needed code or model elements. In practice, however, we have found this is not an issue: Javadoc generated from Umple helps the user locate the needed elements, as do diagrams generated by Umple and simple searching.

Figure 36 presents the File model for distribution of functionality after refactoring. We left the pure Umple class model in RoverController.ump. We moved utility logic such as code that obtains data from servers and calculates values such as bearings to RoverController_code-updateData.ump, and the follow() method to RoverController_FMfollow.ump. We also extracted hard-coded parameters needed for the control algorithm to RoverController_FMparameterSets.ump.

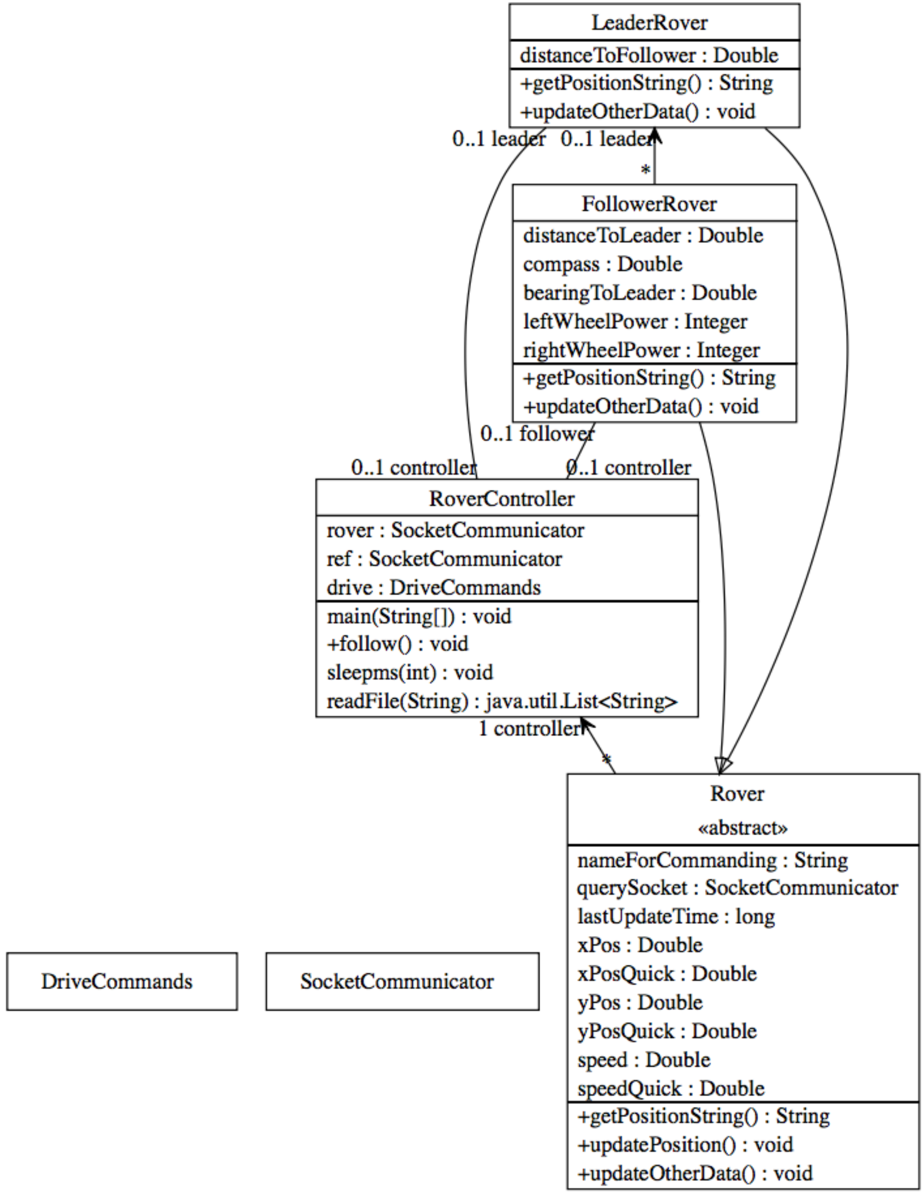

Figure 35: Class diagram from version 2 with a moderately intelligent follow method

As a final step to create version 2a of our controller, we experimented with changing values of key algorithm parameters. In particular, we had originally decided to allow the lower bound for the time-period of power pulses, and also the coasting period between pulses, to be 500 ms (mimicking the time a human might spend pressing and releasing a key to cause acceleration). To make the controller more responsive we reduced the this to 250ms. Table 1 shows this resulted in much better in-zone following performance. We tried to reduce these parameters to be 125ms, or other values between 125 and 250ms, however we started to run into problems: Messages sent to the simulation and the RoverController were sometimes taking more than 125ms for the round trip

(message sends plus processing), and this led to erratic behavior. We settled on 250ms as the shortest time for these parameters that resulted in stable behaviour.

- RoverController.ump Class Model: RoverController, Rover, LeaderRover, Follower Rover
  - Uses: DriveCommands.ump: Originally provided umplified code
  - Uses: RoverController_code-updateData.ump: Updates Data
  - Uses: RoverController_code-general.ump: Utilities
  - Uses if mixset FM: RoverController_FMfollow.ump: Loop follow method
    - Uses RoverController_FMparameterSets.ump
  - Uses if mixset AM: RoverController_AMfollow.ump: State machine
    - Uses RoverController_AMparameterSets.ump

Figure 36: File and feature model.

8.5 Version 3: The 'Amplitude Modulation' Solution

For version 3, we attempted to create a totally different algorithm. In version 2, we pulse full 100% power for varying amounts of time, with full 100% or negative100% power application for direction changes if needed after each pulse. In version three we tried instead adjusting power at values less than 100% at regular intervals. We also attempted to turn by varying the power to left and right wheels – with greater difference when more turning is needed.

This was achieved by using a pair of Umple state machines, to control distance and direction. The diagram for the turning state machine appears in Figure 37, and the Umple code appears below.

```
1   class RoverController {
2     directionControl {
3       directionJustRight {
4         // Set wheels to be equal
5         entry / {setLrDifference(0);}
6         adjust [Math.abs(savedOffHeading) > MIN_TURN_THRESHOLD] ->
7            directionNeedsAdjustment;
8       }
9       directionNeedsAdjustment {
10        // Set wheels to be different
```

```
11         entry / {
12           setLrDifference(-(int)savedOffHeading *
13             DIRECTION_MULTIPLIER);
14         }
15
16         // Direction is OK now
17         adjust [Math.abs(savedOffHeading) <= MIN_TURN_THRESHOLD]
18            -> directionJustRight;
19
20         // re-check direction every cycle
21         adjust [Math.abs(savedOffHeading) > MIN_TURN_THRESHOLD] ->
22            directionNeedsAdjustment;
23       }
24     }
25   }
```

Full details are in the repository. Textual state machines in Umple can be identified as a state machine name followed by a brace, so line 2 of the above marks the start of a state machine. Line 3 marks the start of a state. Line 5 indicates an action to take on entry to the state. Line 6 specifies a guarded transition, with the guard in square brackets, and the target state appearing after the -> symbol. The words in all-capitals in the state machine are tuning parameters that can be adjusted in the file RoverController_AMparameterSets.ump.

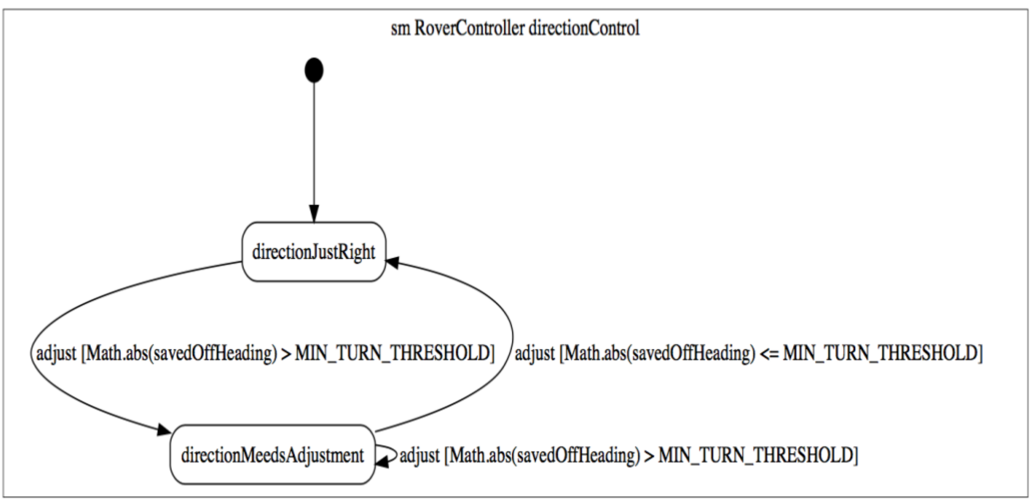

Figure 37: Direction control state machine as a diagram

Although we spent much time trying to fine-tune the parameters for this amplitude modulation approach, we were never able to get it to perform as well as the frequency modulation approach. It seems that in order to keep up with the leader, the follower must apply full power most of the time, making the FM approach better.

We have nonetheless left the AM approach in the code base, but have switched it off by commenting out the mixset (feature) that activates it in the feature model. A user running version 3 of our system will achieve the same results as for version 2a unless they uncomment RoverController.ump line 22 and comment out line 21. Results of executing the Amplitude Modulation solution are found in Table 14.

Table 14. Test results for AM version 3 (10 tries)

	Version 2, 500ms pulses min		
	Percent in zone	Percent too close	Percent too far
Mean	39.9	7.0	53.1
Std. Dev.	15.0	2.3	15.5
Max	71.3	9.6	72.5
Min	21.3	3.3	22.1

8.6 *Conclusions and Key Lessons Learned*

Umple proved easy to use for this exercise. The total number of lines of Umple model/code written were 732, with the generated Java being about double that. We did not find any bugs in Umple itself in this project, and were able to complete it in about a day.

