Copyright Undertaking

This thesis is protected by copyright, with all rights reserved.

By reading and using the thesis, the reader understands and agrees to the following terms:

1. The reader will abide by the rules and legal ordinances governing copyright regarding the use of the thesis.
2. The reader will use the thesis for the purpose of research or private study only and not for distribution or further reproduction or any other purpose.
3. The reader agrees to indemnify and hold the University harmless from and against any loss, damage, cost, liability or expenses arising from copyright infringement or unauthorized usage.

IMPORTANT

If you have reasons to believe that any materials in this thesis are deemed not suitable to be distributed in this form, or a copyright owner having difficulty with the material being included in our database, please contact lbsys@polyu.edu.hk providing details. The Library will look into your claim and consider taking remedial action upon receipt of the written requests.

Pao Yue-kong Library, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong

http://www.lib.polyu.edu.hk

The Pathway to Customised Product Solutions

Yutian Tang

PhD

The Hong Kong Polytechnic University

2018

The Hong Kong Polytechnic University
Department of Computing

The Pathway to Customised Product Solutions

Yutian Tang

A thesis submitted in partial fulfilment of the requirements

for the degree of Doctor of Philosophy

November 2017

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

______________________________ (Signed)

Yutian TANG (Name of student)

Abstract

Software Product Line Engineering (SPLE) allows variants of a software system to be created with customised configurations. However, building a software product line from scratch could be an error-prone task, given the complexity of the task and lacking of domain knowledge. To reduce the complexity of the task and provide a complete procedure to build a software product line, in this thesis, we proposed a strategy to build a software product line from a legacy application. Specifically, we focus on Java based applications. In a Java based software product line, it consists of a configuration space and a code space. The configuration space shows how features are organised and relations between features. Features and the constraints between them are commonly documented in a feature model. The code space contains the implementation of the system. In this thesis, we build a software product line from a legacy application by the following steps: (1) we detect and build the feature model from the implementation by building a variability-aware module system; (2) we further propose a conditional probability based approach to locate features in the legacy system; and (3) at last, we propose a strategy to build variant applications.

For (1), we carefully analyse previous research in building product line feature model and several representative approaches for recovering software architecture. We complement these research which are not suitable for building the feature model, by proposing a novel approach to build variability-aware module system. The programming elements are first composed to be variability-aware modules, then these modules are merged to compose features. We additionally analyse and extract programming

elements and relations between them to build well-typed modules and features. Our work shows that current approaches in software architecture recovery could be used in the product line context. We build a prototype tool LoongFMR, which is an Eclipse plugin to assist the process. Our work also indicates that a fine-granularity approach to build a feature model could significantly improve the performance comparing to a coarse-granularity approach. Since at the fine-granularity level, the programming elements are presented as abstract syntax tree (AST) nodes, and relations between them can be easily extracted.

For (2), we develop a novel approach to locate feature in the legacy system. To overcome the main limitations in other works, which cannot locate feature at a fine-granularity and cannot well-expressed the relation between programming elements, we propose a feature location technique using conditional probability. As demonstrated in the case study, our approach could locate the feature correctly with a performance of 83% for precision and 41% for recall.

For (3), we further propose an effective approach to reengineer an annotated legacy, which is the output of (2), to product variants. We also ensure the product variants generated are syntactical correct, well-typed and feature behaviors well-preserved. The results demonstrate that our approach maintains feature consistency. Concretely, our approach could reach an accuracy of 88% in terms of creating product variants without any syntactic errors. As for behaviour preservation, our approach could pass 93% test cases generated by *evosuite* tool.

Overall, we provide tools and techniques to help developers create software product lines by reusing legacy applications with ease. Our work helps to ensure the product variants created are well-typed and behaviors preserved during the process.

Publications

1. **Yutian Tang**, Xiapu Luo, Ting Chen, and Hareton Leung, "Towards Feature Persistence: From an Annotated Legacy to Product Variants", in *IEEE Transactions on Software Engineering* (under review), 2017.

2. **Yutian Tang** and Hareton Leung, "Constructing Feature Model by Identifying Variability-aware Modules", in *Proceedings of 25th IEEE International Conference on Program Comprehension (ICPC)*, pp 263-274, 2017.

3. **Yutian Tang** and Hareton Leung, "StiCProb: A Novel Feature Mining Approach Using Conditional Probability", in *Proceedings of 24th IEEE Internaltional Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp 45-55, 2017.

4. **Yutian Tang** and Hareton Leung, "Feature Mining for Product Line Construction", in *The First International Conference on Advances and Trends in Software Engineering*, pp 29-33, 2015.

5. **Yutian Tang** and Hareton Leung, "Top-down Feature Mining Framework for Software Product Line", in *Proceedings of International Conference on Enterprise Information System (ICEIS)*, pp 71-81, 2015.

Acknowledgements

It is hard to believe that this day has finally come. I know I would not have reached this stage if it was not for the help, support, and guidance of many great people who I was lucky to have as part of my life.

My Ph.D. advisors, Dr. Hareton Leung and Dr. Daniel Xiapu Luo, definitely make the top of the list. Hareton was always generous with his time providing me the guidance I needed and giving me freedom to pursue my research interests. Also, I was inspired by Daniel Luo, Daniel always provides useful tips and comments to help me think differently on the problem. I really miss the blackboard discussions and drawings with Daniel and Hareton.

I would like to thank my parents for their support in my education when I was a kid. My parents always offer the best they can in my education. Thank you for supporting me in my education. Also, I would like to thank my friends in Hong Kong and Mainland China. Thanks for their help, understanding and support in my life and help me get out of some tough days.

Last but not least, I would like to thank my wife, Lisa Huang. You have been my partner during my last year of Ph.D. study. Thanks for always being there, for taking care of me during my Ph.D. study and daily life. I hope we can have a pleasant journey for the rest of our lives.

Table of Contents

List of Figures **xvii**

List of Tables **xix**

1 Introduction **1**

1.1 Software Product Line and Traditional System 1

1.2 Benefits of Adopting Software Product Line 2

1.3 Feature, Feature Model and Variability 5

1.3.1 Feature 5

1.3.2 Feature Model and Feature Relations 6

1.3.3 Variability, Variant Point and Variant 8

1.3.4 Example 9

1.3.5 Product Variant 10

1.4 Building a Software Product Line from a Legacy System 10

1.4.1 Feature Model Construction 11

1.4.2 Feature Mining 13

1.4.3 Reengineering Features into Product Line Variants 14

1.5 Contribution 15

1.6 Thesis Organization 16

2 Literature Review **17**

2.1 Building a Software Product Line 17

2.2 Building the Feature Model 18

2.2.1 Feature Model Recovery Techniques 18

2.2.2 Software Architecture Recovery Techniques 20

2.3 Mapping features with their implementations 23

2.3.1 Feature Location 24

2.3.2 Asset Mining 28

2.3.3 Feature Mining Tools 28

2.4 Refactoring an annotated legacy application into product variants . . 29

2.4.1 Feature Oriented Reengineering 29

2.4.2 Aspect Oriented Refactoring 30

2.4.3 Reengineering Approaches. 30

2.5 Chapter Summary 31

3 Feature Model Construction **33**

3.1 Overview 34

3.2 Module Modeling 35

3.2.1 Feature and Module 35

3.2.2 Module without Variability 36

3.2.3 Module with Variability 40

3.2.4 Module Constraints 45

3.3 Variability-aware Program Dependency Graph (*varPDG*) 47

3.3.1 Building *varPDG* 47

3.3.2 Tracing Options with Pointer Analysis 48
3.4 VMS Feature Model Recovery Approach 51
3.4.1 Overview 51
3.4.2 Build Module from Source 52
3.4.3 Module to Feature 54
3.4.4 VMS 59
3.5 Case Studies 61
3.5.1 Experimental Settings 61
3.5.2 Subject Systems 62
3.5.3 Tools 64
3.6 Experimental Result 64
3.6.1 Related Approaches 64
3.6.2 Results 66
3.7 Discussion 72
3.7.1 Lessons Learned 72
3.7.2 Threats to Validity 73
3.8 Chapter Summary 74
4 Feature Mining **75**
4.1 Overview 75
4.2 Feature Mining Process Overview 77
4.3 Underlying Model 78
4.3.1 Basis 78
4.3.2 Modeling Closeness between Element and Feature 81

4.3.3 Modeling Closeness between Elements 85
4.4 *StiCProb* Approach 91
4.4.1 Selecting Seeds 92
4.4.2 Building a Uniqueness Table 93
4.4.3 *StiCProb* 97
4.4.4 Stopping Criteria 101
4.5 Case Studies 101
4.5.1 Experimental Settings 101
4.5.2 Subject Systems 102
4.5.3 Tools 103
4.6 Experimental Result 103
4.6.1 Related Approaches 103
4.6.2 Results 105
4.7 Discussion 108
4.7.1 Seeds 108
4.7.2 Threshold 109
4.7.3 Threats to Validity 109
4.8 Chapter Summary 110
5 Reengineering Features into Product Line Variants 111
5.1 Overview 111
5.2 Motivating Examples 115
5.2.1 Syntax Error Example 115
5.2.2 Behaviour Inconsistent Error Example 116

5.2.3 Type Error Example . . . 117

5.3 Configurable AST: Outline and Background . . . 118

5.3.1 Procedure At A Glance . . . 118

5.3.2 Process Modelling . . . 120

5.3.3 Transforming a Configuration into Operations on AST . . . 121

5.3.4 From t to $\Delta(t)$ . . . 122

5.4 Configurable AST: Syntactical Correctness . . . 124

5.5 Configurable AST: Behaviours Preserving . . . 127

5.5.1 Assumption . . . 128

5.5.2 Control Flow Constraint . . . 128

5.5.3 Data Flow Constraint . . . 131

5.5.4 Name Binding Constraint . . . 133

5.5.5 Context-sensitive Constraint . . . 134

5.5.6 Putting All Pieces Together . . . 136

5.6 Configurable AST: Type Checking . . . 136

5.6.1 $\phi_{\mathbf{T-VAR}}$ . . . 137

5.6.2 $\phi_{\mathbf{T-FIELD}}$ . . . 137

5.6.3 $\phi_{\mathbf{T-INVK}}$ . . . 138

5.6.4 $\phi_{\mathbf{T-NEW}}$ . . . 138

5.6.5 $\phi_{\mathbf{CAST}}$ . . . 139

5.6.6 $\phi_{\mathbf{METHOD}}(M_c \text{ OK in } C_c)$ . . . 140

5.6.7 $\phi_{\mathbf{CLASS}}$ . . . 140

5.6.8 Putting All Pieces Together . . . 142

5.7 Configurable AST: Feature-effect Constraints . . . 142
5.8 Configurable AST: Algorithm . . . 143
5.8.1 Putting all pieces together . . . 143
5.8.2 From Annotated Legacy to Product Line . . . 143
5.9 Case Studies . . . 145
5.9.1 Experimental Settings . . . 145
5.9.2 Subject Systems . . . 145
5.10 Experimental Result . . . 147
5.11 Discussion . . . 156
5.11.1 Issues Studied . . . 156
5.11.2 Threats to Validity . . . 158
5.12 Chapter Summary . . . 159
6 Conclusion **161**
6.1 Summary of Contribution . . . 163
6.2 Future Work . . . 163
References **167**

List of Figures

1.1 A sample feature model . . . 7

3.1 An example of *varPDG* . . . 46

3.2 The core idea of VMS approach . . . 52

3.3 The violin plot for MoJo Similarity . . . 67

3.4 The Heat Map for Cluster-to-cluster Coverage . . . 69

3.5 The violin plot for a2a Measurement . . . 71

4.1 Feature mining process overview . . . 77

4.2 Example program P with its control dependency graph, data dependency graph, program dependency graph, and the slicing scope for node 3. . . . 80

4.3 An example of feature model . . . 81

4.4 The general process of feature annotation . . . 83

4.5 The general process of change on annotation state . . . 84

4.6 Example of context transformation in method invocation . . . 87

4.7 Example of context binding in overriding . . . 88

4.8 A sample code of overriding . . . 90

4.9 The architecture of FLAT^3 . . . 93

4.10 An example of a call relation . . . 94

4.11 Illustration of *StiCProb* 97
4.12 Performance Comparison on Subject Systems 106
4.13 Method comparison using notched box plot in recall 106
4.14 Method comparison using notched box plot in precision 107
5.1 The process of transforming annotated legacy to product variants . . 114
5.2 A syntax error example 115
5.3 A behaviour inconsistent example 116
5.4 A type error example 117
5.5 The configuration sample 119
5.6 The overview of changing process 121
5.7 The example of code-based reengineering 122
5.8 If-else statement 130
5.9 For statement 130
5.10 Switch statement 131
5.11 Error types of LJ^{AR} approach 149
5.12 No Return Error 151
5.13 Unreachable Code Error Example 151

List of Tables

3.1 Notation of Module Modeling without Variability 36

3.2 Notation of Module Modeling with Variability 41

3.3 Points-to approach 49

3.4 Syntax of VMS 53

3.5 Evaluated MoJo Similarity 67

3.6 Evaluated Project and architecture with a2a 68

3.7 Cluster-to-cluster coverage(majority match(50%), moderate match(33%),weak match(10%)) 70

3.8 Runtime Performance in second 71

4.1 Uniqueness Table: e 96

4.2 StiCProb Performance with $threshold = 0.6$ 104

4.3 $f - measure$ on all approaches 106

4.4 Runtime Performance (second) 107

5.1 Valid AST Fragments Rules 125

5.2 Error collection and statistics 148

5.3 Behaviour preserving test and performance 155

Chapter 1

Introduction

Considering software as a composition of features and services can be deemed as one of the most important shifts in thinking on our road to massive customization of software. Software Product Line Engineering (SPLE) is deemed as a main approach to provide a series of products within the same domain to achieve massive customization [78]. It allows developers to reuse software assets within the application systematically. This unique characteristic makes product line itself easier to use and reduce the maintenance effort in extending the products.