Some of the features of Umple that proved particularly useful were the following:

- The mixset feature, central to this work, allowed us to vary both model elements and algorithmic code to create two separate 'product family' versions that could easily be alternated between for experiments.
- The Umple compiler's error messages were precise and helped us to solve syntax and semantics errors very rapidly. In fact, from the time of completion of writing any significant

enhancement to the model/code to the time when we were able to get the next version of the running system functioning, was usually just a matter of minutes. The vast majority of the development time was spent strategizing about the model, tuning the model parameters, and running the simulation. Debugging of Umple or embedded Java was less than 5% of total time spent.

- It was nice to be able to see both textual and graphical views of the model, as they were able to serve as quality checks on each other.
- Embedding of algorithmic code (as needed for various calculations) in the model helped simplify development. These were added as derived attributes, state machine actions, and ordinary methods.
- Injection of code into generated methods, such as to load data from the server and send control messages, using Umple's 'before' and 'after' constructs, was very useful and effective.

Regarding the two approaches to following, it proved easy to tune the first (frequency modulation) approach and get fairly good results. We suspect this is simply because the follower needs to apply full power most of the time in order to keep up with the server.

As future work it would be nice to instrument the simulation so the parameters can be varied programmatically: Genetic algorithms and machine learning could then be applied to optimize the parameter settings. Undoubtedly many variations of the model could also be tried. Additionally, a variety of additional mixsets and nested mixsets could be added with further variations in the algorithms.

Chapter 9 Evaluation

This chapter evaluates mixsets as a combined SPL variability approach. The first section compares mixsets against annotative and compositional SPL technologies that either have been used in practice or featured in the SPL literature. The second section evaluates mixsets in terms of tool support. The last section discusses the limitations of mixsets as a concept, as well as the approach to implement and evaluate mixsets as presented in this thesis.

9.1 Comparing Mixsets Against Related SPL Techniques

To evaluate our approach, we will use criteria discussed in SPL literature and used for comparing and assessing some SPL techniques (Apel et al., 2013; Kästner and Apel, 2008a; Krüger et al., 2016). We will focus on the following criteria: preplanning, adoptability, separation of concerns, traceability, information hiding, granularity, uniformity, and language independence. Table 15 summarizes our assessment of the approaches; we will discuss the rationale for each of these assessments in the following sections.

Table 15: A comparison of our approach to CPP, FOP, AFM, and FeatureCoPP.

	CPP	FOP	AFM	FeatureCoPP	Mixsets
Low Preplanning Effort	good	medium	medium	good	good
Adoptability	excellent	poor	poor	medium	good
Separation of Concerns	none	good	excellent	medium	excellent
Traceability	medium	excellent	excellent	medium	good
Information Hiding	none	good	medium	poor	poor
Granularity	excellent	poor	medium	excellent	good
Uniformity	excellent	excellent	medium	good	good
Language Independence	good	medium	poor	good	medium

Support level: ○ = none , ◔ = poor, ◑ = medium , ◕ = good , and ● = excellent.

We apply the criteria to mixsets and also to four other comparators that cover wide range of variability mechanisms. CPP is a standard annotative approach. FOP is a representative

compositional approach, which can be implemented at the top of a programming language such as Jak extension for Java programs (Prehofer, 2006). AFM is a compositional approach that combines AOP and FOP (Apel et al., 2008). AFM does not have a specific implementation, however, FeatureIDE integrates Jak and AspectJ to support AFM for Java (Thüm et al., 2014). FeatureCPP is a combined approach which allows annotation and composition (Krüger et al., 2016).

We have only used the 'Excellent' rating in the table if the tool meets the criterion as effectively as the best tools, and without any significant drawbacks. Good means the tool is as good as the best tool in most contexts but has minor weaknesses that would only hinder its use in edge cases. Medium means the tool meets the criterion but has small but notable limitations. Poor means there are significant limitations.

It should be noted that mixsets can be used entirely compositionally, keeping the feature code separate from the base code, or entirely annotationally, just tagging feature-specific code using inline mixsets, or a custom blend of composition and annotation. The developer has a choice.

9.1.1 Preplanning Effort

Because a key objective of SPL is to maximize code reuse, some amount of preplanning is a prerequisite no matter what SPL technique is used. Preplanning means determining which variants should exist and what parts of the code should be reusable, as well as ensuring reuse of these parts is straightforward. A design method has high preplanning effort if the variability and reusability decisions *must all be made early in design* to avoid extensive rework, whereas a method has low preplanning effort if variants can be specified, or parts of the codebase can be made reusable, in late stages of evolution, without extensive rework.

Annotation techniques tend to be lightweight since it is possible to introduce new annotations to existing code to mark variants with little overhead or impact on the way the code worked originally. For example, a feature like Disagreement Warning can be easily added to a variant using CPP. It requires insertion of "#ifdef hasDisagreementWarning" and its code at the desired location to raise the warning. Then the feature can be switched on by adding the code "#define hasDisagreementWarning" at the beginning of the file including #ifdef, or in the configuration file. Inline mixsets can be used in a similar way. In addition, compositional mixsets can be used as follows : A label *disagreementWarning:* could be added at the desired location in the method, and

then right after the method a code block *mixset hasDisagreementWarning {after disagreementWarning: methodname {codeToRaiseWarning}}*..

For classic compositional approaches, careful analysis of common patterns of variation needs to be done so that the relevant variant or feature files can be set up, and so that some form of matching can accurately inject (compose) the feature or variant into the base code. The initial phase of designing a common base for a software product line tends to require considerable preplanning effort.

Preplanning includes also anticipating and handling feature interaction. Feature interaction describes the emergent behavior that occurs in when an SPL variant is specified to contain a combination of two or more features that have subtle conflicts or influence how each other behaves. Feature interaction is a broad concept; it can be desirable, but it might be unintended. Interacting features pose a challenge for all SPL techniques; they should be anticipated and managed carefully, especially for large features. There is a significant body of research tackling feature interaction from the composition perspective. For example, Prehofer introduced distinct modules called lifters that explicitly manage interacting feature pairs (Prehofer, 2006).

Mixsets perform well in this criterion in a similar way to CPP and FeatureCPP. Changes to the code are allowed without high effort for adopting new changes. Inline mixsets in our approach resemble code annotations as in CPP. An inline mixset is a syntactic sugar for a virtual decomposed fragment. FOP and AFM are stricter regarding representation of changes, in that separation of feature fragments is required during all stages of SPL development. In addition, mixsets have the advantage of nesting mixsets and utilizing non-feature mixsets to manage feature interaction. Compositional mixsets can handle feature interaction in separate modules as FOP does. This kind of separation is not straightforward in CPP and other annotative approaches. This is because a mixset fragment is reachable from other fragments in the same mixset. For example, a newly added fragment can navigate the structural hierarchy in which a previous fragment was placed to add extra code that manages feature interaction.

We did not rate any approach as ‘excellent’ in Table 15 for preplanning, since even approaches that are annotational requires work, such as good planning of naming conventions, to ensure understandability remains high.

9.1.2 Adoptability

This criterion is mentioned in (Krüger et al., 2016) as "adoption", however, the term "adoption" usually refers to extent of *existing* use of a technique in some domains. We prefer the term "adoptability" because it accurately refers the effort required for software developers to adjust, or change, their routine and existing tools in order to incorporate the variability technique. Ideally, the variability technique should be easy to learn, free, open source and adaptable to any toolchain, as well as cross-platform.

CPP is a well-known mechanism for C/C++ and can be used on top of many programming languages. FOP and AFM require a particular IDE (Eclipse) and require adaptation of composition to separate SPL features. Although there is no tool support for FeatureCoPP, the idea of invoking hook methods using annotations seems appealing, however, it has the challenges of classical compositional approaches. The availability of inline mixsets offers smooth adaption to the combined variability approach.

Mixsets are part of Umple, but the approach still is IDE agnostic. Umple has been designed to work in conjunction with arbitrary text editors, version control tools and build tools. It has command-line, Eclipse plugin, and web-based tool support, therefore, it can be used in almost any toolchain.

Adoption of composition approaches (including FOP and AFM) is quite low compared to CPP. Mixsets is a new technology, so it cannot be judged on a history of adoption, but only on whether it could easily be adopted.

9.1.3 Separation of Concerns

Features are the main concerns in SPL. The code which belongs to a certain feature can be viewed as a concern that should be separated, or at least separable, from other concerns (Apel et al., 2013). Indeed, separation of concerns states that features should be modular, well-defined, and separated in design and in implementation.

In the composition approaches (FOP, AFM, and mixsets when used compositionally), features are comprised of individual modules. Therefore, they adhere well to the separation of concerns principle. FOP is weaker than AFM and mixsets as it does not handle cases captured by aspects.

Separation of concerns in CPP is not easy to achieve; the code belonging to features often is scattered in different places with potential tangling among features (the only possible workaround is to create different CPP macros that take on different values depending on which feature is active, but this makes code very complex). FeatureCoPP is medium in this scale because it leaves some properties of the target features such as feature names within the body of the source code.

We have rated mixsets as excellent (i.e. state-of-the-art) regarding separation of concerns because feature code can always be put in separate files. Umple's need to inject code labels to allow fine-grained injection in methods is no more onerous than any other approach we have rated as excellent (AFM).

9.1.4 Traceability

Traceability expresses the ability to map SPL features from high-level representation in the problem space (i.e. feature models) into their low-level representation in the solution space. FOP, AFM and mixsets are composition approaches, supporting direct mapping from features to their code. CPP and FeatureCoPP do not support direct traceability; however, it can be achieved virtually through tools. Mixsets in the annotative style has reduced feature traceability, however, the mixset compositional format can be automatically generated from the annotative form.

It should be noted that in a planned release of Umple, requirements (in plain text or in a requirements domain-specific language) can be managed in addition to from code and model representations. Since the requirements can be in mixsets, it will be possible to generate the requirements for a particular product variant or feature. This feature was still under early beta testing at the time this thesis was published.

9.1.5 Information Hiding

Information hiding describes the ability to offer external interfaces through which feature modules can be integrated with the system components without exposing their internal parts (Prehofer, 2006). This criterion is essential for module reasoning since software engineers can reason about modules without deeply delving into their internal parts. Information hiding should be applicable to SPL features' context in that communication within features should pass through explicit interfaces.