1.1 Software Product Line and Traditional System

With the increasing popularity of traditional software (desk application), mobile application, cloud service and so forth, software users are increasing rapidly. However, there are two things are not fully satisfied currently.

1. Although, the end-user brought an application or a mobile app, (s)he might not need all services, but (s)he still has to pay for the whole package;

2. Individual systems are rather expensive and standard application packages often lack of diversification;

Customers are not content with standardized product modules and like to assemble modules to create their own applications, since different customers have different concerns. As a result, software product line engineering is a solution for this demand.

Software Product Line Engineering (SPLE) is defined as the engineering of managing a portfolio of related products, which are in the same domain, sharing a set of software assets and containing unique features for tailored service [78]. It is also regarded as an effective means for constructing software products within the same domain that contains multiple customised assets [78].

In software product line engineering, the means of producing software applications have been changed significantly in favors of customization, personalization, and mass needs on software and services. In summary, there are two conflicts: (1) customers bought the functions they do not need; and (2) products lack sufficient customizations and often costly.

1.2 Benefits of Adopting Software Product Line

Successful adoptions of product lines could assist stakeholders provide applications with low costs, fast time to market and high quality, since code and design are highly reused in each variant within a product family[78, 41].

1. **Providing tailored services to customers.** The key attribute of SPLE, is providing mass customised applications to customers. This is the main contribution and the motivation of adopting SPL. That is, SPL just provides what

users ask for. Previously, the applications are mainly design for two situations: (1) customers provide the requirements, then developers build the application based on the requirements; and (2) developers define the domain, and then provide the software for a group of customers. For example, a software company develops a personal task management application for people who need it. However, these two directions suffer from a lack of customisation and difficulty in responding to change. As a result, it will reduce the product life cycle. Accordingly, software packages are developed to support common functions and unique services are provided to different types of customers. Moreover, this allows customers to purchase these application packages at a reasonable price and also receive specific the functions they really need;

2. **Reduce the development costs.** SPLE reduces the development costs by reusing software artefacts. In a SPL, features are roughly grouped into *common* features and *optional* features. *Common* features are shared in all product variants in the product line. Investments are necessary for defining *common* features and creating them. Once these features are created, they will be shared and used in all products. This means that each product reuses the *common* features and can reduce the cost per system. For example, if there are two *common* features, we can create ten product variants, with each product contains these two features. It saves 18 development units, if we consider each feature as one development unit;

3. **Reduce time to markets.** With the development of agile programming and its tools, providing applications that can response to timely demand is critical.

That means an application its self should be organised well. Apparently, this can be achieved by having a well-designed architecture. However, with the increase of components in the architecture, itself becomes more complex and hard to maintain;

4. **Enhance the quality of applications.** Software artefacts in the product line are repeatedly reviewed and tested as, the testing effort needed is system testing for a product variant to the combination of artefacts in the variant rather than testing each artefact. For example, given a product contains artefact a and b, and another product contains artefact b and c. Then, we only need to test three artefacts in total rather than four. Therefore, the quality assurance of the software product implies a higher chance of detecting defects in the product family when compared to testing each product individually;

5. **Adaptable to changes and evolutions.** The change on the existing version will lead to a new version, especially when adding new artefacts into the platform or changing some components. Another similar case is that developers want to add new features into the system. Software product lines could cope with the demand by adding new optional features to the platform and modifying existing ones without changing the core features; and

6. **Cope with the complexity of the system.** With the increasing demand of adding new features, the complexity of a traditional software will increase. However, in software product line, features are well modularized and organised in the feature model. Adding new features will increase complexity, however, since common features are reused and shared among all products within the

product family, the overall complexity is not increase that much comparing to adding features to individual products.

1.3 Feature, Feature Model and Variability

1.3.1 Feature

In product line engineering, features are used to describe all behaviors of a system [27]. For instance, a business transaction system is normally customized to realize different banking services in various currencies, with each service deemed as a feature. A feature in the system could be considered as a system property that is defined by stakeholders and used to represent functions concerned by stakeholders or discriminate the systems within the same domain [28]. Furthermore, features could be "user-visible" properties for all types of stakeholders, including developers, customers, managers, architects, administrators and so forth. As for its source, features could be collected from different perspectives in terms of capabilities, domain technologies, implementation specification, environments and so forth. Specifically, the capabilities are the functions, which are visible to end users, especially, those services that directly used by end users. For example, in mobile phone product line, the feature "call a number" should be a feature that is visible for users to contact a person. Also, the network type could be 2G, 3G, and 4G. For the mobile product, whereas, the network type for PCs and laptops could be wireless network or cable network. Therefore, features are highly dependent on the domain. Implementation techniques distinguish the techniques used to implement domain functions. Operating environment shows the environments in which the applications are executed.

As for the type of a feature, in the *feature oriented domain analysis* (FODA) feature diagram notation, a feature could be *mandatory*, *optional* or *alternative* [45].

Mandatory features are the part of application in the exactly same form. Mandatory feature is also known as *common feature*. For example, in the banking system product line, the feature "view account" is a common feature, since no matter what kind of banking system, it must contain this function to allow users to check their accounts.

For a given parent feature, it often contains a group of sub-features. If any number of sub-features could be selected in a variant, then these sub-features are called *optional* feature. If exactly one feature must be selected from the sub-feature group, then the feature is called *alternative* feature.

1.3.2 Feature Model and Feature Relations

Feature model shows the external visible functions and characteristics in the domain and these functions are organised in a feature model [60]. The main concern in a feature model is how features are organised and relations or constraints between features. Normally, a feature model is represented in a hierarchical view as the example shown in Fig. 1.1.

The relations and constraints in a feature contains the following types: *mandatory*, *optional*, *or*, *alternative(xor)*, *implies* and *mutually exclusive*. Specifically, the relations mandatory, optional, or and alternative(xor) describe the relationship between a parent feature and its child feature. The other two relations, *implies* and *mutually exclusive*, show the cross-tree relations.

The *mandatory* relation exists between two features, represented as $f_1 \Leftrightarrow f_2$,

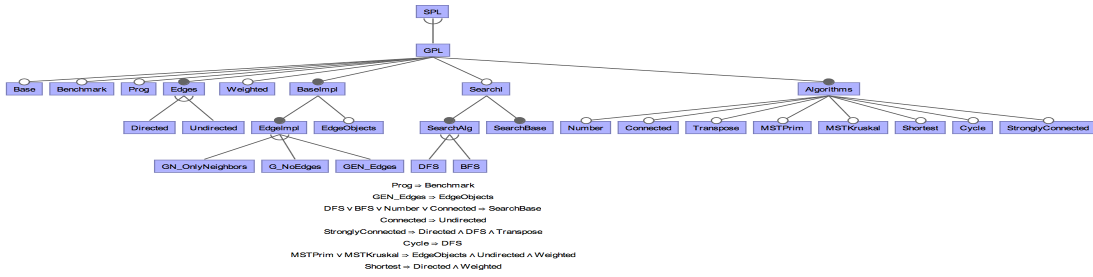


Figure 1.1: A sample feature model

if a child feature f_2 is a mandatory sub-feature of feature f_1. That means, in a software product line, if feature f1 is selected, feature f_2 must also be selected. From a product perspective, if f_1 exists in this product, we can assert that f_2 must also exist in this product. Distinguish from mandatory(common) feature, if a feature is *mandatory (common)* feature, it must be contained in all variants. However, the mandatory relation only shows the relationship between two features. For example, if child feature f_2 is a mandatory sub-feature of feature f_1, f_2 may not be a mandatory feature, since f_1 may not be selected in some variants.

The *optional* relation between parent and child features shows that feature f_2 is an optional sub-feature of feature f_1 as represented by $f_2 \Rightarrow f_1$. If feature f_2 is selected, its parent feature f_1 must be selected. The opposite may not be true as when f_1 is selected, f_2 could be discarded.

The relation *or* shows that for a parent feature, there is at least one of its child features must be selected as represented by $f_1 \vee \ldots \vee f_n \Leftrightarrow f$. That is, for parent feature f, there is at least one feature from its child features $(f_1 \ldots f_n)$ must be selected. In addition, the relation xor is a special case of or, where it denotes the case that a parent feature f contains only two sub-features f_1 and f_2; and at least

one sub-feature is selected.

The *implies* relation describes the cross-tree constraint between two features f_1 and f_2, where f_1 and f_2 do not need to follow a "child-parent" relation. It means that the selection of a feature f_1 implies the selection of another feature f_2 represented by $f_1 \Rightarrow f_2$.

The *mutually exclusive* relation between two features f_1 and f_2 represents that both feature f_1 and f_2 cannot appear together in a product. It is represented as $\neg (f_1 \vee f_2)$.

1.3.3 Variability, Variant Point and Variant

The term "variability" itself is used to describe the capability to change. An object with high variability means it has a large tendency to change to another object. As for our context, variability in a software product line represents the ability of creating customised applications by reusing predefined artefacts. For example, a banking system could decide which currency to support.

Variability Subject: A variability subject is a variable item of a real world or an attribute of such an item. The variability subject could be a category or a variable, which contains a set of variability objects. For example, the "network type" is a variability subject, since it contains different instances, including 2G, 3G, and 4G.

Variability Object: A variability object is a specific instance of the variability subject. Followed the network type example, a specific type, like 2G, is a variability object. In SPLE, variability subjects are not independence objects, because they are associated with context information, which comes from the domain and the particular

software product line.

Variant Point: A variant point represents a variability subject within the domain artefacts with the context information. The variant point could be all kinds of artefacts, including requirement specifications, code assets, test cases, architectures and so forth. The context information describes the detail of this variant point, such as, how this variant point is introduced and the type of this variant point. Especially it introduces the relation with other variant points.

Variant: A variant is variability object within the domain artifact. The variant in SPL is a concept that highly associated with variant point. That is, a variant is a concrete option for a variant point and it could be associated or interact with other variants to compose a particular configuration for the product line.

1.3.4 Example

To further illustrate these concepts, we use an example of variants, variant points, and features from a feature model. All these subfeatures are variants. Since Fig.1.1 is a feature model, all entities are features. The variant could be the code and other artefacts. As Fig.1.1 shown, the "algorithms" is a variant point, which contains several attributes and algorithms, including *number*, *connected*, *transpose*, *MSTPrim*, *MSTKruskai*, *shortest*, *cycle*, and *stronglyconnected*. However, the variant has a larger scope, as it could be code base, requirement specification, architecture, and other related software artefacts. For example, the variant number contains the code fragments that implement feature *number*, the architecture of feature *number*, the test cases of feature *number* and other artefacts.

1.3.5 Product Variant

A product variant is different from variant. The product variant is a software product within a software product line (or product family), whereas the variant is a component in the software product line. More specific, a product variant is an instance of a software product line and a variant is part of the software product line.

1.4 Building a Software Product Line from a Legacy System

To construct a software product line, developers could start from building individual modules and then compose them to a product line. However, this approach seems not practical especially for the following cases: (1) the total development effort could be high since it starts from scratch; and (2) since product lines allow developers to reuse some components, software assets, and architectures, therefore, it might be hard to figure out the reusable parts at the beginning stage of the software lifecycle. As a result, developers turn to an alternative approach of migrating a legacy application to a product line.

To compose a product line by migrating a legacy application, the developers should have a good overview of the legacy system (understanding the architecture of the system), understand the code structure and organisation within the system (code dependency and mapping code to architecture), and use an effective strategy to reengineer the legacy into a product line. In general, the procedure of building an SPL from a legacy application contains three steps: (1) constructing the feature model; (2) mapping the features in the feature model with their implementations (a.k.a feature

mining); and (3) reengineering the features into different product variants.

1.4.1 Feature Model Construction

To migrate a legacy application into a product line, the first step is to understand the legacy system by creating the feature model. To build a feature model, normally a domain expert identifies features and describes the underlying dependencies and constraints. The domain expert should define the features required in the product line and the dependencies and constraints from hierarchical and cross-tree viewpoints respectively.

The work on constructing a feature model for migrating a legacy system into a product line system is essential in a lot of cases. For example, if the domain expert is not available, or if the system documents and other supporting materials are not reachable, it could be hard to build a feature model for a product line. Therefore, in this thesis, we first propose an approach to recover the feature model from the source code. We choose to start from the code base on two concerns: (1) regardless of the type of the system or the system domain, the source code is always available; and (2) sometimes, domain experts might make mistakes in identifying features, especially when defining the underlying constraints between features.

In summary, this part of work is defined as follows:

- **Input.** The input of a feature model building work is the pure code base.
- **Strategy.** A strategy is proposed to automatically parse and analyse the source code and their relations to create a feature model.
- **Output.** A feature model which describes the features and relations in the

product line.

Main challenges. The main challenges for building feature model from code base are:

1. Features are often scattered among the code base rather than embedded in specific modules, containers and so forth. Therefore, a suitable approach should be possible to find the relation at a fine-granularity level. Specifically, relations are extracted at the AST node level.

2. Feature model is not the same as architecture from the following key perspectives:

   (1) Different concerns: A feature model mainly describes the features in the systems. In practice, these features often represent modules to address the functional requirements. On the contrast, an architecture focuses more on both functional and non-functional requirements of the system.

   (2) Different granularities: An architecture describes relations at package, class or method level, whereas, the features, sometimes, are described in fine-granularity, like a field and an enum constant.

   (3) Different relation types and constraint types: The relations in architecture are used to show the relation from the program structure aspect. For example, class A implements interface I. Differently, the relations in the features model show the relations from the functional perspective. For example, feature F requires feature T, which means if feature F appears in a product variant, feature T must also be presented in this variant.

1.4.2 Feature Mining

Along with constructing the feature model, to construct a product line from a legacy system, the mapping between the feature and its implementation should be explored. The result can serve as the input to reengineering, since a product line is actually a series of products with different configurations. Therefore, to construct a product line, *Feature location* try to build the mapping between a feature and its implementation. This step aims at finding features' implementation in the source code, and we name this procedure as "feature mining". The resulting feature model can serve as an input to show the constraints and dependency relations of features.

In summary, this part of work is defined as follows:

- **Input.** The input for mapping features to code fragments should be: (1) a feature model; and (2) seed for each feature.
- **Strategy.** A feature mining approach is based on the feature model and the seed for each feature to extract features' implementation.
- **Output.** An annotated product line, in which codes are annotated with features.

Main challenges. The main challenges for feature mining are:

1. As the success of this task is highly influenced by the input seeds for features, providing a better approach to recommend seeds for features is essential.
2. The proposed feature mining should tackle fine-granularity requirement, as we observed that features are mostly implemented in fine-granularity. For

example, a field, a statement and an expression could be a feature. Therefore, a qualified feature mining approach should be able to analyse the program at the statement level.

3. Another challenge is how to describe the relationship between programming elements. To mine features from the code base, a key component should be an algorithm to depict the relation between two programming elements, since programming elements with high similarity should be considered belonging to the same feature.

1.4.3 Reengineering Features into Product Line Variants

The next step after feature location is transforming the annotated legacy into product line variants. Following the previous task, reengineering an annotated legacy into product variants will generate physical variants rather than separate concerns virtually. Separating concerns virtually means that features are only separated via annotations rather than physically into methods, classes or packages.

In summary, this part of work is defined as follows:

- **Input.** An annotated legacy system and valid configurations

- **Strategy.** A proposed approach should be able to generate a product variant with a given configuration.

- **Output.** Product variants based on valid configurations.

Main challenges. The main challenges for refactoring an annotated legacy into product variants are:

1. Transforming a legacy into product variants should not introduce any compilation errors, parser errors and type errors.

2. The transformation should be at a fine granularity and should be suitable for AST-based transformation. For example, in some cases, the arguments of a function could belong to different features, therefore, generating a variant might need rewriting the function.

3. Behaviour preserving should be considered during the reengineering procedure.

1.5 Contribution

In this thesis, we propose a series of procedures to semi-automatically migrate a legacy application to product variants. Specifically, our approach starts from the code base and ends with the code base. Except for domain knowledge and manual checking needed during the procedure, we do not rely on any other supports from the documentation, APIs and so forth. The contribution of our work contains following parts:

1. We propose a fine-granularity approach to build the feature model for a product line from code base.

2. We design a semi-automatic approach to map the features with their implementations to generate an annotated product line.

3. We further reengineer the annotated product line to physically separated product variants.

4. We develop a full set of type checking rules to ensure a software product line is type safe.

5. We create a variability-aware AST representation for modifying, rewriting, and updating AST with specific concerns.

1.6 Thesis Organization

The thesis is organized as follows. Chapter 2 presents a review of the related work In Chapter 3, we proposed an approach to build the feature model from the given system. Chapter 4 shows the work on mapping features to code fragments, which will guide developers to locate the code fragments for a specific feature. Chapter 5 follows the work in Chapter 4 and further physically reengineers the feature-annotated legacy into product variants. Chapter 6 summarises the thesis and gives the conclusion and future directions to investigate.

Chapter 2

Literature Review

We provide an overview on work related to our study of building an SPL from a legacy application. First, we describe the existing studies on variability modeling; second, we describe work on building and modeling feature models; and third, report on analysis techniques that allow developers to map features with their implementation. Finally, we describe the related work on reengineering an annotated legacy into product variants. We carefully anticipate some of our study to explain the relationship between our work and other discussed publications.

2.1 Building a Software Product Line

As reported in several surveys [40, 98, 11], there is little research on how to build a software product line exists especially for starting from a legacy system. Specifically, Batory [9] and Apel [8] applied *feature-oriented* programming to software artifacts which are not object oriented composed. This work is not be applied to most modern programming languages. Instead it mainly works as a modelling approach. The features in [8, 9] are separated at the beginning, which is not possible for the legacy

system. In practice, features will scatter in the system and feature interactions are more complex [49]. Schaefer et al [91] proposed *delta-oriented* programming to build software product lines. In delta-oriented programming, there is a core module, which is a standard application written in the host language, and a set of delta-modules. The delta-module can add and remove methods, classes or even change the superclass of an existing class. However, *delta-oriented* approach has two limitations: the work cannot suitable for fine-grained operations and the work requires features are well-separated at the beginning. For the first limitation, *delta-oriented* programming does not support fine-grained feature representation. For example, sometimes a statement may represents a feature. For this case, it cannot be implemented with *delta-oriented* programming. Therefore, it is not a suitable approach to cope with fine-grained task on building a software product line.

2.2 Building the Feature Model

Since we only rely on the code base to build the software product line, the feature model is also built based on the code base. Building a feature model is highly related to research work on architecture recover[21, 65], program understanding[83, 77, 25], feature identification [31] and other relative subfields.

2.2.1 Feature Model Recovery Techniques

Recover the feature model by analysing all products within the product family [108, 110]. Specifically, Yang's work [110] uses *data access semantics* and *formal concept analysis* to build the feature model. First, the data access semantics using database schema are used to build the database containing all methods in the pro-

gram and then the domain experts review the data model and build the mappings from the application data schema to domain schema using *data access semantics* , which specifies how methods access data. Then, all data semantics are merged iteratively to generate meaningful clusters for features. Xue's work [108] adopts a sandwich approach to recover the feature model. For bottom-up analysis, it uses a clone detection tool to find the cloned candidates. As for top-down analysis, a set of product feature models serves as input of this approach to extract features and relations. Then, these information are combined to build the feature model.

Recover the feature model by tracking the version change [109]. Xue's work [109] uses the clone detection approach to track the changes in different versions, then the changes are mapped to features.

Recover the feature model from requirement specifications[81, 18, 30]. Davril's work [30] extracts the feature model from a large collection of product descriptions. It uses a tool named SoftPedia to parse the description and recommend features. Then it uses *tf-idf* to provide weighting information to features. Then features are formalised and the feature model is built based on the clustering techniques. Stoiber et al's work [81] transfers the descriptions into constraints and then checks these constraints using a boolean satisfiability solving tool to build the feature model. Buhne et al's work [18] provides a meta model for extracting and organising the variability information from requirements and then all information collected to build the feature model. Yue et al's work [111] presents a reverse engineering approach to recover requirement from structured and unstructured code. It uses *extract method* to refactor legacy code, and then the refactored code is transformed into an abstract structured program. The goal model is built from the abstract structured program's

AST. Based on the relation and tracing between these ASTs, the feature model is extracted.

Other works [92, 1] require multiple input, like source code, requirement specification, even architecture information, rather than starting from a pure code base. For example, She et al's work [92] builds the feature model by identifying parent candidates for the given feature. Acher et al's work [1] supports the transition from descriptions to feature model in tabular format. It also relies on architecture knowledge to reinforce the extraction process.

2.2.2 Software Architecture Recovery Techniques

Software Architecture Recovery (SAR) techniques have been broadly used to rebuild the product architecture by collecting syntax and structural information from the system[24, 42, 55, 66, 101].

Granularity. Most architecture recovery techniques are implemented at the class and file level, which means they parse a single class file as an unit for recovery [32, 55, 66].

Methodology. For the target of reconstructing or learning architecture, some approaches deem it as an optimization problem[66, 55, 80], in which some objective functions, like modularity, fan-in, fan-out, are built and the architecture that satisfies and offers maximal or minimal target values are considered as solutions; some representative works use pattern searching along with users' instructions to find target patterns and rebuild the architecture by sequentially finding these components[87, 88]. In addition, adopting clustering algorithms to resolve architecture problem is another trend, since programming components, like class with similar function should

be grouped into the same cluster[6, 67]. Another direction to achieve the target is using textual information, namely natural language processing technique, like the strategies shown in [42, 24]. In the following paragraphs, we will introduce several representative approaches that are frequently used to abstract program architecture.

Corazza's work [24] groups the source file based on lexical information. In detail, it provides a vector representation of each file and weights for each is computed by EM algorithm based on a probabilistic model. The clustering algorithm is further applied to these vectors to identify the software clusters.

Garcia's work [42] uses information retrieval technique, in which a statical language model LDA is used to compute the similarity between programming entities. The clustering is provided by the tool Weka and different clusters are built based on the topics. That is, each cluster has a main topic and it tries to classify the documents into different topics, where a programming unit (.java file in the case study) is treated as a document.

Kobayashi et al.'s work [55] develops the SArF algorithm to build the architecture. Specifically, SArF accomplishes the task for clustering based on computing a dedication score between programming elements.

Mancoridis et al.'s work[66] regards the recovery task as an optimisation problem. It starts with a random partition and iteratively updates each cluster by optimizing the objective function called Modularization Quality (MQ) until it cannot find a better solution.

Algorithm for Comprehension-Driven Clustering (ACDC) recovers the architecture of system by inspecting certain patterns that could be presented in systems[101]. Specifically, ACDC contains *source file pattern*, *body-header pattern*, *leaf collection*

and support library pattern, and *ordered and limited subgraph domination*. ACDC identifies clusters by using these patterns with orphan adoption techniques, which were originally proposed in [100].

Maqbool's work [67] uses a set of measurement, including distance measurement (Euclidean distance, Canberra distance, Minkowski distance); association coefficients; and correlation coefficients to measure the distance between programming elements. Then, the architecture is built by conducting clustering on the entities.

Sartipi et al's work [87, 88] first represents the system as a source graph and the source graph is then separated into different sub-spaces, with each space representing a main-seed node. An AQL query is proposed and the graph matching engine will re-organize the sub-spaces to the pattern-graph presented in the AQL query.

Praditwong el al's work [80] studies two objective functions: Equal-size Cluster Approach (ECA) and Maximising Cluster Approach (MCA). The two objective functions are tested on both weighted and unweighted module dependency graphs to find a better solution by optimising the target function.

scaLable InforMation BOttleneck (LIMBO) optimizes the usage of information loss when conducting clustering on a system. It builds on *Information Bottleneck (IB)* framework and can collect relevant information during clustering[6].

Architecture Recovery using Concern (ARC) as defined in [42] uses a generative probabilistic model for text corpora named Latent Dirichlet Allocation (LDA) to retrieve concerns and identify programming elements belonging to concerns. It treats the source as a series of *documents* that contains various topics and then measures the similarity using the Jensen-Shannon divergence (D_{js}).

W-UE and W-UENM Weighted Combined Algorithm(WCA) combine hierarchi-

cal clusters into larger sets by computing the inter distance between two possible variants using Unbiased Ellenberg (UE) and Unbiased Ellenberg-NM(UENM) distance measurement respectively[67].

However, these architecture recovery techniques could not cope with variability issues in software product line and normally these approaches are only suitable for coarse-granularity. That is, they could only collect the relations between programming elements at the method or class level rather than at the statement level. In addition, current approaches do not consider type safety issue, which is a critical problem in building a software product line.

2.3 Mapping features with their implementations

Furthermore, features in the feature model should be mapped into the implementation. To build the mapping between the features and their implementation, the system will first be analyzed and a typical method involves *decomposing* a program. From a large scale, a program could be decomposed into modules and containers. Whereas, a small scale will separate the system into distinct functions, classes, and packages. The programming languages usually provide hierarchical decomposition of the program (eg. packages, classes/interfaces, methods, fields), which is not sufficient for the task of mapping features into their implementations. We observed that in a software product line, a feature is often scattered in the system rather than fitting in fixed classes or modules. Therefore, it would not be appropriate to use a package or a class to represent a feature.