CPP does not support any level of information hiding for implementing features. FeatureCoPP supports referencing feature fragments which can be considered a kind of information hiding. FOP has the best mark since allows method overriding for classes and interfaces. AFM is not good as FOP because aspect injection highly depends on statement sequence and internal calls inside methods. FeatureCoPP and inline mixsets breach the principle of the information hiding and require exposure methods' content. Mixsets go further by allowing access to local context too.

9.1.6 Granularity

Granularity describes the level of variability that can be achieved; either coarse-grained or fine-grained. This criterion has been discussed in the background of Chapter 5 . CPP and FeatureCoPP have high granularity, in the sense that one can inject characters essentially anywhere in the code, even within a statement. Mixsets are better than FOP and AFM because of the capability to add code at the statement level, but mixsets cannot be used to make variation *within* a statement – duplication of the entire statement would be required, with one copy having the desired change. For most purposes this is not a significant limitation, and in reality it imposes a useful discipline that can make programs easier to understand.

9.1.7 Uniformity

Batory formulated the principle of uniformity as the extent to which a variability approach can be applied across a wide range of artifacts including non-code artifacts (Batory et al., 2004). The annotation mechanism of CPP can be applied to source code and no-code artifacts. FeatureCoPP uses a single encoding technique that is similar to CPP. FOP has high uniformity because it suggests, even for extending non-code artifacts, using a common style to reference variant artifacts; this is similar to the use of super keyword to reference a parent class in object-oriented programming. AFM uses two different techniques (FOP and AOP). Hence, uniformity is not well supported in AFM.

Uniformity in mixsets is better than AFM's because mixsets can be uniformly applied to all constructs like #ifdef of CPP. For example, in Umple it is applicable to classes, methods, traits, state machines, associations, comments, other mixsets, use-statements activating mixsets, and so on. We have not yet, however, released a version of the system that applies mixsets to formats

such as XML or JSON, which has been done with CPP. Inline mixsets can be applied to non-code artifacts similarly to CPP.

9.1.8 Language Independence

Language independence measures the effort and tool changes required to generalize the variability mechanism across several languages. CPP is mostly used for C/C++ but as a preprocessor, it is completely language independent. FeatureCoPP is also a preprocessor, hence, it is also language independent but there is no actual implementation for FeatureCoPP. FOP is based on AHEAD, which is a general concept to layout features in containment hierarchies that include feature's code and non-code artifacts but requires a tailored compositional mechanism. AFM is an architectural approach that builds on AHEAD and requires both FOP and AOP. Mixsets require a composer and conditional compilation. The former is language-specific while the latter is a general concept.

Use of mixsets currently requires use of Umple, which assumes the codebase consists of Umple, Java, C++, PHP or Ruby. Other languages can be used with Umple, but this has not been tested. A lightweight composer for mixsets targeting a specific language could be another possibility. For instance, a mixset composer can be implemented particularly for the C language since there are excellent case studies that are publicly available and could be used to measure the usefulness of mixsets.

9.2 Tool Support Comparison

In this section we compare mixsets' tool support as in the Umple suite (command line interface (CLI), UmpleOnline web tool, Eclipse plug in) versus a set of related tools that can be used to manage SPL development. The difference in emphasis between this section and the last section is that the last section focused on general methodologies, whereas here we focus on specific implemented tools. The approaches we compare beside the Umple implementation of mixsets are:

- C preprocessor directives (#define, #ifdef, #include) (Stallman and Weinberg, 1987).This is the classic way variants have been managed for decades in languages such as C and C++. It has a tendency to make code very complex, however (Somé and Lethbridge, 1998). CPP is used as CLI software and can be used separately or with different languages.

- Git Branches. It is possible to manage versions of software in branches in a configuration management system, merging only certain sets of changes into branches that correspond with a particular variant (Rubin et al., 2012). Git CLI as well as GUI Git tools such as GitHub Desktop can be used to merge feature branches to a mainstream SPL base branch.
- Aspect Orientation: This is a group of technologies that can inject code into other code to add features (Kiczales et al., 1997). Most examples of its practical use are for features like logging and authentication. AspectJ is a popular and active implementation (Kiczales et al., 2001). AJDT is an Eclipse plugin that offers support for AspectJ for Java code (Clement et al., 2003).
- File inclusion with mixins: This is Umple's state before the extensions described in this thesis. The approach is also used in the mixin and the reopening technologies in Ruby (Günther and Sunkle, 2009, pp. 9–10).
- Clafer (Bąk et al., 2016). This is a textual language offers a class modeling language with first-class support for feature modeling. It supports certain modeling capabilities such as state machines and parts hierarchies. However, it does not support variability management of source code.
- Pure::variants (pure::variants, 2021). This is a sophisticated commercial tool that plugs in to Eclipse and works with a variety of other tools, enabling managing of variants. It allows generation of variants of executable systems in a language-independent way; for example it can generate C preprocessor directives.
- FeatureIDE (Thüm et al., 2014). FeatureIDE is an Eclipse plugin that enables FOSD through different variability approaches. It supports C preprocessor directives, Antenna preprocessor (Pleumann et al., 2002), Munge preprocessor (*Munge Maven Plugin*, 2011), feature-oriented programming (FOP) using FeatureHouse (Apel, Kästner, et al., 2009), FOP using AHEAD (Batory et al., 2004) and aspect-oriented programming using AspectJ (Kiczales et al., 2001).

9.2.1 Comparison Criteria

In Table 16 we characterize the above approaches, along with Umple, based on the criteria listed below. Capitalized items in the table are strengths. Lowercase, italicized items are weaknesses.

— ***Free and open source***. This is self-explanatory. Only pure::variants is commercial.

— ***Mechanism weight***. This is judged based on the number of syntactic elements in the language, the commands, and the software footprint needed to control the use of the language. The lower the weight, the more quickly and easily the technology can be learned and applied. A technology will naturally be heavier if it can do more; the challenge in technologies is to be able to do more with less through synergy. Umple and the C preprocessor are the lightest. The C preprocessor's capabilities are centered around the #define and #ifdef keywords, whereas Umple's capabilities are centered around the mixsets, the use and the require keywords. C preprocessor and Umple can be managed through the CLI, which facilitates integration with other tools. At the opposite end are pure::variants and FeatureIDE, which require installation of a powerful but heavyweight tool.

— ***IDE agnosticism***: If a tool requires a particular IDE (such as Eclipse), then it is marked 'No' here. FeatureIDE and pure::variants are IDE-dependent. The remaining tools can be used with Eclipse, on the command-line or in a variety of other IDEs. Without IDE agnosticism the scope of usage of a technology is clearly limited.

— ***Feature model visibility and traceability***: This is judged based on whether the feature model can be viewed by the developers as a distinct entity, separate from other information managed by the technology, and also the extent to which the tool provides a way for user can clearly see what aspects of the system's design and source code correspond to each feature. Tools such as Git and the C preprocessor were not designed with visibility of the feature model in mind, so are ranked low here. We have tried to make this one of the strengths of Umple: Umple can generate a feature model diagram and allows easy searching for the elements of each mixset. pure::variants also has a lot of strength in this area. FeatureIDE offers feature modeling capability but the linking between features in feature model and their associated fragments in code is not clear.

— ***Feature model expressiveness***: This is high if all the most important constraints among features can be described by the technology. This includes constraints indicating that one feature requires another, some features are optional, and so on. The C preprocessor, Git and Umple prior

to the changes described in this thesis had no capability for representing such constraints. Clafer, pure::variants and the new Umple with mixsets and require statements, explicitly allow such representation so are rated high. If pointcuts are well-described, some of the needed representational power can be harnessed, so we have rated aspect orientation as moderate in this regard.

— ***Textual Notation***: Textual notations allow developers to use arbitrary text editors, to search easily, and to apply numerous text-manipulation tools. If the feature model has a textual notation, then this is tagged as follows: *Separate* indicates that the model is a distinct language separate from source code of the rest of the system. *Embedded* means that it is a distinct language that can be added to source code (as is classically done for the C preprocessor). *Blended* extends the idea of *Embedded*, such that the language can be integrated with source code (and other models if relevant) or it can be separated in certain files. This has been one of the key design goals of Umple. Umple, in particular, allows mixsets and require statements to be manipulated by other mixsets, and blends them tightly with other language features.

— ***Can manage source code***: This indicates whether the technology can be used to control features that differ only with respect to elements of code in languages such as Java and C++. Umple and aspect-orientation are intermediate in this: They can manage variations among blocks of code, but not, differences within any arbitrary line of code. The ability to manage traditional code is important for several reasons: The need to express detailed algorithms will always exist, and variants may need different algorithms. This ability is also important so existing bodies of code can be managed when describing variants.

— ***Can manage UML models***: This indicates whether the technology can manage variants in full class models (including associations), state machines and other modeling elements. It is important in order to raise the abstraction of a system's description. Of the tools listed, only Umple is designed to do this directly and reasonably comprehensively. Clafer does have a growing capability in this direction but does not generate code from its models. The notation '*only indirectly*' means that the technology can manage other languages, so could be made to manage a textual modeling language (including managing Umple).

— ***Analysis sophistication for variants***: This is judged based on the capabilities of the logic language used to generate configurations out of variability models, and the deduction engine available during the configuration process to optimize, find conflicts, and fix problems. This is a strength of Clafer, pure::variants, and FeatureIDE. The ability to analyze variants is important to prevent errors such as inclusion of conflicting features.

Summary

Overall, our mixsets technology has been designed to have a high rating in almost all of the above criteria. In particular, it is the only open-source tool that can textually and directly manage variants in both source code and models. Umple is currently not the strongest tool when it comes to feature modeling analysis: Our assessment is that the high level of sophistication available in other tools are practically limited to a few use cases and is not commonly needed for most SPL projects, however. Mixsets also cannot control variations at the expression level of traditional code. However, there tend to be relatively few instances of expression level variability. These instances necessarily require refactoring (e.g. by extracting methods) so that mixsets can control code variability at the needed granularity.