2.3.1 Feature Location

Feature location in the software product line context is very different from the normal feature mining process, which aims to identify the location in the source base for a functionality concerned. The work in product line context should also take the variability and all underlying dependencies and constrains into consideration. The techniques used in feature location are mainly static analysis[14, 77], dynamic analysis[105, 25] and hybrid strategies[33].

Static feature location techniques

Static feature location techniques do not rely on the execution of the program. Instead, it extracts the relations from the code base directly. During the analysis, several types of dependencies are extracted, including data dependency, control dependency and so forth. They will be used as a profile of the system and the location techniques are applied then.

Chen and Rajlich's work [20] builds a dependency graph named *Abstract System Dependence Graphs (ASDG)*. ASDG is composed by nodes, which represent functions, global variables, fields, and edges. Together they indicate the control dependencies between functions and data flow between variables and function. The seed is a programming element that developers concern. Then, the engineer searches the graph and returns the nodes. Users' feedbacks on these nodes are collected to decide whether the process should continue. The process will be stopped as controlled by the users.

Robillard et al's work [84, 85] gives a presentation of program as a *Concern Graph*. The concern graph reorganises the programming elements into a set, which contains

programming elements and underlying relations. The concern (a feature in our context) is presented in an abstract way, in which it contains programming elements and relations. The concern graph is used to guide developers in the maintenance task.

Saul et al's work [89] (FRAN, *Finding with RANdom work*) is a random searching based approach starting with an input. It searches the programming elements associated with the current element under concern. Generally, it extends the work of[84], ranking the programming elements with scores, and the scores are computed by random walk algorithm upon the program dependency graph.

In addition, data flow is used in another study from Trifu [99]. Specifically, developers select a set of variables as input and then the variables will be propagated and tracked along data flows. During the tracking, developers are required to mark those programming elements interested.

Dynamic feature location techniques

On the contrast, dynamic feature location techniques rely on the execution of the system, during which the required information is collected. Several approaches have been proposed to deal with the feature location problem.

Wong et al.'s work[106] proposes an execution slice-based technique to capture features' implementation. This approach requires several test cases for each feature of interest. The dynamic information collected includes the statements executed. Then, the mapping between a feature and statements is built.

Eisenbarth et al.'s work[34] generates feature component maps from dynamic information. Specifically, it uses concept analysis to extract the relations between features and relations between features and programming elements as well.

Safyallah et al.'s work[86] uses a data mining approach upon execution traces. This approach allows execution patterns to be found in the execution traces. Then these patterns undergo a refinement upon the execution traces. As a result, a set of fragments of execution will be extracted to be components.

Bohnet et al[16] proposes an approach to visualise the characteristics of execution information on how features are implemented. The execution traces served as input and the visualisation is generated based on these. Views are created for executions, features, and interactions.

Textual-based mining

Besides relying on the program structure, some approaches treat the program from a textual prospective[90] or explore some dependency relations[14, 33, 69]. Here, we introduce two works [79, 7] that are highly related to our work using probability ranking.

Marcus et al.[68] uses LSI to map the words and features in the text. The corpus is built based on all identifiers and comments, which could be extracted from the source code of a system. Each document is built to be a vector based on the corpus. The feature is described as an input query. Then, the query is transferred to a vector, and the distance between the query vector and document vectors will be measured.

Poshyvanyk et al.'s work[79] combines formal concept analysis with the information retrieval techniques. Specifically, it first creates the corpus of the system, then the corpus is indexed and the vector space is created. The feature concerned is represented by a set of terms and they are then further translated into a vector by referencing the corpus. All documents are ranked by the similarity computed between

query and documents. From the top matched documents, the similar attributes(class, methods) are grouped into the implementation of the feature.

Cleary et al.'s work[23] uses information retrieval to locate features; however it mainly works on non-source artefacts, including mailing list, documentation, bug reports and so forth. Then the relations between source code and non-source artefact are built. The query first finds the non-source artefact, then the corresponding code artefacts are extracted.

Hill et al's work[43] proposes an approach using natural language (NL) queries. Specifically, it captures the nouns, verbs, and phrases from the method and field signatures. The NL phrase mapping will map the searching query with programming elements.

Combined approaches

Poshyvanyk et al.'s work[79] is essentially a combined approach of dynamic information from execution scenarios and textual information to locate features. Antoniol's work [7] combines dynamic and static data to identify the relevant methods. Different from these two approaches, which are merely suitable for methods rather than all types, our approach also contributes the fine granularity elements, like statement and local variable. Another difference between our approach *StiCProb* and these approaches is that the probability in these two approaches are obtained by tracing the runtime information and our approach collects probability statically. That is, these two approaches are dynamic approaches, while our approach is a static one, which collects information from the structure of the program.

2.3.2 Asset Mining

Feature mining is sometimes named *asset mining*[10, 12, 94]. The works in asset mining mainly recover variant relations and models by locating, documenting and analyzing features in the feature model. These works contribute to what to mine, and what should be considered in the process. They could be deemed as preliminary work to our contribution as **Step 1** in our process, and we replace the process by adopting existing feature models.

2.3.3 Feature Mining Tools

Two tools are closely related to our work: *LEADT*[50] and *CIDE+*[102]. In *LEADT* and *CIDE+*, the authors perform the same task of finding the feature code at a fine granularity. Our work basically contains all strategies in *LEADT* and adds our own *StiCProb* approach. On the contrast, the work in *CIDE+* mainly depends on type-checking-like mechanism assisted with Cerberus's dependency analysis[33]. Differently, in *CIDE+*, feature dependency is not explored, and *CIDE+* requires a large number of seeds to reach an acceptable performance.

Even though several approaches have been proposed to locate features in the system, the main problem of these coarse-granularity approaches in terms of constructing a product line is that they cannot recover a feature's implementation at a fine granularity. For example, inside a method, statements might belong to various features.

2.4 Refactoring an annotated legacy application into product variants

In the previous step, we annotate the legacy application with features. The last step should be refactoring an annotated legacy application into product variants. This procedure contains following sub-steps:

1. A mapping between features and AST nodes will be extracted from the legacy system.
2. A configuration is given to build a product variant. Features are categorised into two groups: the first group contains features that should be preserved in the variant and the second group should be removed.
3. Generate a series of actions on AST nodes for hiding unselected features in the configuration.
4. Generate the variant.

During the above transformation, the variant product generated should be checked in the following perspectives to ensure its quality: (1) syntactical correctness: the product variant created should not contain any syntax errors, (2) behaviour preservation: during the transformation, the behaviours of features should keep consistent, and (3) typing safety: the variant created should be well-typed.

2.4.1 Feature Oriented Reengineering

Feature oriented reengineering is also named as feature decomposition(for considering its purpose), targets on deriving feature in a program. Specifically, Liu's work[63]

builds a mathematical foundation to cope with the fact that features have distinct implementations in different variants based on the feature expression. Schulze et al.'s work[90] addresses the same problem but focuses on object-oriented approaches. Kästner et al.'s work[47] builds a model between physically and virtually separated features. The task involves refactoring based *Lightweight Java* and starts refactoring from the class level and gradually moves to the method level. Also, these works are either (1) lack of fine-granularity support[4, 56]; or (2) not sensitive to additional constraints from the code base [63, 90, 47]. We argue that they could not explore underlying constraints not covered in the feature model.

2.4.2 Aspect Oriented Refactoring

Aspect oriented refactoring is another close area of our work. Current efforts on aspect-oriented refactoring are mainly focusing on reeningeering #ifdef statements into aspects [2, 17] with the key concern of exploring the usage of preprocessor. Some approaches enforce discipline annotation without considering the optional feature problem[2, 17, 82].

2.4.3 Reengineering Approaches.

During the reengineering, the type-correctness[53, 54, 76, 61] and semantics preservation[53] are the main concern. However, changing a certain method, class or interface is entirely different from changing the operations in a system as what we do in this work, since the reengineering process in the legacy system will introduce additional cross-constraints (i.e. the constraints between programming elements) that need to be coped with. These approach could not keep features' behaviour consistent during

the reengineering.

2.5 Chapter Summary

Our literature review shows that building a software product line from a legacy application is a rich field with many techniques and formalizations. However, some major research issues in this field remain: building the feature model from the code base, mapping the features with their implementations at a fine granularity, and keeping features' behaviour consistent during reengineering them into product variants. Therefore, in the coming chapters of this thesis, we will describe our approaches to handle the limitations in current research and how our works fill the gap.

Chapter 3

Feature Model Construction

The traditional way of building a software product line is using either model-driven software development (MDSD) or aspect-oriented software development (AOSD). Specifically, MDSD[13] improves the way of developing software by capturing the features in a system. It targets at abstracting the knowledge, services, and functions in an application domain. MDSD is intended to improve the productivity by improving compatibility between systems. In MDSD, each module is developed and then the entire product line is composed by synchronising, combining and refining each module. AOSD[38, 39] focuses on modularising crosscutting concerns. That is, using AOSD in software product line development, features and concerns are sequentially added. That is, the software product line is built by composing software artefacts. Different from these approaches in building a software product line, we start from a legacy software. To build a software product line, the first step is to understand the legacy and construct the feature model for the legacy. This chapter describes our approach to build a feature model from the code base of a legacy system.

3.1 Overview

A feature model presents a diagram containing all features along with underlying dependencies and constraints[78]. Explicitly, the feature model can be recovered either from requirement specification[30] or code base. Thereby, recovering the feature model from legacy code is a primary step to construct a product line system from a legacy. This task is paramount for working on an open-source project considering the requirement specification is normally unavailable in that situation. Unfortunately, current work on recovering the feature model from the source mainly start from a collection of product variants instead of legacy[3]. To cope with this issue, we aim at providing a semi-automatic approach to explore legacy source code and construct a feature model for the system. A fully automatic approach is unrealistic since the user has to determine the needed features. Specifically, in this work, we target on Java, and it can be extended to most other object-oriented languages.

Motivation. Constructing feature model is highly related to research work on architecture recovery[21, 65], program understanding[83, 77, 25], feature identification[31] and other relative subfields. However, it is especially different from these works in the following aspects: (1) it recovers the architecture in a variability fashion, that means the variability should also be mined during the process; (2) apart from the hierarchical relations between features, dependencies and constraints should also be discovered; and (3) in architecture recovery, both functional and non-functional requirements are concerned, but in a product line only functional requirements are considered.

Our Contributions. The main contributions of this chapter are listed as follows:

- We define a variability-aware module system to ensure that each module is well-typed and interfaces for modules are well-defined; moreover, we give a set of definitions and a series of constraints to check whether a module is well-formed.
- We give a variability-aware representation of the program to further define a set of similarity measurement to assist in merging modules into features.
- We provide a comprehensive comparison with six representative approaches for architecture recovery by investigating four systems and compare performance from four aspects by constructing over **80** experiments.

This chapter is organised as follows. Section 3.2 shows our variability-aware module system. Furthermore, Section 3.3 defines a variability-aware program dependency graph to support our module system; our variability-aware system (VMS) approach is introduced in Section 3.4. Section 3.5 and 3.6 exhibit case study and experimental results respectively. We discuss the results in Section 3.7. We conclude this chapter in Section 3.8.

3.2 Module Modeling

3.2.1 Feature and Module

In this section, we will first introduce the module, which is an isolated unit, containing programming elements. To compose a product line, developers usually develop a module for each configuration option and by combining some modules, the variability of the product line can be well expressed. For example, in a database product line, it could contain a storage module *persist*, an XML module *xml*, a backup module (from

Table 3.1: Notation of Module Modeling without Variability

Notation	**Remark**
$e \in \mathcal{E}$	expressions
$t \in \mathcal{T}$	types
$x \in \mathcal{X}$	function names
$\Gamma \in \mathcal{X} \xrightarrow{p} \mathcal{T}$	import functions/imports
$\Delta \in \mathcal{X} \xrightarrow{p} \mathcal{T} \times \mathcal{E}$	function defintion
$m = (i, j, \Gamma, \Delta) \in \mathcal{M}; i, j \subseteq \mathcal{O}$	module

local history) *backup_local*, another backup module(from remote) *backup_remote* and core module for fundamental architecture *core*. Different systems are derived from the product line by combining different modules. For example, the combination of XML module, backup(local), storage and core module could be a product, represented as: *xml* · *backup_local* · *storage* · *core*. However, in the software product line, features are used. Note that "feature" is a different mean for representing the product line. A feature could be a module or a combination of modules, highly dependent on how features are defined by the domain expert. For example, the module *storage* could be a feature *f_storage*. Moreover, a backup feature *f_back* could contain two modules: *backup_local* and *backup_remote*. Specifically, in this chapter, we adopt Java as the target language for our approach and case study.

3.2.2 Module without Variability

Prior to defining a module with variability, here we first define a normal module without variability concern. Concretely, our module follows Cardelli's work on modularity system[19] and Kastner's work[52]. A module normally contains:

1. a set of imported declarations, in which functions used but are not defined in

this module are resolved.

2. a list of functions along with function bodies.

. The notation for module without variability is displayed in **Tab.**3.1, specifically,

1. the imports are represented by the partial mapping from names to types as $\Gamma \in \mathcal{X} \xrightarrow{p} \mathcal{T}$, where $\mathcal{X} \xrightarrow{p} \mathcal{T}$ represents the partial mapping from a function name to its type.