Table 16. Advantages of certain approaches to managing variants and product lines

	Free and Open Source	Mechanism Weight	IDE agnostic	Feature model visibility	Feature model expressiveness	Textual notation	Can manage source code?	Can manage UML models?	Analysis sophistication for variants
C pre-processor	YES	VERY LIGHT	YES	*low*	*low*	Embedded	YES	*only indirectly*	*None*
Clafer	YES	LIGHT	YES	Moderate	HIGH	Separate	*no*	Partly	HIGH
pure::variants	*no*	*Heavy*	*No*	HIGH	HIGH	*no*	YES	*only indirectly*	HIGH
Git Branches	YES	Moderate	YES	*low*	*low*	Separate	YES	*only indirectly*	*Low*
Aspect Orientation	YES	Moderate	YES	*low*	Moderate	Separate	Block level	*only indirectly*	*Low*
Alternate file inclusion with mixins	YES	VERY LIGHT	YES	*low*	*low*	Separate	File level	DIRECTLY	*Low*
FeatureIDE	Yes	*Heavy*	No	Moderate	HIGH	*no*	Block level	only indirectly	HIGH
Umple	**YES**	**LIGHT**	**YES**	**HIGH**	**HIGH**	**BLENDED**	**Block level**	**DIRECTLY**	**Moderate**

9.3 Limitations

This section discusses the weaknesses of the mixset concept in general as well as our approach to implement and validate the technology we have developed.

9.3.1 Labeled Aspect Breaching of Information Hiding

Information hiding is a core principle to code modularity. Yet to offer finer granularity that also overcomes the need of many hook methods in a classical compositional SPL approach, revealing modules' implementation details become inevitable. In our approach, statement level

injection via labels requires exposing method implementation details. It is also the case with aspect-oriented programming with a lower degree due to its limited access. This exposure to some extent sacrifices method information hiding to achieve the goal of improving feature modularity.

To mitigate this weakness and to maintain method modularity, access restrictions can be imposed on code labels (what can accessed, changed, etc.) in advance or at least to be published as an "API" for a module/method with adequate documentation. For example, restrictions inside method statements can be further limited by employing Design by Contract (DbC) techniques (Meyer, 1998). Figure 38 exemplifies a rough idea to employ DbC contracts to limit the variability scope that is granted for code labels. This extension has not been implemented in Umple.

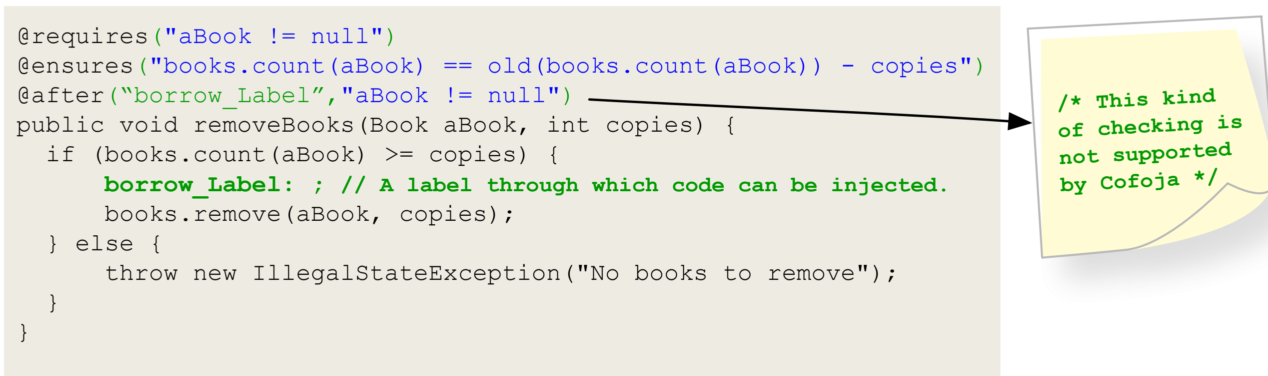

Figure 38: A modified DoC example to show the idea of restricting the variability inside methods using Contracts for Java, or Cofoja (Finney, 2011).

9.3.2 Insufficient Evaluation for Mixsets as Proactive SPL

The chosen version of Berkeley DB and Umple both were refactored into feature driven SPLs to demonstrate the feasibility of mixsets. The refactoring process followed is classified as a *reactive* SPL adoption process. On the other hand, there are SPLs which are designed from scratch following *proactive* SPL adoption. The mixset approach still has not been fully tested as a proactive process to adopt SPL, although some development of Umple has continued since the refactoring, resulting in new features being added with their own mixsets.

Therefore, more formal empirical evidence is needed to determine whether there are improvements in understandability, maintainability, and code size when mixsets are used, in comparison to other SPL proactive techniques such as frameworks, and control version branches.

9.3.3 Biases of Mixsets Evaluation

In Section 9.1 and Section 9.2 we reported the evaluation of mixsets in comparison to other existing techniques and tools in terms of several desired capabilities (separation of concerns, traceability, etc.). The analysis methodology employed to rate the techniques comes from manual and logical exercise, which entails personal bias. This is an internal threat to the validity and reliability of the findings reported.

To increase the replicability of the results, the population validity and the ecological validity need be addressed. The population validity can be treated by having other groups of people such as graduate students in a course project repeat the same case study of refactoring Berkeley DB. The ecological validity could be verified by applying the same refactoring process to other software systems in different domains, comparing the mixset refactoring to the refactoring achieved with other SPL tools. The ecological validity is more difficult to achieve since there are limited case studies that refactor software systems into SPLs.

9.3.4 Limitation of Mixsets Currently to Umple

Mixsets offer an integrated variability approach that seeks explicit variability relationships between varying elements and supports annotative and compositional variability in a unified way. However, mixsets have been implemented only so far within Umple, which limits the applicability to use them in a broader scope.

Generalization of mixsets as both preprocessor and composer that handles diverse language syntaxes is future work. The generalized form of mixsets should support almost any textual language, for instance, through recognizing the AST represented in its grammar. The main challenge in this goal is to offer an abstraction that handles diverse AST representations.

Processing annotative variability may not be difficult to accomplish, since conditional compilation does not account for the underlying syntactic structure. However, it is not easy to do universal composers that also transform inline mixsets into their compositional counterparts.

In this work, we have bound mixsets to Umple, which limits their scope of applicability. But this was not a bad decision since Umple enabled the testing of mixsets with a wide variety of

syntactic and semantic elements. Also, Umple has been in use for large groups of students across the globe, and our implementation in Umple allows mixsets to be exposed to a large audience.

Chapter 10 Conclusions and Future Work

This thesis has demonstrated the mixset concept which is a textual variability approach that supports both annotative and compositional variability in a unified manner. This could be implemented in any language, but we have implemented it as an extension to Umple. It allows both feature-oriented and product-line development. The key additions are:

- The *mixset* concept and keyword, used to tag any set of elements (Umple elements in our implementation) as parts of features that can optionally be included.
- Extension of the *use* keyword to activate mixsets (previously it just resulted in the inclusion of files).
- The *require* keyword (and several associated operators), used to describe constraints among features, and therefore keep the system logically consistent.
- Synchronizing mixsets appearing in models and code. Models and pure code can be parts of the same mixsets. At the code level, they can be formulated as *inline* mixsets, or *labels* through which aspects can inject code.
- A generator in Umple that refactors annotative fragments into compositional fragments.

Mixsets' capabilities for feature and product-line modeling allow code and model to be arranged in a variety of ways and have a set of advantages that bring together the strengths found in both annotative and compositional SPL technologies.

As outlined in the introduction chapter (Section 1.3), the thesis follows a design-science approach. The concept of mixsets has been through a continues loop of improvement and validation. The concept has been shaped through researching software variability literature as well as synthesizing our experience, thoughts, and reviewers' feedback received from submitted papers.

To validate our approach, three case studies have been presented, based on open-source projects. The first case study (Berkeley DB) is an SPL system referred to elsewhere in the literature, that has been used to compare different annotative and compositional SPL techniques. The second case study refactors Umple into an SPL and demonstrates our perspective on the practicality and the experience of using mixsets to refactor non-SPL software to an SPL. The study

also showed the flexibility of refactoring SPL annotations into compositional fragments using mixsets. The third case study applies Mixsets to solve the Rover Control Challenge Problem. It demonstrated the viability of mixsets to synchronize mixsets as a variability technique at the model level and at the code level.

Our work is available through the Umple command line interface (CLI), an Eclipse plugin and UmpleOnline. Umple exposes mixsets to a large number of university students (graduate and undergraduate), researchers and other developers in an easy manner due to the documentation and the useful examples. We capitalize on such exposure to improve and know better what strengths and weakness mixsets have as a SPL variability mechanism.

10.1 Contributions

The contributions of this thesis are answers to the research questions that are posed in Chapter 1. These contributions can be summarized into the following points:

1. The general concept of mixsets, including the ability to separate the concerns of systems using a mix of annotative and compositional approaches (introduced in Chapter 3).
2. The syntax of the mixset and require notation as added to Umple (Section 4.4, 4.7, 4.8, and 4.9).
3. The algorithms for applying mixsets (discussed in Section 4.6).
4. The implementation of mixsets in Umple (Lethbridge, 2021).
5. The ability to separate concerns in a software system at various levels of abstraction from high-level models down to statements within methods (Section 5.4).
6. Three case studies implemented for mixsets (form Chapter 6 , Chapter 7 , and Chapter 8).
7. The examples, test cases, and user manual documentation (as in UmpleOnline).
8. The comparison with other approaches and tools (presented in Chapter 9).

The items 1-4 are partial answers to Research Question 1 which is:

RQ1: What would be an effective technique to facilitate SPL construction via a combined SPL variability approach in which composition would require less preplanning?

RQ1.1 It is possible to mimic annotation, or offer an approach that allows representation at the same time?

RQ1.2 What would be an effective technique to directly transform annotative fragments into compositional fragments?