2. the function definition is a partial mapping from a function name to its function body, which is a combination of expressions with types as $\mathcal{T} \times \mathcal{E}$.

The interface of a module should be all exported functions from all functions declared in the module. However, in a module (abstract level), we do not distinguish the privacy of a module. The exported functions in a module could be decided by checking the signature of functions from the code base (implementation level). Here, we mainly discuss the composition of a module from an abstract level instead of the implementation level. Therefore, we can omit the discussion on private functions for now.

Furthermore, the signature of a module represents a list of functions exported and their binding types. Therefore, the signature is defined as

$$sigature : (\mathcal{X} \rightarrow \mathcal{E} \times \mathcal{T}) \rightarrow \mathcal{X} \times \mathcal{T} \tag{3.1}$$

Moving on, we will introduce the type checking system on module without variability. Logically, a module m is well-typed if all functions declared in m are well-typed in terms of imported contexts or under an empty context. That is, a function

is well-typed, if it is well-typed by itself, that means without the input context; or it is well-typed under the input context.

A well-typed module. First, we provide the type checking for a single module to determine whether a module is well-typed as (m, OK).

$$\frac{dom\,(\Gamma) \cap dom\,(\Delta) = \varnothing \qquad \Gamma \vdash \Delta}{(\Delta)\,OK}$$

Symbol $dom(\Gamma)$ represents the domain of Γ. When the domains of Γ and Δ do not overlap, which means that the function itself does not rely on the input context, and function Δ is well-typed under environment Γ, we could derive that the function is well-typed as $(\Delta)\,OK$.

In addition, when a function(Δ)'s typing environment requires extra signatures from imported functions, its type judgment could be represented as follows.

$$\frac{\forall x \in dom\,(\Delta)\,.\Gamma \cup signature\,(\Delta) \vdash e : t \qquad \text{where } \Delta\,(x) = (e, t)}{\Gamma \vdash (\Delta)\,OK}$$

First, for any function x from the function definitions in the module($dom\,(\Delta)$), with the overall context coming from the input context Γ and signature from Γ as $signature\,(\Gamma)$, we could always derive that e is under type t. Then, this relation could be found in function with a name x, as $\Delta\,(x) = (e, t)$. As a result, the functions in a module are well-typed under import context Γ as $\Gamma \vdash (\Delta)\,OK$.

Modules compatibility. Moving on, we discuss the process of composing two modules into one as $m = m_1 \cdot m_2$ along with the typing issues. To combine two modules yielding a new module as $\bullet : \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$, the following checking should be conducted:

1. To combine two modules into one: each module should be checked in isolation to ensure it is well-typed(mOK).

2. When two modules are well-typed, the imported context should be combined and any invalid imports be resolved. For example, module m_1 may import functions from module m_2, when merging m_1 and m_2, these cross-imports should be removed as invalid.

3. Also, we should check the module compatibility to determine whether the merge should be approved.

4. At last, these two modules are merged and a well-typed module generated. The process could be represented as: $(m_1, OK) \wedge (m_2, OK) \wedge (m_1 \div m_2, OK) \rightarrow (m_1 \cdot m_2)OK$. Where $m_1 \div m_2$ represents module m_1 is compatible with m_2.

To check whether two modules are compatible, we use the following judgements.

$$\frac{\begin{array}{c} dom\,(\Delta_1) \cap dom\,(\Delta_2) = \varnothing \\ \text{compatible}(\Gamma_1, \Gamma_2) \\ \text{compatible}(\Gamma_1, signature\Delta_2) \\ \text{compatible}(\Gamma_2, signature\Delta_1) \end{array}}{(\Gamma_1, \Delta_1) \div (\Gamma_2, \Delta_2)}$$

The compatibility of two modules involves the following checking:

1. First, the domain of functions from two modules should be independent ($dom\,(\Delta_1) \cap dom\,(\Delta_2) = \varnothing$), which mean a function cannot exist in both modules.

2. Second, the imported context should be compatible as: compatible(Γ_1, Γ_2).

3. Third, cross checking for imported context from module m_1 with the signature of functions in module m_2 as compatible$(\Gamma_1, signature \Delta_2)$.

4. Forth, cross checking for imported context from module m_2 with the signature of functions in module m_1 as compatible$(\Gamma_2, signature \Delta_1)$.

After passing these checkings, two modules should be compatible as $(\Gamma_1, \Delta_1) \div (\Gamma_2, \Delta_2)$.

In summary, the merging of two modules could be represented as follows.

$$\frac{\begin{array}{c}\Delta = \Delta_1 \cup \Delta_2 \\ \Gamma = \Gamma_1 \cup \Gamma_2 \setminus (signature\,(\Delta_1) \cup signature\,(\Delta_2))\end{array}}{m_1 \cdot m_2 = m}$$

Specifically, when two modules are compatible, they could be merged by (1) combining the functions declared ($\Delta = \Delta_1 \cup \Delta_2$); and (2) resolving the imported context by combining imported context from two modules $\Gamma_1 \cup \Gamma_2$ and removing those imports found in m_1 and m_2 as
$(signature\,(\Delta_1) \cup signature\,(\Delta_2))$.

3.2.3 Module with Variability

As shown in **Tab.**3.2, a variability-aware module is a five-tuple $(v, i, j, \Gamma, \Delta)$ mainly inspired by Christian's work [52] and the calculus follows Cardelli's module system formalization [19]. v and i represent two sets of configuration options imported from other modules, j is defined in this module, Γ is the import contexts and Δ represents functions. We will introduce how to build a module from source in detail

[1] i: configuration options imported, j: configuration options defined

Table 3.2: Notation of Module Modeling with Variability

Notation	Remark
$e \in \mathcal{E}$	expressions
$t \in \mathcal{T}$	types
$x \in \mathcal{X}$	function names
$o \in \mathcal{O}$	configuration options
$c \in \mathcal{C} = 2^{\mathcal{O}}$	configurations
$v \in \mathcal{V} = 2^{\mathcal{C}}$	variability model
$\Gamma \in \mathcal{C} \overset{p}{\to} \mathcal{X} \overset{p}{\to} \mathcal{T}$	import functions/imports
$\Delta \in \mathcal{C} \overset{p}{\to} \mathcal{X} \overset{p}{\to} \mathcal{T} \times \mathcal{E}$	function defintion
$m^v = (v, i, j, \Gamma, \Delta) \in \mathcal{M}^{\mathcal{V}}; i, j \subseteq \mathcal{O}$	module[1]
$\phi_{DUC \wedge TC}\,(m^v)$	constraint function

in Section 3.4.2. Besides these fundamental components in a module, we also define a constraint function $\phi_{DUC \wedge TC}\,(m)$, which will return a boolean value to evaluate whether a module is in a well-typed status or not.

Specifically, $o \in \mathcal{O}$ is a possible configuration option. And $\mathcal{C} = 2^{\mathcal{O}}$ contains all possible configurations ($2^{\mathcal{O}}$) that could be derived, given a configuration option can either be selected or unselected. In a module, an import or context Γ describes a general environment for a module. The import function signature is described as a continuous projection relation from configurations to function names to types as $\Gamma \in \mathcal{C} \overset{p}{\to} \mathcal{X} \overset{p}{\to} \mathcal{T}$, where $\overset{p}{\to}$ shows a projection relation. The method defined in a module (Δ) can only be compiled if the configuration is available as represented as $\mathcal{C} \overset{p}{\to} \mathcal{X}$. Furthermore, this projection will be furthermore projected to a set of expressions under certain types as $\mathcal{T} \times \mathcal{E}$.

Signature First, we define the signature of module as follows:

$$signature : (\mathcal{C} \to \mathcal{X} \to \mathcal{E} \times \mathcal{T}) \to (\mathcal{C} \to \mathcal{X} \to \mathcal{T}) \tag{3.2}$$

The signature of a module is a partial mapping from function definition $\mathcal{C} \rightarrow \mathcal{X} \rightarrow \mathcal{E} \times \mathcal{T}$ to $\mathcal{C} \rightarrow \mathcal{X} \rightarrow \mathcal{T}$.

A well-typed module with variability Thereby, a tuple (v, Γ, Δ), which contains a feature model f, function declaration Δ, and import context Γ, could be used to represent a module m^v. Here, different from the notation for a module in the previous subsection, a module under a variability model v is represented as m^v. Note that, here we introduce a new concept *variability model*, which represents all valid configurations for the module. It is different from the feature model in two main aspects:

1. The feature model is the variability representation of the entire product line, whereas, the variability model is designed for our module system and each module contains a variability model. The feature model is shared among all modules; and

2. The variability model is actually a set, which contains all valid configurations for this specific module m^v.

However, when introducing variability, it means only a subset of functions could be imported rather than all, considering the fact that imported functions are highly restrained by configuration options. Therefore, the imported functions should be a partial mapping chain starts from the configuration and ends with a type as

$$\Gamma \in \mathcal{C} \rightarrow \mathcal{X} \rightarrow \mathcal{T}$$

.

Similar to module without variability, for all valid configurations ($\mathcal{C}$), which could be extracted from the variability model v, imports, declarations, and the mapping between imports and declarations should be well-defined and well-typed. A well-typed module with variability requires a two-step verification as: (1) despite variability, the module should be well-typed, as discussed in module without variability; and (2) with variability, each configuration c that could be extracted from the variability model v should be checked and ensure the module m^v is well-typed. Therefore, if a variability module m^v is well-typed, it should satisfy the following judgement.

$$\frac{\begin{array}{c} v \subseteq dom\,(\Gamma) \qquad v \subseteq dom\,(\Delta) \\ \forall c \in v.\,(\Gamma\,(c)\,, \Delta\,(c))\,OK \qquad\qquad v \neq \varnothing \end{array}}{(v, \Gamma, \Delta)\,OK}$$

Specifically, the variability of the module should be subjected to the domain of Γ and the domain of Δ. That is, the variability model v should be defined within the scope of Γ and Δ, with invalid configurations removed from v. Then, for each configuration c in v ($\forall c \in v$), the module should be well-typed as $(\Gamma\,(c)\,, \Delta\,(c))\,OK$, where v should not be empty.

Modules compatibility with variability. Furthermore, to merge two modules with variability into one ($m_1^v \cdot m_2^v = m^v$), the following checking should be conducted. Two modules are incompatible if there is a conflict when merging them. Conflicts could happen in two different cases:

1. Two modules import functions are of different types; or
2. A function is defined in one module and imported with a different type in another module.

Specifically, it could be translated into the following checking:

C1: for any configuration, a function should not exist in both modules, which is represented as follows:

$$c \in dom\,(Sig(\Delta_1)) \cap dom\,(Sig(\Delta_2) \mid dom\,(Sig(\Delta_1\,(c)) \cap dom\,(Sig(\Delta_2\,(c)) \neq \varnothing \quad (3.3)$$

, which means shared configurations $(dom\,(Sig(\Delta_1) \cap dom\,(Sig(\Delta_2))$ should be subjected to the expression that $dom\,(Sig(\Delta_1\,(c)) \cap dom\,(Sig(\Delta_2\,(c))$. It means that the signatures of two modules are checked with the constraint that for any configuration c from the *shared* domain of two modules($c \in dom\,(Sig(\Delta_1)) \cap dom\,(Sig(\Delta_2))$, the function should not exist in both modules as represented in $dom\,(Sig(\Delta_1\,(c)) \cap dom\,(Sig(\Delta_2\,(c)) \neq \varnothing$.

C2: there should not be any typing errors for the two input contexts, which is represented as follows:

$$\begin{aligned} conflict\,(\Gamma_1, \Gamma_2) = \{&c \in dom\,(\Gamma_1) \cap dom\,(\Gamma_2) \\ &\mid \exists x \in dom\,(\Gamma_1\,(c)) \cap dom\,(\Gamma_2\,(c))\,.\Gamma_1\,(c)\,(x) \neq \Gamma_2\,(c)\,(x)\} \end{aligned} \quad (3.4)$$

which shows a conflict is raised when a configuration c is a shared configuration from $dom\,(\Gamma_1) \cap dom\,(\Gamma_2)$ and the types for the same($\exists x \in dom\,(\Gamma_1\,(c))$) imported function are different $((x) \neq \Gamma_2\,(c)\,(x))$.

C3 and **C4**: for the conflict that could also come from the import context with the declarations as $conflict(\Gamma_1, Sig\,(\Delta_2))$ and $conflict(\Gamma_2, Sig\,(\Delta_1))$.

In summary, we can resolve compatibility issue for modules with variability by:

$$\frac{v' = \bigcup_{x \neq y} conflict_{C1,C2,C3,C4}\,(\Gamma_1, \Delta_1, \Gamma_2, \Delta_2) \qquad v = v_1 \cap v_2 \qquad v \setminus v' \neq \varnothing}{\div\{(v_1, \Gamma_1, \Delta_1)\,, (v_2, \Gamma_2, \Delta_2)\}}$$

where the function $\mathit{conflict}_{C1,C2,C3,C4}$ is represented by checking on $C1$ to $C4$. And the overall variability model v is created by combining two variability models $(v_1 \cap v_2)$ and then remove those introduced conflicts as $v \setminus v' \neq \varnothing$. The symbol $\div$ represents two modules are compatiable.