A) What is the sufficient level of granularity that should be allowed in annotative fragments that allow direct refactoring?

B) How to identify points of variation in code with no identifiable structure such as a specific line inside a method?

From a practical perspective, SPL engineers can benefit from the mixset approach by applying inline mixsets as disciplined annotations and then refactoring inline mixsets into compositional mixsets as the case study in Chapter 5 showed. A semi-automatic inline-fragment refactoring generator can map annotative fragments to fully compositional fragments using the AST in a straightforward way. Statement-level granularity can be achieved for a composer with the help of code labels which can identify arbitrary code such as a specific line inside a method. Mixsets elevate composition to the statement level, which avoids many hook methods that merely act as workarounds and damage understandability of the code. Still, there will be a few cases in which workarounds are needed for expression-level granularity in our approach, but this kind of change is less common than statement level changes. The mixsets approach has the advantage of being flexible to represent variability in SPLs using the three choices: annotative, compositional, and the mixture of both types.

Contributions 5-7 are partial answers to Research Question 2 which is:

RQ2: What types of evaluation techniques should be applied in order to ensure the result of RQ1?

RQ2.1 How to compare mixsets to other SPL approaches?

To evaluate the mixset approach in this thesis, we have used three case studies. The evaluation criteria are centered around the goal of reduced preplanning, higher adoption, easier separation of concerns and increased granularity. These criteria are achieved through mixsets' flexibility to represent annotative and compositional fragments for both models and embedded code, the number of workarounds required to form features, the possibility to transform annotative fragments into compositional fragments, and the ability to document and express feature modeling within the source code. Tool support offered for mixsets is very comparable to its alternatives through a textual language that works comprehensively with all elements of Umple and directly manages variants in both source code and models. The availability of the online tool "UmpleOnline" is an advantage for mixsets. It would spread the use of Mixsets and would increase its adoption.

10.2 Future Work

Some ideas for future work have arisen from our work. These have been mentioned at the end of previous chapters or sections. The following summarizes:

- It would be good to experiment with the application of mixsets to support diverse language syntaxes beside current support for Umple and embedded code. The generalized form should be able to serve almost any textual language through its grammar. The main challenge in this goal is to offer an abstraction to handle diverse AST representations. The second alternative is to handle each language's constructs in a special way.
- Employing the Design by Contract (DbC) techniques (Meyer, 1998) to enforce some restrictions on variability inside method statements. Methods' contracts should be extending to limit what injection can access when labels are used. This goal would be to maintain information hiding as well as feature modularity.
- Automate the refactoring from annotative fragments to compositional fragments. The offered generator still requires some manual work to remove mixsets occurring in the original Umple files.
- Enable the syntactic checking of mixsets that are used in isolation of other mixsets and the rest of the language elements as a first step. The second step would be to formally

analyze the SPL or part of it to ensure the syntax correctness. Currently, any code written inside a mixset that is not used is only scanned for the presence of other mixset-related syntax. Therefore, there will be a portion of the code that is not comprehensively parsed.

In addition, there are several other ideas that have emerged through our research and have not been tried. These ideas include:

- Allowing renaming, selection and deletion operators on mixset inclusion, as is currently supported in Umple traits. This would allow generic features to be included in somewhat-different base systems and would enable Umple to gain the capabilities of delta-oriented product-line tools.
- Explore the idea of refactoring mixsets into existing compositional SPL approaches. This implies implementing a generator to produce fully AOP SPL, for example, out of mixsets whereas annotative fragments are refactored into aspect injections. Such a generator would require a capability to assess the refactoring complexity and guide the process for fragments that contain code which include several accesses/changes to local variables. Furthermore, the issue of handling cascading changes (when a feature builds the change of another feature's change) and scoped keywords (such as continue, break, and catch exceptions) would eventually raise for such generators.
- Conduct a comprehensive empirical study of the effectiveness of the approach with end users. The study would assess population validity. In addition, an empirical study may offer helpful insight software developer perspective on the learning curve, adoption, and the preferable type of mixsets.
- Improve tool support for mixsets. It would be good in Umple IDEs to be able to show all use statements referring to a mixset, to show all fragments, to show differences (diffs) between variants if a certain mixset is applied or not, and so on.

References

Abdelzad, V., and Lethbridge, T. C. (2017). Promoting traits into model-driven development. *Software & Systems Modeling*, *16*(4), 997–1017. https://doi.org/10.1007/s10270-015-0505-x

Adams, B., Meuter, W. D., Tromp, H., and Hassan, A. E. (2009). Can we refactor conditional compilation into aspects? *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development*, 243–254. https://doi.org/10.1145/1509239.1509274

Adesina, O. O., Lethbridge, T. C., Somé, S. S., Abdelzad, V., and Belle, A. B. (2018). Improving formal analysis of state machines with particular emphasis on and-cross transitions. *Computer Languages, Systems & Structures*, *54*, 544–585. https://doi.org/10.1016/j.cl.2017.12.001

Algablan, A. (2020). *BerkeleyDbUmple GitHub Repository*. https://github.com/gublan24/BerkeleyDbUmple

Algablan, A. (2021). *UmpleSPL GitHub Repository*. https://github.com/gublan24/umpleSPL

Aljamaan, H., and Lethbridge, T. C. (2012). Towards Tracing at the Model Level. *2012 19th Working Conference on Reverse Engineering*, 495–498. https://doi.org/10.1109/wcre.2012.59

Almaghthawi, S. (2020). *Model-Driven Testing in Umple (Doctoral dissertation)*. University of Ottawa. http://hdl.handle.net/10393/40344

Ambler, S. W. (2004). *The Object Primer: Agile Model-Driven Development with UML 2.0* (3rd ed.). Cambridge University Press.

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines, Concepts and Implementation*. Springer-Verlag Berlin An. https://doi.org/10.1007/978-3-642-37521-7

Apel, S., Janda, F., Trujillo, S., and Kästner, C. (2009). Model Superimposition in Software Product Lines. *ICMT 2009: Theory and Practice of Model Transformations*, 4–19. https://doi.org/10.1007/978-3-642-02408-5_2

Apel, S., and Kästner, C. (2009). An Overview of Feature-Oriented Software Development. *The Journal of Object Technology*, *8*(5), 49. https://doi.org/10.5381/jot.2009.8.5.c5

Apel, S., Kästner, C., and Lengauer, C. (2009). FEATUREHOUSE: Language-independent, automated software composition. *2009 IEEE 31st International Conference on Software Engineering*, 221–231. https://doi.org/10.1109/icse.2009.5070523

Apel, S., Leich, T., Rosenmüller, M., and Saake, G. (2005). FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. *Proceedings of the 4th International*

Conference on Generative Programming and Component Engineering, 125–140. https://doi.org/10.1007/11561347_10

Apel, S., Leich, T., and Saake, G. (2008). Aspectual Feature Modules. *IEEE Transactions on Software Engineering*, *34*(2), 162–180. https://doi.org/10.1109/tse.2007.70770

Apel, S., and Lengauer, C. (2008). Superimposition: A Language-Independent Approach to Software Composition. *International Conference on Software Composition*, 20--35.

Atkinson, C., and Kühne, T. (2003). Model-driven development: a metamodeling foundation. *IEEE Software*, *20*(5), 36–41. https://doi.org/10.1109/ms.2003.1231149

Badreddin, O., Forward, A., and Lethbridge, T. C. (2014a). *Exploring a Model-Oriented and Executable Syntax for UML Attributes* (R. Lee, Ed.; pp. 33–53). Springer International Publishing.

Badreddin, O., Forward, A., and Lethbridge, T. C. (2014b). *Improving Code Generation for Associations: Enforcing Multiplicity Constraints and Ensuring Referential Integrity* (["Roger Lee"], Ed.; pp. 129–149). Springer International Publishing.

Badreddin, O., Lethbridge, T. C., Forward, A., Elaasar, M., Aljamaan, H., and Garzon, M. A. (2014). Enhanced code generation from UML composite state machines. *2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, 235–245.

Bąk, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., and Wąsowski, A. (2016). Clafer: unifying class and feature modeling. *Software & Systems Modeling*, *15*(3), 811–845. https://doi.org/10.1007/s10270-014-0441-1

Batory, D. (2005). Feature models, grammars, and propositional formulas. *International Conference on Software Product Lines*, 7–20.

Batory, D., Sarvela, J. N., and Rauschmayer, A. (2004). Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, *30*(6), 355–371. https://doi.org/10.1109/tse.2004.23

Behringer, B., Palz, J., and Berger, T. (2017). PEoPL: Projectional Editing of Product Lines. *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 563–574. https://doi.org/10.1109/icse.2017.58

Benavides, D., Segura, S., and Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: A literature review. *Information Systems*, *35*(6), 615–636. https://doi.org/10.1016/j.is.2010.01.001

Benduhn, F., Schroter, R., Kenner, A., Kruczek, C., Leich, T., and Saake, G. (2016). Migration from Annotation-Based to Composition-Based Product Lines: Towards a Tool-Driven

Process. *The Second International Conference on Advances and Trends in Software Engineering*.

Berg, K., Bishop, J., and Muthig, D. (2005). Tracing Software Product Line Variability: From Problem to Solution Space. *Proceedings of the 2005 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*, 182–191.

Berger, T., She, S., Lotufo, R., Wasowski, A., and Czarnecki, K. (2013). A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering*, *39*(12), 1611–1640. https://doi.org/10.1109/tse.2013.34

Bettini, L., Damiani, F., and Schaefer, I. (2015). Implementing type-safe software product lines using parametric traits. *Science of Computer Programming*, *97*, 282–308. https://doi.org/10.1016/j.scico.2013.07.016

Beuche, D. (2003). *Variant Management with pure:: variants*. Technical report, pure-systems GmbH, 2003. http://www. pure-systems. com.