3.2.4 Module Constraints

Inspired by conditional compiling in C, which is mainly represented by the`#ifdef` directive, in order to include a code fragment at build-time, several constraints should be satisfied and pre-checked by translation units[72]. We will introduce the constraint space with a running example.

Def-Use Constraints(DUC): A module, regardless of its size, should be compiled safely without any errors. In this part, we try to simulate how JVM converts source code to bytecode and reports error if necessary. Unlike JVM which really processes the code, we just record the constraints required in this step. Specifically, we recover the following constraints as compiling constraints: All variables(global and local), fields, methods, classes, and interfaces that are used in this module but not defined should be added as **DUC**, except those defined in third-party API or fundamental framework, like JDK. Namely, it mainly includes the constraints from the *def-use* chain.

Type Constraints(TC): A parser will return an error when it encounters a type error, which is further extended as type checker tools in variability context[51]. In type constraints, we consider the program from two perspectives:

- A type error may come from a type used but not defined; technically, no class or interface can be bound with this type;

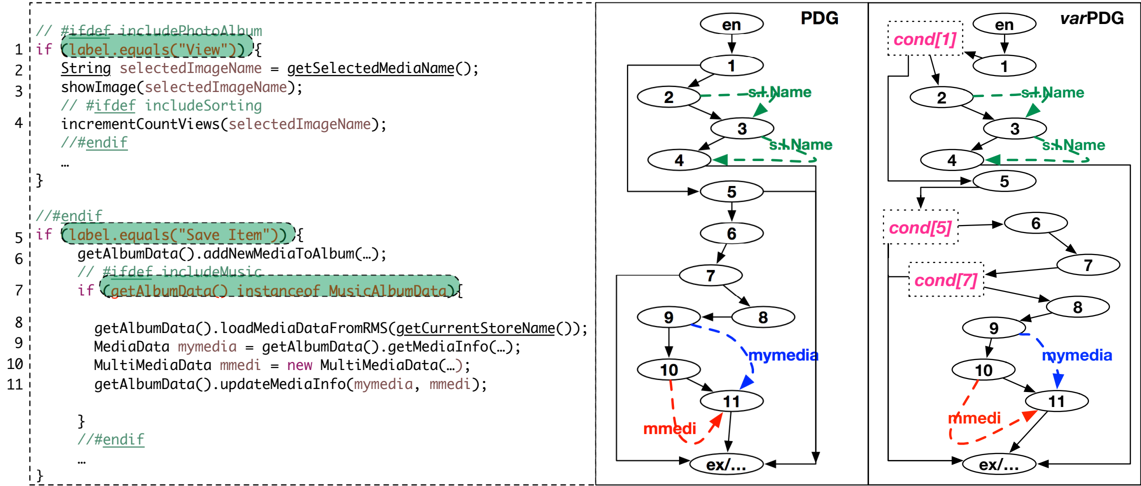


Figure 3.1: An example of *varPDG*

- When using a type in a module, its parent type should also be covered. We extend this to three cases: (1) variable/field; (2) method; (3) interface/class. For example, in (1), a variable in a child class can be used without defined when it has been defined in its parent class; (2) is simply referred as method overriding; and (3) is considered as a case of inheritance.

More formally, we create a formula to represent these constraints as:

$$\phi_{DUC \wedge TC}(m), \tag{3.5}$$

which will give a boolean value to show whether a module m is well-typed. Technically, we also use this function to report all missing types. For example, if a method is used, then the `TC` will check whether this method is declared under certain type and the compiler knows the type. If not, this checking will report a type missing.

3.3 Variability-aware Program Dependency Graph (*varPDG*)

Even with the variability-aware module system, it is not sufficient to recover the feature model, considering operations are required to be defined on these modules to compose features. For example, what is a safe scenario to combine two modules into one, and how to map the source code to modules? In this section, we will illustrate how to give a variability-aware presentation of source code and how to explore the configuration options from the source code base.

3.3.1 Building *varPDG*

A program dependency graph (PDG) is a combination of a program's control dependency graph (CDG) and its data dependency graph (DDG)[70]. We extend this graph by associating conditions with method calls, which means that if and only if several conditions are satisfied the method invocation or instance creation can occur. For example, considering the code excerpt shown in **Fig.**3.1, PDG is built by extracting and tracing the control dependency and data dependency. To build a variability-aware PDG, namely *varPDG*, the options within the program should be tracked. In our example, there are three options at line 1, 5 and 7 respectively, and the PDG is modified by adding these options as conditions; literately, we use *cond[+]*('+' shows the line number) to depict an option at a certain line.

By tracking all these options in source along with the underlying relations, the program can be represented in a variational matter, similar to variational Abstract Syntax Tree (varAST) in C[48]. We label each node in PDG with a *presence condition*

if possible. By tracking these, we can further extend our module system (in Sec.3.2) with *presence condition* in a variability-aware manner. Concretely, it helps to define the conditions that should be satisfied to execute some functions or code fragments. For example, in **Fig.**3.1, `cond[7]` controls node 8 to 11. But there is a dependency between `cond[5]` and `cond[7]` (`cond[5]`$\rightarrow$`cond[7]`), which leads to a result that node 8 to 11 can be executed, if and only if the condition `cond[7]`$\wedge$`cond[5]` is `TRUE`.

3.3.2 Tracing Options with Pointer Analysis

To build a *varPDG*, the configuration options should be explored and extracted from the source code. Unlike the macro strategy used in **C**, in OO programming language, like Java, the configuration options are extremely difficult to track and explore.

Listing 3.1: Running example of core idea in option controller

```
1 if(!batch){...
2  if (doSplash) {
3  splash = initializedSplash(); .... }
4 }
5  ...
6 if(splash!=null){...}
```

Points-to Analysis

Points-to analysis aims at providing a judgement on whether a variable p can point to variable q in some execution. This will explore the relation between variables and how the program is executed. Specifically, the main types of *points-to* are represented in **Tab.**3.3. The type of a points-to analysis could be flow-aware or context-aware with different concerns for precision. Specifically, the flow-aware points-to analysis computes a separate solution for each program point. Context-sensitive points-to

Table 3.3: Points-to approach

	Less Precision	**High Precision**
Flow	flow-insensitive	context-sensitive
Context	context-insensitive	context-sensitive

analysis takes the input context as a special concern and the input context will be propagated during the analysis.

CFL-reachability Points-to Analysis For Controlling

To resolve this, we adopt *Context-Free-Language* (CFL) reachability points-to analysis to track options with a higher precision[107]. Specifically, we only use language L_F and discard the regular language R_C, which ensures calling context sensitivity. L_F gives a graphical representation G of a Java program. There are four canonical statements that could define edges:

- Allocation `x = new O: edge` $o \xrightarrow{new} x \in G$
- Assignment `x = y:  edge` $y \xrightarrow{assign} x \in G$
- Field write `x.f = y:  edge` $y \xrightarrow{store(f)} x \in G$
- Field read `x = y.f:  edge` $y \xrightarrow{load(f)} x \in G$

Within G, we use the symbol $flowsTo \rightarrow new\,(assign)^*$ to indicate the `new` and `assign` edges. That is, `o` $flowsTo$ `v` in G represents `o` is within the points-to set of `v`. Moreover, the inverse symbol $\overline{flowsTo}$ is used to trace points-to for field access. Literately, if there is an $flowsTo$ edge from `o` to `v`, there must be a $\overline{flowsTo}$ relation

from `v` to `o`. Therefore, we design algorithm **Alg.** 1 to extract statements under different cases.

Algorithm 1: Option Controller

```
   Input: cond
   Output: StmtcondE,StmtcondUn
 1 for each x flowsTo cond do
 2 |  Add x to StmtcondE and StmtcondUn;
 3 Add enable statements to StmtcondE, unable statements to StmtcondUn;
 4 while TRUE do
 5 |  for statement s in StmtcondE do
 6 |  |  if there is a c flowsTo or flowsTo‾ s then
 7 |  |  |  Add c to StmtcondE;
 8 |  if StmtcondE not change then
 9 |  |  break;
10 The same process for StmtcondUn(line 4-9);
11 return StmtcondE,StmtcondUn;
```

Input. The input for this algorithm is the condition expression *cond* in the program.

Output. The outputs for this algorithm are: (1) *StmtcondE* represents statements when this condition is satisfied; and (2) *StmtcondUn* represent the opposite case.

Algorithm Body. For *StmtcondE*, its statements come from two sources: one from the statements when the condition *cond* is enabled(line 3), and another is from points to analysis, which contains (1) the $flowTo$ relation ends with *code* and (2) iteratively adding the statements having $flowsTo$ or $\overline{flowsTo}$ relation to the statements in *StmtcondE* until the set does not change (line 4-9). And we apply a similar process for *StmtcondUn*.

Example. As the code segment shown in **List.**3.1, the option controller algorithm will extract three options: `batch`, `doSplash` and `splash`. And it can find the underlying relations between these options as: $\neg$ `batch` $\wedge$ `doSplash` $\rightarrow$ `splash`, which mean the option `splash` is dependent on both $\neg$ `batch` and `doSplash` being `TRUE`.

3.4 VMS Feature Model Recovery Approach

3.4.1 Overview

Now, we will introduce our main idea. Basically, building a feature model from a legacy source requires processing the source code and clustering similar code fragments into several clusters. However, this faces some obstacles when coping with certain cases (see Section 3.1 **motivation**). Our approach, variability-aware feature model recovery, *VMS*, will build the variability-aware modules to resolve this. In general, it contains three steps:

1. **Step 1**: Initially, *VMS* will extract all common modules from the system. For example, in **Fig.**3.2, *VMS* will first find the source code for `SPL` and `PrevalyerSPL` without any optional features;

2. **Step 2**: Then, *VMS* will enable one configuration option in the common modules found in **Step 1** to trace all optional features. In each iteration, if there are some modules that become valid during this iteration, we will try to package these modules into a cluster, which is an *optional feature* in our context. In the example, we try to find *optional feature* like `Replication` along with its configuration option; and

3. **Step 3**: For each potential *optional feature* found in **Step 2**, we perform a type-checking and concern separation checking to adjust the modules in each feature if necessary. For example, this step will check whether the *feature* `Replication` is well-typed and whether there is a need to divide it into two subfeatures. And we also apply this checking to those common features found in **Step 1**.

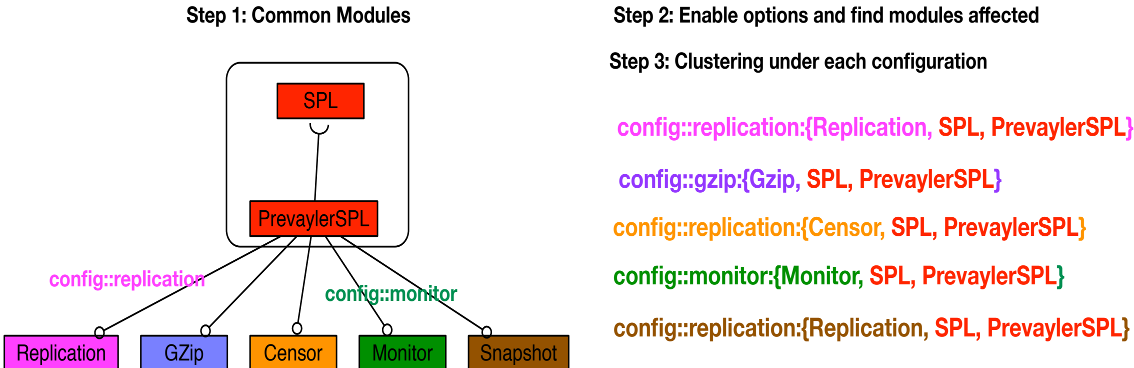


Figure 3.2: The core idea of VMS approach

3.4.2 Build Module from Source

First, we will broadly introduce how to construct a module from the source base. We build a module for each class or interface in a target program P and group the programming elements parsed by the following syntax. Specifically, in the syntax, we give an ASTNode expression for the mapping from source code to terms defined in our variability-aware module system. And all ASTNodes' types are defined under Eclipse Java development tools (JDT) [2].

[2] Eclipse JDT: `http://www.eclipse.org/jdt/`

Table 3.4: Syntax of VMS

Notation	**Syntax**	**Remark**	**Subremark**
$m \in \mathcal{M}$	::=	module	
	$v \in \mathcal{V}$		varability model
	$i \in \mathcal{O}$		import option
	$j \in \mathcal{O}$		export optio
	Γ		import func. sig.
	Δ		func. definition
$e \in \mathcal{E}$	::=	expression	
	`Exp.`		Expression
$o \in \mathcal{O}$	::=	configuration option	
	`pt(Do)`		
	`pt(For)`		
	`pt(If)`		
	`pt(While)`		
	`pt(Switch)`		
Δ	::=	function definition	
	`MethDecl.`		
$t \in \mathcal{T}$	::=	types	
	`ITypeBind.`		

Specifically, within a module m, it contains v for variability model, i for import configuration option, j for export option, Γ for import function signatures and Δ for functions defined in the module. The expression $e \in \mathcal{E}$ in a module could be extracted from ASTNode `Expression`. And the option $o \in \mathcal{O}$ in a module could be obtained by applying points-to analysis with function $pt(A)$, where $pt(A)$ returns all pointers defined in the conditional expression of A. In detail, it will check all ASTNodes that can lead to branches, including `DoStatement`, `ForStatement`, `EnhancedForStatement`, `IfStatement`, `WhileStatement`, and `SwitchStatement`. Here, we define the configuration option as the conditional expression in Java, due to the following reasons: (1) a conditional expression can lead to a certain path in the control flow graph with a

specific context. That means a conditional expression could give a variability context for configuration according to several empirical studies [103, 36]; and (2) our points-to analysis gives a comprehensive understanding of underlying relations between these options. The type $t \in \mathcal{T}$ could be obtained using type-resolving techniques provided by Eclipse with an input of the binding of the type. As for the function definition Δ, it can be extracted by visiting all `MethodDeclaration` node in the AST. The rest of components in module m, including, $i, j \in \mathcal{O}$, Γ and $v \in \mathcal{V}$, could be built based on these basic elements using our definition in Section 3.2.3.