Bhushan, M., Negi, A., Samant, P., Goel, S., and Kumar, A. (2020). A classification and systematic review of product line feature model defects. *Software Quality Journal*, *28*(4), 1507–1550. https://doi.org/10.1007/s11219-020-09522-1

Bosch, J., Capilla, R., and Hilliard, R. (2015). Trends in Systems and Software Variability [Guest editors' introduction]. *IEEE Software*, *32*(3), 44–51. https://doi.org/10.1109/ms.2015.74

Bosch, J., and Lee, J. (2010). Software Product Lines: Going Beyond. *14th International Software Product Line Conference, SPLC 2010*. https://doi.org/10.1007/978-3-642-15579-6

Bracha, G., and Cook, W. (1990). Mixin-based inheritance. *ACM SIGPLAN Notices*, *25*(10), 303–311. https://doi.org/10.1145/97946.97982

Brambilla, M., Cabot, J., Wimmer, M., and Baresi, L. (2017). Model-Driven Software Engineering in Practice. *Synthesis Lectures on Software Engineering*, *3*(1), 1–207. https://doi.org/10.2200/s00751ed2v01y201701swe004

Braz, L., Gheyi, R., Mongiovi, M., Ribeiro, M., Medeiros, F., and Teixeira, L. (2016). A Change-Centric Approach to Compile Configurable Systems with #ifdefs. *SIGPLAN Not.*, *52*(3), 109–119. https://doi.org/10.1145/3093335.2993250

Brown, A., Booch, G., Iyengar, S., Rumbaugh, J., and Selic, B. (2004). *An MDA Manifesto*.

Capilla, R., Gallina, B., Cetina, C., and Favaro, J. (2019). Opportunities for software reuse in an uncertain world: From past to emerging trends. *Journal of Software: Evolution and Process*, *31*(8). https://doi.org/10.1002/smr.2217

Chacón-Luna, A. E., Gutiérrez, A. M., Galindo, J. A., and Benavides, D. (2020). Empirical software product line engineering: A systematic literature review. *Information and Software Technology*, *128*, 106389. https://doi.org/10.1016/j.infsof.2020.106389

Clarke, D., Diakov, N., Hähnle, R., Johnsen, E. B., Schaefer, I., Schäfer, J., Schlatte, R., and Wong, P. Y. H. (2011). *Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language* (pp. 417–457). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-21455-4_13

Classen, A., Boucher, Q., and Heymans, P. (2011). A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, *76*(12), 1130–1143. https://doi.org/10.1016/j.scico.2010.10.005

Clement, A., Colyer, A., and Kersten, M. (2003). Aspect-oriented programming with AJDT. *ECOOP Workshop on Analysis of Aspect-Oriented Software*, *10*.

Clements, P., and Northrop, L. (2002). Software product lines - practices and patterns. *SEI Series in Software Engineering*.

Combemale, B., Barais, O., Alam, O., and Kienzle, J. (2012). Using CVL to Operationalize Product Line Development with Reusable Aspect Models. *The VARiability for You Workshop: Variability Modeling Made Useful for Everyone*. https://hal.inria.fr/hal-00730274

Couto, M. V., Valente, M. T., and Figueiredo, E. (2011). Extracting Software Product Lines: A Case Study Using Conditional Compilation. *2011 15th European Conference on Software Maintenance and Reengineering*, 191–200. https://doi.org/10.1109/csmr.2011.25

Czarnecki, K. (2013). *Variability in Software: State of the Art and Future Directions* (["Vittorio Cortellessa" and "Dániel Varró"], Eds.; pp. 1–5). Springer Berlin Heidelberg.

Czarnecki, K., and Antkiewicz, M. (2005). Mapping Features to Models: A Template Approach Based on Superimposed Variants. In R. Gluck and Michael Lowry (Eds.), *ACM SIGSOFT/SIGPLAN International Conference on Generative Programming and Component Engineering (GPCEextquoteright05)* (Vol. 3676, pp. 422–437). Springer-Verlag. https://doi.org/10.1007/11561347_28

Czarnecki, K., Antkiewicz, M., Kim, C. H. P., Lau, S., and Pietroszek, K. (2005). Model-driven software product lines. *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA '05*, 126–127. https://doi.org/10.1145/1094855.1094896

Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., and Wąsowski, A. (2012). *Cool features and tough decisions: a comparison of variability modeling approaches*. 173–182. https://doi.org/10.1145/2110147.2110167

Czarnecki, K., and Pietroszek, K. (2006). Verifying feature-based model templates against well-formedness OCL constraints. *Proceedings of the 5th International Conference on Generative Programming and Component Engineering - GPCE '06*, 211–220. https://doi.org/10.1145/1173706.1173738

Czarnecki, K., and Ulrich, E. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

Dijkstra, E. W. (1968). The structure of the multiprogramming system. *Communications of the ACM*, *11*(5), 341–346. https://doi.org/10.1145/363095.363143

Dimovski, A. S., Brabrand, C., and Wasowski, A. (2018). Variability abstractions for lifted analyses. *Science of Computer Programming*, *159*, 1–27. https://doi.org/10.1016/j.scico.2017.12.012

Eichelberger, H., and Schmid, K. (2013). A systematic analysis of textual variability modeling languages. *Proceedings of the 17th International Software Product Line Conference on - SPLC '13*, 12–21. https://doi.org/10.1145/2491627.2491652

Eichelberger, H., and Schmid, K. (2015). Mapping the design-space of textual variability modeling languages: a refined analysis. *International Journal on Software Tools for Technology Transfer*, *17*(5), 559–584. https://doi.org/10.1007/s10009-014-0362-x

El-Sharkawy, S., Yamagishi-Eichler, N., and Schmid, K. (2019). Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Information and Software Technology*, *106*, 1–30. https://doi.org/10.1016/j.infsof.2018.08.015

Ernst, M. D., Badros, G. J., and Notkin, D. (2002). An empirical analysis of c preprocessor use. *IEEE Transactions on Software Engineering*, *28*(12), 1146–1170. https://doi.org/10.1109/tse.2002.1158288

Feigenspan, J., Kästner, C., Apel, S., Liebig, J., Schulze, M., Dachselt, R., Papendieck, M., Leich, T., and Saake, G. (2013). Do background colors improve program comprehension in the #ifdef hell? *Empirical Software Engineering*, *18*(4), 699–745. https://doi.org/10.1007/s10664-012-9208-x

Fenske, W., Krüger, J., Kanyshkova, M., and Schulze, S. (2020). *#ifdef Directives and Program Comprehension: The Dilemma between Correctness and Preference*. 255–266. https://doi.org/10.1109/icsme46990.2020.00033

Filman, R. E., and Friedman, D. P. (2000). Aspect-oriented programming is quantification and obliviousness. *Workshop on Advanced Separation of Concerns, OOPSLA*, *2000*.

Finney, L. (2011). *Design by Contract in Java with Google*. https://objectcomputing.com/resources/publications/sett/september-2011-design-by-contract-in-java-with-google

Frakes, W. B., and Kang, K. (2005). Software reuse research: status and future. *IEEE Transactions on Software Engineering*, *31*(7), 529–536. https://doi.org/10.1109/tse.2005.85

Garzón, M. A., Lethbridge, T. C., Aljamaan, H., and Badreddin, O. (2014). Reverse Engineering of Object-Oriented Code into Umple Using an Incremental and Rule-Based Approach. *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, 91–105.

Glinz, M. (2011). A glossary of requirements engineering terminology. *Standard Glossary of the Certified Professional for Requirements Engineering (CPRE) Studies and Exam, Version*, *1*, 56.

Gomaa, H. (2004). *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc.

Grönninger, H., Krahn, H., Rumpe, B., Schindler, M., and Völkel, S. (2014). Textbased modeling. *Proceedings of the 4th International Workshop on Software Language Engineering*.

Günther, S., and Sunkle, S. (2009). Feature-oriented programming with Ruby. *Proceedings of the First International Workshop on Feature-Oriented Software Development - FOSD '09*, 11–18. https://doi.org/10.1145/1629716.1629721

Gutierrez, F. (2014). *Introducing Spring Framework, A Primer*. https://doi.org/10.1007/978-1-4302-6533-7

Hailpern, B., and Tarr, P. (2006). Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, *45*(3), 451–461. https://doi.org/10.1147/sj.453.0451

Harrison, W., and Ossher, H. (1993). Subject-oriented programming: a critique of pure objects. *ACM SIGPLAN Notices*, *28*(10), 411–428. https://doi.org/10.1145/165854.165932

Haugen, Ø., Wąsowski, A., and Czarnecki, K. (2013). CVL: common variability language. *17th International Software Product Line Conference*, 277–277. https://doi.org/10.1145/2491627.2493899

Hause, M., and others. (2006). The SysML modelling language. *Fifteenth European Systems Engineering Conference*, *9*, 1--12.

Heo, S.-H., and Choi, E. M. (2006). Representation of Variability in Software Product Line Using Aspect-Oriented Programming. *Fourth International Conference on Software*

Engineering Research, Management and Applications (SERA'06), 66–73. https://doi.org/10.1109/sera.2006.57

Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly*, *28*(1), 75–105. http://www.jstor.org/stable/25148625

Hunsen, C., Zhang, B., Siegmund, J., Kästner, C., Leßenich, O., Becker, M., and Apel, S. (2016). Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering*, *21*(2), 449–482. https://doi.org/10.1007/s10664-015-9360-1

Hunt, J. (2006). *Feature-Driven Development* (pp. 161–182). Springer London. https://doi.org/10.1007/1-84628-262-4_9

Igarashi, A., Pierce, B. C., and Wadler, P. (2001). Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *23*(3), 396–450. https://doi.org/10.1145/503502.503505

Jalote, P., Briand, L., Hoek, A. van der, Nadi, S., Berger, T., Kästner, C., and Czarnecki, K. (2014). Mining configuration constraints: static analyses and empirical results. *Proceedings of the 36th International Conference on Software Engineering*, 140–151. https://doi.org/10.1145/2568225.2568283

Jézéquel, J.-M. (2008). Model driven design and aspect weaving. *Software & Systems Modeling*, *7*(2), 209–218. https://doi.org/10.1007/s10270-008-0080-5

Jézéquel, J.-M. (2012). Model-Driven Engineering for Software Product Lines. *ISRN Software Engineering*, *2012*, 1–24. https://doi.org/10.5402/2012/670803

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. https://doi.org/10.21236/ada235785

Kästner, C. (2007). Aspect-oriented refactoring of Berkeley DB. *University of Magdeburg, Germany*.