3.4.3 Module to Feature

Moving on, we will cluster these modules into features, where a feature is composed of one or more modules. We first introduce the measurement for computing the distance between a module and a feature. The distance between a module and a feature (a collection of modules) could be used to decide the feature that a module should belong to.

Topology based Method Reference

Adopted from Robillard's topology work[83], we adjust it to compute the uniqueness of a module to a feature. The core idea of topology analysis is it computes the similarity using metrics *specificity* and *reinforcement*. Specifically, *specificity* suggests that the case of an element A only refers to another element B should be ranked higher comparing with the case that element C refers to many elements including B. And the intuition behind *reinforcement* is that if elements refer to (or referred from) many elements that are in one cluster, possibly they should be considered as a part

of that cluster. Therefore, we simply compute the uniqueness of a module m to a feature f as follows.

$$w_{tmr}(m, f) = \frac{1 + |targets(m) \cap f|}{|targets(m)|} \cdot \frac{|sources(m) \cap f|}{|sources(m)|}, \tag{3.6}$$

where $targets(m) = \{m' \,|(m, m') \in R\}$ and $sources(m) = \{m' \,|(m', m) \in R\}$. Here $(m', m) \in R$ represent there is a method invocation that starts from m' and ends with m.

Type Reference

Type reference ensures consistency and type-safety and works at a fine granularity. The underlying idea in type reference is that for each module, it looks up all possible references, such as method reference - from method invocation to method definition, variable reference - from variable access to its definition, or type reference (from a type reference to its declaration) and explore the types referred in all these references. For example, in module m, a method defined in type t is invoked, then we add the reference from module m to type t. For type reference, we define two types of vectors: $def(X)$ and $ref(X)$. Specifically, the vector $def(X) \in \mathbb{R}^n$ defines all types within X, where n represents the number of types in the subject program. $def(X) = [d_1, ..., d_n]$ is defined as if type i is defined in X, then d_i should be 1; otherwise d_i is 0. On the contrast, $ref(X) = [r_1, ..., r_n]$ shows the reference information. If a type i is referenced by X, then the ith element r_i in $ref(X)$ should be 1, otherwise it should be 0. Therefore, the type reference distance w_{tr} is defined as follows.

$$w_{tr}(m, f) = \frac{1}{2n}\left(\sum_{m_i \in f}\left(\begin{array}{c} csd(def(m), ref(m_i)) + \\ csd(ref(m), def(m_i)) \end{array}\right)\right), \tag{3.7}$$

where n is the number of modules currently in feature f and csd is defined as cosine similarity between two vectors:

$$csd\,(X,Y) = \frac{X \cdot Y}{\|X\|\,\|Y\|}.$$

The subtlety of this approach is that it uses cross reference to check how a module m_i in feature f relies on m with $csd(def(m), ref(m_i))$ and the opposite case showing how the module m relies on a module m_i. Here, we exclude the type defined outside of this program, like a type defined in a third-party API or a type defined in Java runtime environment.

Documental Topic Similarity

In a broader sense, a program can be considered as a set of *documents* and defined as a *corpus*. Upon this *corpus*, an information retrieval approach named Latent Dirichlet Allocation (LDA)[95] can be used to extract the topic distribution. Furthermore, a topic z is given based on a multinomial probability distribution upon a set of words ws obtained from a Dirichlet distribution with the shape parameter β. For example, given a topic label “life” and relative words “biology”, “gene”, “water”, and “oxygen” can be represented with certain probabilities, by learning from the *corpus*. Using LDA allows us to build a bridge between a module and a feature. That is, for each module, a vector is defined with each item inferring the probability on each topic. For example, if there are 5 topics and a module m is represented as a topic distribution $\theta_m = [0.5, 0.3, 0.1, 0.9, 0.1]$, which states that this module should be considered a part of topic with $id = 4$, then the probability 0.9 from a textual perspective. Then some similarity measure could be used to measure the distance of

two modules' topic distributions, like cosine distance Kullback-Leibler. Technically, we use the MALLET tool to create the topic model and adopt an empirical setting for parameters in LDA with $\alpha = 50/T$ and $\beta = 0.01$, since it has shown its strength across different corpora[95]. Therefore, we define the topic similarity w_{dt} using cosine similarity as:

$$w_{dt}\left(m, f\right) = \frac{1}{n} \sum_{m_i \in f} \frac{\theta_{m_i} \cdot \theta_m}{\left\|\theta_{m_i}\right\| \left\|\theta_m\right\|}, \tag{3.8}$$

where θ_m defines the topic distribution of m and n represents the total number of modules in feature f.

Putting All Pieces Together

For all distance values extracted (w_{tmr}, w_{tr} and w_{dt}), we denote an overview distance value as w_*. The overview distance w_* is defined by following Robillard's approach, which uses operator $x \uplus y = x + y - x \cdot y$ to combine two values[83]. This operator yields a result by equally treating all arguments and return its overall result within the range $[0, 1]$. Thereby, we calculate the overview distance by: $w_* = w_{tmr} \uplus w_{dt} \uplus w_{ss}$. In addition, we have to apply the variability-aware constraints on w_*. w_* represents the overall similarity between a module and a feature; the closer w_* to 1, the more similar they are.

However, combining two modules with high similarity may still introduce errors, which may come from: (1) merging import functions' signature ($\Gamma_x\left(c\right) \cup \Gamma_y\left(c\right)$) under a configuration $c \in \mathcal{C}$, (2) combining variability modules ($v_x \cup v_y$) and (3) merging imported and self-defined configuration options in each module ($i_x \cup i_y$ and $j_x \cup j_y$).

[2] MALLET: http://mallet.cs.umass.edu/

Therefore, to merge two compatible modules to yield a new one, any potential conflict should be checked to ensure all modules are well-formed. The conflict checking process is shown by the following logical proofing.

$$\frac{\begin{array}{c} m' = (v', i', j', \Gamma', \Delta') \\ v' = v_x \cap v_y \setminus conflict\,(\Gamma_x, \Delta_x, \Gamma_y, \Delta_y) \\ \Gamma'\,(c) = \Gamma_x\,(c) \cup \Gamma_y\,(c) \setminus (sig\,(\Delta_x\,(c)) \cup sig\,(\Delta_y\,(c))) \\ \Delta'\,(c) = \Delta_x\,(c) \cup \Delta_y\,(c) \\ i' = i_x \cup i_y \setminus (j_x \cup j_y)\,, j' = j_x \cup j_y \end{array}}{(v_x, i_x, j_x, \Gamma_x, \Delta_x) \bullet (v_y, i_y, j_y, \Gamma_y, \Delta_y) = m'}$$

The module checking for merging two modules into one is represented with combining option $\bullet$. The type checking detects the errors in these two modules. Specifically, the module checker will check: (1) the conflicts from two modules' merged variability model v' ($v' = v_x \cap v_y \setminus conflict\,(\Gamma_x, \Delta_x, \Gamma_y, \Delta_y)$), which mean the new conflict should merge v_x and v_y by excluding the conflict from configurations. This exclusion of configuration ensures that the new variability module v' can fully map the functions defined within the new module m' and all imports. This checks for the conflict from both modules import functions with the same name but different types; (2) defining sig gives a mapping $sig : (\mathcal{X} \rightarrow \mathcal{E} \times \mathcal{T}) \rightarrow (\mathcal{X} \rightarrow \mathcal{T})$. Therefore $sig\,(\Delta)$ returns a mapping $\mathcal{X} \rightarrow \mathcal{T}$, which is a Γ in our definition. Therefore, for the merged import function $\Gamma'(c)$ under the configuration c with exclusion of functions required by one module but defined in another module; (3) merge the configuration-option defined $j' = j_x \cup j_y$ and functions defined $\Delta'\,(c) = \Delta'_x\,(c) \cup \Delta'_y\,(c)$; and (4) merge the configuration-option imports and remove the import configuration option from one module, but already defined in another $i' = i_x \cup i_y \setminus (j_x \cup j_y)$. Therefore, we adopt

this checking to check whether a module mergings action should be allowed or not. Formally, we define a function to do this checking as follows.

$$\phi\left(m_x \bullet m_y\right). \tag{3.9}$$

This function will return a boolean value to show whether the merge will lead to any conflict. It returns `TRUE`, if this merge is safe, and `FALSE` for an unsafe merge.

3.4.4 VMS

Our ultimate target is to build a feature model by analyzing the source base.

Input. The input for *VMS* approach includes two values: $\#op$ representing the number of option features preferred in this feature model, and $\#cf$ showing the number of common features preferred.

Output. The output returns the feature model fm proposed by *VMS* automatically.

Algorithm Body. Now, we will introduce the main process of *VMS* approach to explore the feature model for source code. Due to the length of algorithm 2, we separate it into several sections and present them respectively.

- Line 1 - 6: First, extract structural information to build $varPDG$ and for each class/interface create a module;

- Line 7 - 12: We define the common modules as all modules that must be executed regardless of the input context. Therefore, we follow a Breadth-First-Search structure to add the "must" invoked functions into the queue Q iteratively. To find these "must" invoked functions, *VMS* discards method invocation enclosed in statements under a certain configuration option $op \in \mathcal{O}$. With

Algorithm 2: *VMS* feature model constructing approach

```
   Input: #op,#cf,P
   Output: fm
 1 Build the varPDG for input program P;
 2 Create an empty module to class/interface mapping D;
 3 for class c in P do
 4 |  create a module m using c;
 5 |_ add (m, c) to D;
 6 Create set CommonM;
 7 Create empty queue Q, add entry class en's main function main to Q;
   /* find common modules                                              */
 8 while Q not empty do
 9 |  f_head ← Q.poll;
10 |  if f_head not visited then
11 |  |  Find all functions needprocess, which not defined in the scope of a
   |  |    condition cond in varPDG;
12 |_ |_ Add all modules contain needprocess to CommonM;
   /* hierarchical clustering all optional module set and common module set
      respectively                                                     */
13 For each module commonM_m in commonM into a cluster commonM_i;
14 Let optional modules optionalM be optionalM ← D.keyset() \ commonM;
15 while commonM.size > #cf do
16 |  Find two cluster commonM_i and commonM_j with maximum
   |    w_*(commonM_i, commonM_j) and (commonM_i · commonM_j, OK);
17 |_ Update all reference information;
18 while optionalM.size > #cf do
19 |  Find two cluster optionalM_i and optionalM_j with maximum
   |    w_*(optionalM_i, optionalM_j) and (optionalM_i · optionalM_j, OK);
20 |_ Update all reference information;
21 Create fm from cluster recovered;
22 return fm;
```

that, the modules associated with these functions are considered as common modules;

- Line 13 - 14: As illustrated in the core idea(see Section 3.4.1), we create two sets: one contains all modules for common features $commonM$ and another contains all modules for optional features $optionalM$;
- Line 15 - 17: We conduct a hierarchical clustering on the common module set $commonM$ based on our distance measurement w_* under a module conflict checking function (X, Y, OK). Here, (X, Y, OK) represents there is no module conflict between module X and Y; Specifically, it will do two checking steps: (1) check the module constraints and (2) check potential errors for variability merging. Thereby, the conflict checking function could be represented by.

$$(X, Y, OK) = \phi_{DUC \wedge TC}(X) \wedge \phi_{DUC \wedge TC}(Y) \wedge \phi(X \bullet Y)$$

- Line 18 - 22: We also apply the hierarchical clustering upon all modules belonging to the optional module set $optionM$.

3.5 Case Studies

3.5.1 Experimental Settings

We will introduce the infrastructure selected for our study, how we do constraints checking in terms of constraints identified, and the ground truth for assessing the performance.

Infrastructure and Constraints Checking

To implement our work, we develop an Eclipse plugin system and integrate it with `TypeChef` system. The `TypeChef` system is a variability-aware parsing tool, which

gives a customized compiling service and variability-aware presentation for code fragments[51]. `TypeChef`[3] is used to provide constraint checking for our module systems. Unlike the normal use of `TypeChef`, which gives the type error during the checking, in our work, we only need it to return types required to be resolved at a certain point.