Kästner, C. (2012). *CIDE GitHub Repository*. https://github.com/ckaestne/CIDE/tree/master/CIDE_Samples/cide_samples

Kästner, C., and Apel, S. (2008a). *Integrating Compositional and Annotative Approaches for Product Line Engineering*.

Kästner, C., and Apel, S. (2008b). Type-checking Software Product Lines – A Formal Approach. *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 258–267. https://doi.org/10.1109/ase.2008.36

Kästner, C., Apel, S., and Batory, D. (2007). A Case Study Implementing Features Using AspectJ. *Proceedings of the 11th International Software Product Line Conference*, 223–232.

Kästner, C., Apel, S., and Kuhlemann, M. (2008). Granularity in Software Product Lines. *Proceedings of the 30th International Conference on Software Engineering*, 311–320. https://doi.org/10.1145/1368088.1368131

Kästner, C., Apel, S., Thüm, T., and Saake, G. (2012). Type Checking Annotation-Based Product Lines. *ACM Trans. Softw. Eng. Methodol.*, *21*(3). https://doi.org/10.1145/2211616.2211617

Kästner, C., Apel, S., Trujillo, S., Kuhlemann, M., and Batory, D. (2008). Language-independent safe decomposition of legacy applications into features. *School of Computer Science, University of Magdeburg, Germany, Tech. Rep*, *2*.

Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. (2011). Variability-aware parsing in the presence of lexical macros and conditional compilation. *ACM SIGPLAN Notices*, *46*(10), 805–824. https://doi.org/10.1145/2048066.2048128

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. *European Conference on Object-Oriented Programming*, 327--354.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In ["Mehmet Akşit" and "Satoshi Matsuoka"] (Eds.), *Europ. Conf. Object-Oriented Programming (ECOOP)* (pp. 220–242). Springer Berlin Heidelberg.

Kienzle, J., Abed, W. A., and Klein, J. (2009). Aspect-oriented multi-view modeling. *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development*, 87–98. https://doi.org/10.1145/1509239.1509252

Klein, J., Hélouët, L., and Jézéquel, J.-M. (2006). Semantic-Based Weaving of Scenarios. *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, 27–38. https://doi.org/10.1145/1119655.1119662

Klein, J., and Kienzle, J. (2007). Reusable aspect models. *Abstract Book of 11th Workshop on Aspect Oriented Modeling, AOM at Models' 07*.

Koscielny, J., Holthusen, S., Schaefer, I., Schulze, S., Bettini, L., and Damiani, F. (2014). DeltaJ 1.5: Delta-Oriented Programming for Java 1.5. *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, 63–74. https://doi.org/10.1145/2647508.2647512

Krueger, C., and Clements, P. (2013). Systems and software product line engineering with BigLever software gears. *Proceedings of the 17th International Software Product Line Conference Co-Located Workshops*, 136--140.

Krueger, CharlesW. (2002). *Easing the Transition to Software Mass Customization* (["Frank van der Linden"], Ed.; pp. 282–293). Springer Berlin Heidelberg.

Kruger, J., Ludwig, K., Zimmermann, B., and Leich, T. (2018). Physical Separation of Features: A Survey with CPP Developers. *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2042–2049. https://doi.org/10.1145/3167132.3167351

Krüger, J., Schröter, I., Kenner, A., Kruczek, C., and Leich, T. (2016). FeatureCoPP: Compositional Annotations. *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*, 74–84. https://doi.org/10.1145/3001867.3001876

Laddad, R. (2003). *AspectJ in action: practical aspect-oriented programming*. Dreamtech Press.

Lahire, P., Morin, B., Vanwormhoudt, G., Gaignard, A., Barais, O., and Jézéquel, J.-M. (2007). Introducing Variability into Aspect-Oriented Modeling Approaches. In ["Gregor Engels", "Bill Opdyke", "Douglas C. Schmidt", and "Frank Weil"] (Eds.), *International Conference on Model Driven Engineering Languages and Systems* (pp. 498–513). Springer Berlin Heidelberg.

Le, D. M., Lee, H., Kang, K. C., and Keun, L. (2013). *Validating Consistency between a Feature Model and Its Implementation* (J. Favaro and M. Morisio, Eds.; pp. 1–16). Springer Berlin Heidelberg. https://doi.org/https://doi.org/10.1007/978-3-642-38977-1_1

Le, D., Walkingshaw, E., and Erwig, M. (2011). #ifdef Confirmed Harmful: Promoting Understandable Software Variation. *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, *1*, 143–150. https://doi.org/10.1109/vlhcc.2011.6070391

Lethbridge, T. C. (2018). *Umple/roverChallenge GitHub Repository*. Umple Case Study Implementing an Autonomous Rover. https://github.com/umple/roverChallenge

Lethbridge, T. C. (2021). *Umple GitHub Repository*. https://github.com/umple/umple

Lethbridge, T. C., Abdelzad, V., Orabi, M. H., Orabi, A. H., and Adesina, O. (2016). *Merging Modeling and Programming Using Umple* (T. Margaria and B. Steffen, Eds.; pp. 187–197). Springer International Publishing.

Lethbridge, T. C., Abdulaziz, Abdelzad, V., Garzón, M., Forward, A., Aljamaan, H., Almaghthaw, S., Adesina, O., and Ng, R. (2021). *umple/umple: Umple Release 1.31.1*. https://doi.org/10.5281/zenodo.5218626

Lethbridge, T. C., and Algablan, A. (2018a). Applying Umple to the rover control challenge problem: A case study in model-driven engineering. *MODELS Workshops*.

Lethbridge, T. C., and Algablan, A. (2018b). Using Umple to Synergistically Process Features, Variants, UML Models and Classic Code. *Leveraging Applications of Formal Methods,*

Verification and Validation. Modeling. ISoLA, 69–88. https://doi.org/10.1007/978-3-030-03418-4_5

Lethbridge, T. C., Forward, A., and Badreddin, O. (2010). Umplification: Refactoring to Incrementally Add Abstraction to a Program. *17th Working Conference on Reverse Engineering*, 220–224. https://doi.org/10.1109/wcre.2010.32

Lethbridge, T. C., Forward, A., Badreddin, O., Brestovansky, D., Garzon, M., Aljamaan, H., Eid, S., Orabi, A. H., Orabi, M. H., Abdelzad, V., Adesina, O., Alghamdi, A., Algablan, A., and Zakariapour, A. (2021). Umple: Model-driven development for open source and education. *Science of Computer Programming*, *208*, 102665. https://doi.org/10.1016/j.scico.2021.102665

Letovsky, S., and Soloway, E. (1986). Delocalized Plans and Program Comprehension. *IEEE Software*, *3*(3), 41–49. https://doi.org/10.1109/ms.1986.233414

Levin, J. (2009). *System generation for time and activity management product lines (Master's thesis)*. University of Ottawa. https://www.site.uottawa.ca/~tcl/gradtheses/jlevin/JenyaLevinMastersThesis.pdf

Liao, X., Zhou, S., Li, S., Jia, Z., Liu, X., and He, H. (2018). Do You Really Know How to Configure Your Software? Configuration Constraints in Source Code May Help. *IEEE Transactions on Reliability*, *67*(3), 832–846. https://doi.org/10.1109/tr.2018.2834419

Liebig, J., Apel, S., Lengauer, C., Kästner, C., and Schulze, M. (2010). An analysis of the variability in forty preprocessor-based software product lines. *2010 ACM/IEEE 32nd International Conference on Software Engineering*, *1*, 105–114. https://doi.org/10.1145/1806799.1806819

Liebig, J., Kästner, C., and Apel, S. (2011). Analyzing the discipline of preprocessor annotations in 30 million lines of C code. *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development - AOSD '11*, 191–202. https://doi.org/10.1145/1960275.1960299

Lopez-Herrejon, R. E., Batory, D., and Cook, W. (2005). Evaluating Support for Features in Advanced Modularization Technologies. *Proceedings of the 19th European Conference on Object-Oriented Programming*, 169–194. https://doi.org/10.1007/11531142_8

Malaquias, R., Ribeiro, M., Bonifácio, R., Monteiro, E., Medeiros, F., Garcia, A., and Gheyi, R. (2017). The Discipline of Preprocessor-Based Annotations Does #ifdef TAG n't #endif Matter. *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 297–307. https://doi.org/10.1109/icpc.2017.41

Mankin, B. (2008). *JCPP - A Java C Preprocessor*. https://www.anarres.org/projects/jcpp/

Marot, A., and Wuyts, R. (2008). A DSL to declare aspect execution order. *Proceedings of the 2008 AOSD Workshop on Domain-Specific Aspect Languages - DSAL '08*, 7. https://doi.org/10.1145/1404927.1404934

McKinney, S. (2019, August 5). *Manifold: A Preprocessor for Java*. https://jaxenter.com/manifold-preprocessor-for-java-160712.html

MDETools 2018 Challenge Problem. (2018). https://mdetools.github.io/mdetools18/challengeproblem.html

Medeiros, F., Kästner, C., Ribeiro, M., Nadi, S., and Gheyi, R. (2015). The Love/Hate Relationship with the C Preprocessor: An Interview Study. In J. T. Boyland (Ed.), *29th European Conference on Object-Oriented Programming (ECOOP 2015)* (Vol. 37, pp. 495--518). Schloss Dagstuhl--Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/lipics.ecoop.2015.495

Meliá, S., Cachero, C., Hermida, J. M., and Aparicio, E. (2016). Comparison of a textual versus a graphical notation for the maintainability of MDE domain models: an empirical pilot study. *Software Quality Journal*, *24*(3), 709–735. https://doi.org/10.1007/s11219-015-9299-x

Merkle, B. (2010). Textual modeling tools: overview and comparison of language workbenches. *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion - SPLASH '10*, 139–148. https://doi.org/10.1145/1869542.1869564

Meyer, B. (1998). Design By Contract. The Eiffel Method. *Proceedings. Technology of Object-Oriented Languages. TOOLS 26 (Cat. No.98EX176)*, 446–446. https://doi.org/10.1109/tools.1998.711043

Mezini, M., and Ostermann, K. (2003). Conquering aspects with Caesar. *AOSD03: 2nd International Conference on Aspect-Oriented Software Development*, 90–99. https://doi.org/10.1145/643603.643613

Michelon, G. K., Obermann, D., Assunção, W. K. G., Linsbauer, L., Grünbacher, P., and Egyed, A. (2020). Mining Feature Revisions in Highly-Configurable Software Systems. *Proceedings of the 24th ACM International Systems and Software Product Line Conference - Volume B*, 74–78. https://doi.org/10.1145/3382026.3425776

Moody, D. (2009). The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, *35*(6), 756–779. https://doi.org/10.1109/tse.2009.67

Morin, B., Klein, J., Barais, O., and Jézéquel, J.-M. (2008). A generic weaver for supporting product lines. *Proceedings of the 13th International Workshop on Early Aspects*, 11–18. https://doi.org/10.1145/1370828.1370832

Munge Maven Plugin. (2011). https://sonatype.github.io/munge-maven-plugin/

Murashkin, A., Antkiewicz, M., Rayside, D., and Czarnecki, K. (2013). *Visualization and exploration of optimal variants in product line engineering*. 111–115. https://doi.org/10.1145/2491627.2491647

Murphy, G. C., Lai, A., Walker, R. J., and Robillard, M. P. (2001). Separating Features in Source Code: An Exploratory Study. *Proceedings of the 23rd International Conference on Software Engineering*, 275–284.

Mussbacher, G., Whittle, J., and Amyot, D. (2009). Semantic-Based Interaction Detection in Aspect-Oriented Scenarios. *2009 17th IEEE International Requirements Engineering Conference*, *1*, 203–212. https://doi.org/10.1109/re.2009.13

Noda, N., and Kishi, T. (2008). Aspect-oriented Modeling for Variability Management. *2008 12th International Software Product Line Conference*, 213–222. https://doi.org/10.1109/splc.2008.44

Odersky, M., Spoon, L., and Venners, B. (2008). *Programming in scala*. Artima Inc.

Oh, H.-S., Kim, B.-J., Choi, H.-K., and Moon, S.-M. (2012). Evaluation of Android Dalvik virtual machine. *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems - JTRES '12*, 115. https://doi.org/10.1145/2388936.2388956

Olson, M. A., Bostic, K., and Seltzer, M. I. (1999). Berkeley DB. *USENIX Annual Technical Conference, FREENIX Track*, 183–191.

Osmani, A. (2012). *Learning JavaScript Design Patterns: A JavaScript and jQuery Developer's Guide*. " O'Reilly Media, Inc."

Paulisch, F., Bosch, J., Pohl, R., Höchsmann, M., Wohlgemuth, P., and Tischer, C. (2018). *Variant management solution for large scale software product lines*. 85–94. https://doi.org/10.1145/3183519.3183523

Pleumann, J., Yadan, O., and Wetterberg, E. (2002). *Antenna: An Ant-to-End Solution For Wireless Java*. http://antenna.sourceforge.net/wtkpreprocess.php

Pohl, K., Böckle, G., and Linden, F. J. van der. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag.

Prehofer, C. (2006). Feature-oriented programming: A fresh look at objects. *Lecture Notes in Computer Science*, 419–443. https://doi.org/10.1007/bfb0053389

Przybylek, A. (2018). An Empirical Study on the Impact of AspectJ on Software Evolvability. *Empirical Softw. Engg.*, *23*(4), 2018–2050. https://doi.org/10.1007/s10664-017-9580-7

pure::variants. (2021). *pure::variants User's Guide*. https://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf

Reynolds, A., Fiuczynski, M. E., and Grimm, R. (2008). On the feasibility of an AOSD approach to Linux kernel extensions. *Proceedings of the 2008 AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software - ACP4IS '08*, 8. https://doi.org/10.1145/1404891.1404899

Robillard, M. P., and Murphy, G. C. (2007). Representing Concerns in Source Code. *ACM Trans. Softw. Eng. Methodol.*, *16*(1), 3–es. https://doi.org/10.1145/1189748.1189751

Rubin, J., Kirshin, A., Botterweck, G., and Chechik, M. (2012). Managing forked product variants. *Proceedings of the 16th International Software Product Line Conference on - SPLC '12 -Volume 1*, 156–160. https://doi.org/10.1145/2362536.2362558

Saidani, I., Ouni, A., Mkaouer, M. W., and Palomba, F. (2021). On the impact of Continuous Integration on refactoring practice: An exploratory study on TravisTorrent. *Information and Software Technology*, *138*, 106618. https://doi.org/10.1016/j.infsof.2021.106618

Schaefer, I., Bettini, L., Damiani, F., and Tanzarella, N. (2010). Delta-Oriented Programming of Software Product Lines. *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, 77–91.

Schaefer, I., Rabiser, R., Clarke, D., Bettini, L., Benavides, D., Botterweck, G., Pathak, A., Trujillo, S., and Villela, K. (2012). Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer*, *14*(5), 477–495. https://doi.org/10.1007/s10009-012-0253-y

Schärli, N., Ducasse, S., Nierstrasz, O., and Black, A. P. (2003). Traits: Composable Units of Behaviour. *ECOOP 2003 – Object-Oriented Programming*, 248–274. https://doi.org/10.1007/978-3-540-45070-2_12

Schmidt, D. C. (2006). Guest Editor's Introduction: Model-Driven Engineering. *Computer*, *39*(2), 25–31.

Schulze, S., Liebig, J., Siegmund, J., and Apel, S. (2014). Does the discipline of preprocessor annotations matter?: a controlled experiment. *ACM SIGPLAN Notices*, *49*(3), 65–74. https://doi.org/10.1145/2637365.2517215

Selic, B. (2006). Model-Driven Development: Its Essence and Opportunities. *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, 7–13. https://doi.org/10.1109/isorc.2006.54

Seltzer, M. I. (2007). Berkeley DB: A Retrospective. *IEEE Data Eng. Bull.*, *30*(3), 21–28.

Smaragdakis, Y., and Batory, D. (2002). Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *11*(2), 215–255. https://doi.org/10.1145/505145.505148

Soley, R., and others. (2000). Model driven architecture. *OMG White Paper*, *308*(308), 5.

Somé , S. S., and Lethbridge, T. C. (1998). Parsing minimization when extracting information from code in the presence of conditional compilation. *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, 118–125. https://doi.org/10.1109/wpc.1998.693328

Spencer, H., and Collyer, G. (1992). #ifdef Considered Harmful, or Portability Experience with C News. *USENIX Summer 1992 Technical Conference (USENIX Summer 1992 Technical Conference)*. https://www.usenix.org/conference/usenix-summer-1992-technical-conference/ifdef-considered-harmful-or-portability

Spinellis, D. (2008). A tale of four kernels. *2008 ACM/IEEE 30th International Conference on Software Engineering*, 381–390. https://doi.org/10.1145/1368088.1368140

SPL2go. (2011). Retrieved March 25, 2021 from http://spl2go.cs.ovgu.de/

Stahl, T., Voelter, M., and Czarnecki, K. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley Sons, Inc.

Stallman, R. M., and Weinberg, Z. (1987). *The C Preprocessor*. Free Software Foundation. http://gcc.gnu.org/onlinedocs/cpp/

Strüber, D., Anjorin, A., and Berger, T. (2020). Variability Representations in Class Models: An Empirical Assessment. *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 240–251. https://doi.org/10.1145/3365438.3410935

Sullivan, K., Griswold, W. G., Song, Y., Cai, Y., Shonle, M., Tewari, N., and Rajan, H. (2005). Information Hiding Interfaces for Aspect-Oriented Design. *SIGSOFT Softw. Eng. Notes*, *30*(5), 166–175. https://doi.org/10.1145/1095430.1081734

Tarr, P., Ossher, H., Harrison, W., and Jr., S. M. S. (1999). N degrees of separation: multi-dimensional separation of concerns. *Int'l Conf. Software Engineering (ICSE), IEEE Computer Society*, 107–119. https://doi.org/10.1145/302405.302457

Tërnava, X., and Collet, P. (2017). Tracing Imperfectly Modular Variability in Software Product Line Implementation. *16th International Conference on Software Reuse*, 112–120. https://doi.org/10.1007/978-3-319-56856-0_8

Thomas, D., and Barry, B. M. (2003). Model driven development: the case for domain oriented programming. *OOPSLA03: ACM SIGPLAN Object Oriented Programming Systems Languages and Applications Conference*, 2–7. https://doi.org/10.1145/949344.949346

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. (2014). FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, *79*, 70–85. https://doi.org/10.1016/j.scico.2012.06.002

Umple Grammar. (2021). http://grammar.umple.org

Umple Metamodel. (2021). http://metamodel.umple.org

Unified Modeling Language. (2017). https://www.omg.org/spec/UML/

Unity. (2021). https://unity3d.com/unity

VA, S. P. C. H. (1993). *Reuse-Driven Software Processes Guidebook. Version 02.00.03*. https://doi.org/10.21236/ada273644

Voelter, M., and Groher, I. (2007). Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. *11th International Software Product Line Conference (SPLC 2007)*, 233–242. https://doi.org/10.1109/spline.2007.23

Voelter, M., Siegmund, J., Berger, T., and Kolb, B. (2014). Towards User-Friendly Projectional Editors. *International Conference on Software Language Engineering*, 41–61. https://doi.org/https://doi.org/10.1007/978-3-319-11245-9_3

Warmer, J. B., and Kleppe, A. G. (2003). *The object constraint language: getting your models ready for MDA*. Addison-Wesley Professional.

Zhang, B., Duszynski, S., and Becker, M. (2016). Variability Mechanisms and Lessons Learned in Practice. 2016 IEEE/ACM 1st International Workshop on Variability and Complexity in Software Design (VACE), 14–20. https://doi.org/10.1145/2897045.2897048