Ground Truth for Performance Assessment

In order to assess the performance of our work and compare it with other tools, we select the systems that have been well-researched with two kinds of information available publicly: (1) given our final target is to build a feature model from source base, the feature model should be available, which could be used to assess whether our approach could extract the correct feature relations. For example, if there is an *implies* relation from a feature f to feature f', ideally a competitive approach should give this relation as a part of output; and (2) some information should be available that shows how code fragments and features are mapped.

3.5.2 Subject Systems

We carefully select subject systems from different domains to verify our approach in multiple dimensions. A special factor to consider is the specific type of subject system, namely, we want to test our approach using systems from different domains and in different size scales, including small systems, medium-size systems and large-scale systems. For subject systems, we mainly target on systems in Java, with systems in `C` and `CPP` out of the scope, since they use directives to implement variability

[3] TypeChef is designed in C, a Java version of TypeChef's core function is implemented as a part of LEADT tool, available at: `https://github.com/ckaestne/LEADT`

and associated with configuration management techniques. Therefore, the following systems were selected for our study.

- **Prevayler**[4]. An open-source object persistence library for Java with 8009 LOC. It was a well recognized product for product line research[102, 63], although it is not originally developed as a product line application. It contains five features: *Censor*, *Gzip*,*Monitor*, *Replication*, and *Snapshot* with a dependency $Censor \Rightarrow Snapshot$.

- **MobileMedia v8** . Originally developed by University of Lancester, UK as a product line with 4653 LOC[38]. It contains several features: *Photo, Music, SMS Transfer, Copy Media, Favourites*, and *Sorting*. The dependencies include: $Photo \vee Music$, $SMS\,Transfer \Rightarrow Photo$ and

$$Media\,Transfer \Leftrightarrow (SMS\,Transfer \vee Copy\,Media)\,.$$

- **ArgoUML**[5] With 120 KLOC, ArgoUML provides modelling support for UML v1.4 diagrams and supports multiple programming languages. In ArgoUML, the following seven features are selected: *Cognitive*, *Activity Diagram*, *State-Diagram*, *Collaboration Diagram*, *Sequence Diagram*, *Use Case Diagram*, and *Deployment Diagram* from an empirical research[26]. The feature *Logging* is not covered in our mining work, as it is not a callable feature by the end customers. In another study[102], a dependency *ActivityDiagram* $\Rightarrow$ *StateDiagram* is added. In our experiment, we adopt the same setting.

[4] Prevayler: available at `http://prevayler.org`

[5] ArgoUML: available at: `http://argouml.tigris.org`

- **Berkeley DB (in Java)**[6] It is a database application with 84KLOC and 38 features. Berkeley DB could be an embedded application to other applications. It performs safe transaction and offers several useful APIs to cope with IO, logging, memory and so forth. A full feature list and feature relation is enclosed in our project webpage[7].

3.5.3 Tools

We have implemented a prototype of our work and integrated other related approaches into an Eclipse plug-in named *LoongFMR*. We have released the experimental data, ground truth, and source code on our project page: `http://www.chrisyttang.org/loong_fmr/`.

3.6 Experimental Result

In this section, we will first introduce several related approaches for comparing with our approach in Sec. 3.6.1. Then we list the results in Sec.3.6.2.

3.6.1 Related Approaches

ACDC

Algorithm for Comprehension-Driven Clustering (ACDC) recovers the architecture of a system by inspecting certain patterns that could exist in systems[101]. Specifically, ACDC contains *source file pattern*, *body-header pattern*, *leaf collection and*

[6] Berkeley DB: `http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/`

[7] LoongFMR: `http://www.chrisyttang.org/loong_fmr/`

support library pattern, and *ordered and limited subgraph domination.* ACDC identifies clusters by using these patterns with orphan adoption techniques, which were originally proposed in [100].

LIMBO

scaLable InforMation BOttleneck (LIMBO) optimizes the usage of information loss when conducting clustering on a system. It builds on *Information Bottleneck (IB)* framework and can collect relevant information during clustering[6].

ARC

Architecture Recovery using Concern (ARC) as defined in [42] uses a generative probabilistic model for text corpora named Latent Dirichlet Allocation (LDA) to retrieve concerns and identify programming elements belonging to concerns. It treats the source as a series of *documents* that contain various topics and then measures similarity using Jensen-Shannon divergence (D_{js}).

Bunch

Bunch regards the recovery task as an optimization problem[66]. It starts with a random partition and iteratively updates each cluster by optimizing the objective function called Modularization Quality (MQ) until it cannot find a better solution.

W-UE and W-UENM

Weighted Combined Algorithm(WCA) combines hierarchical clusters into larger sets by computing the inter distance between two possible variants using Unbiased Ellenberg (UE) and Unbiased Ellenberg-NM(UENM) distance measurement respectively[67].

3.6.2 Results

We measure the performance of different approaches by four aspects. We adopt three metrics from software architecture recovery: *MoJo similarity*,*architecture-to-architecture measurement* and *cluster-to-cluster coverage*. The runtime performance returns the execution speed of the algorithm.

MoJo Similarity

The SimilarMoJo metric[104] gives a representation of closeness between two architectures with a percentage. It helps to analyze two different architecture strategies. SimilarMoJo is defined as:

$$SimilarMoJo\,(A,B) = \left(1 - \frac{MoJo\,(A,B)}{N}\right) \times 100\%, \tag{3.10}$$

where $MoJo(A,B) = min\,(mno\,(A,B)\,, mno\,(B,A))$, $mno\,(A,B)$ represents the minimum number of *Move* or *Join* operations needed to transform from A to B and N represents the number of units in the system. The algorithm in [112] gives a way to calculate $mno\,(A,B)$. Furthermore, the symbol $\forall A$ in the denominator represents a partition of A and $\max\,(mno\,(\forall A,B))$ means the maximal distance from any partition A to B.

Table 3.5: Evaluated MoJo Similarity

Algorithm	Prevayler	MobileMedia	ArgoUML	BerkeleyDB
ACDC	83.0	75.0	63.69	83.93
LIMBO	55.56	68.75	52.03	78.31
Bunch	74.07	68.42	63.92	88.77
ARC	59.25	73.43	53.09	84.95
VMS	83.33	71.87	79.78	81.62
W-UE	66.67	78.12	51.66	83.41
W-UENM	68.52	71.87	51.66	82.14

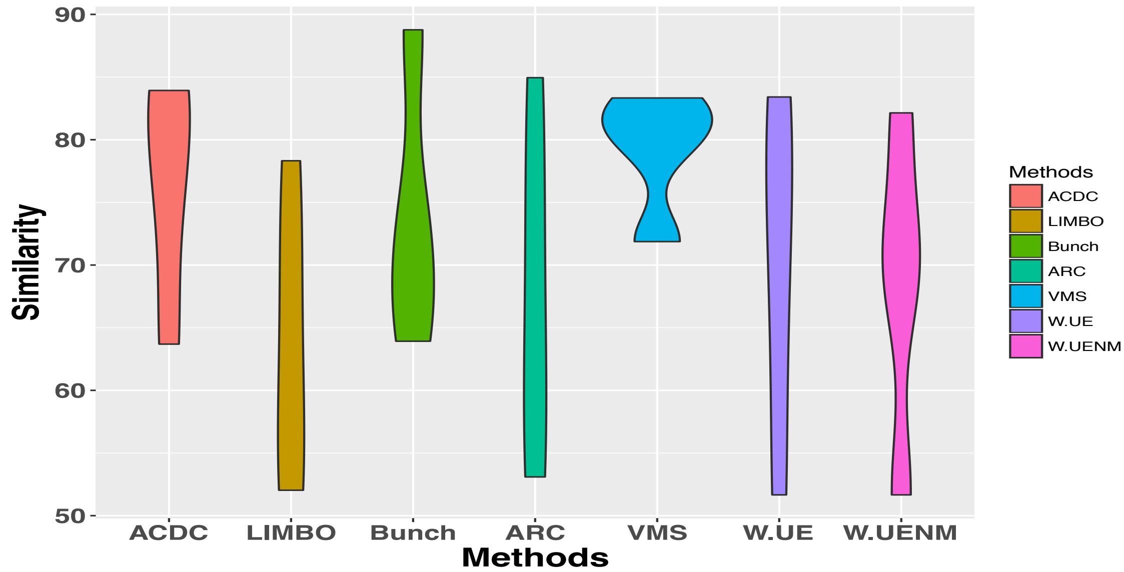


Figure 3.3: The violin plot for MoJo Similarity

Architecture-to-architecture Measurement(*a2a*)

a2a is developed to overcome the limitation of *MoJo* measuring discrepancy of files between the recovered result and ground truth[59]. *a2a* measures two architectures

A_i and A_j, one is the recovered and another is the ground truth by computing:

$$a2a\left(A_i, A_j\right) = \left(1 - \frac{mto\left(A_i, A_j\right)}{aco\left(A_i\right) + aco\left(A_j\right)}\right) \times 100\% \tag{3.11}$$

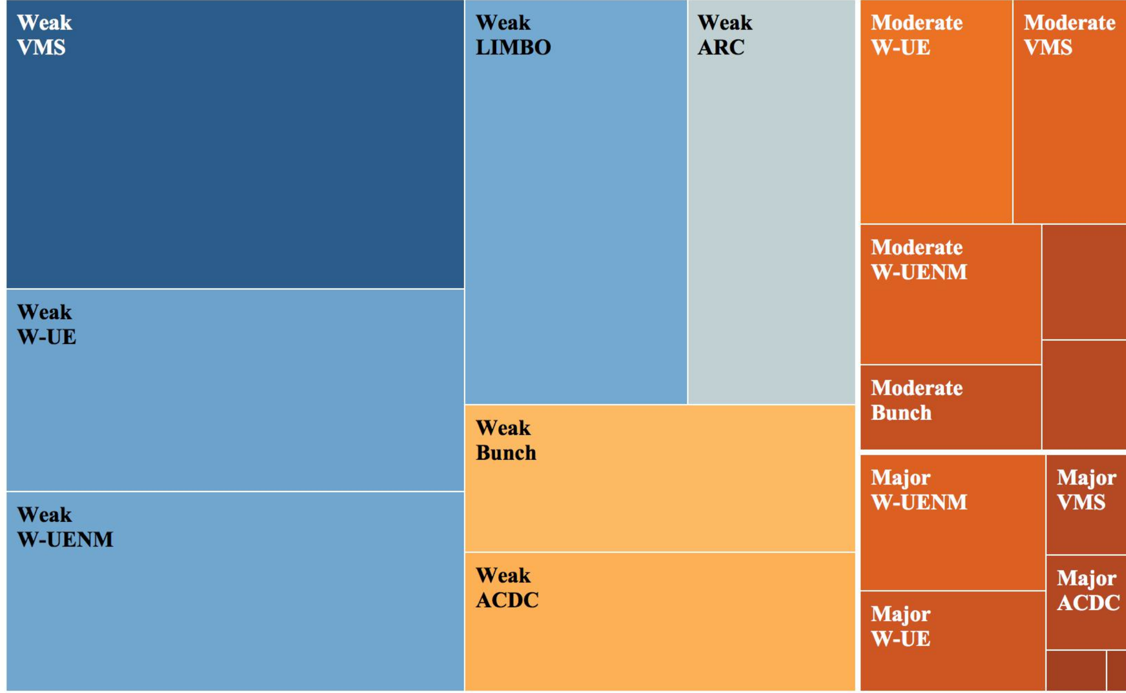
$mto\left(A_i, A_j\right) = remC\left(A_i, A_j\right) + addC\left(A_i, A_j\right) + remE\left(A_i, A_j\right) + addE\left(A_i, A_j\right) + movE\left(A_i, A_j\right)$ and $aco\left(A_i\right) = addC\left(A, A_i\right) + addE\left(A, A_i\right) + movE\left(A, A_i\right)$, where the symbol $mto\left(A_i, A_j\right)$ is the number of minimum changes from architecture A_i to A_j and $aco\left(A_i\right)$ represents the total number of operations from a "null" architecture A into A_i. There are five operations that could be used to transform an architecture to another including additions($addE$), removals($remE$), and moves($movE$) from one cluster to another.

Table 3.6: Evaluated Project and architecture with a2a

Algorithm	Prevayler	MobileMedia	ArgoUML	BerkeleyDB
ACDC	21.18	30.31	51.46	17.97
LIMBO	47.19	63.68	47.67	55.57
Bunch	45.07	56.75	49.06	49.08
ARC	49.23	67.39	49.76	63.48
VMS	52.16	67.70	52.87	62.92
W-UE	50.0	73.56	49.90	62.01
W-UENM	50.0	73.56	49.90	61.64

The results shown in **Tab.**3.5 and **Tab.**3.6, and associate violin plots, including **Fig.**3.3 and **Fig.**3.5, indicate that at the system level[8], our *VMS* approach could reach a competitive result and more importantly the result is stable comparing to others. From these two violin plots, we can draw the following conclusions: (1) the median value from *VMS* gives a better performance than others; and (2) from the

[8] Metrics MoJo and arch2arch give a system-level assessment, and cluster2cluster coverage returns a cluster-level assessment



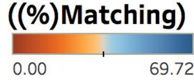


Figure 3.4: The Heat Map for Cluster-to-cluster Coverage

distribution and range between first and third quartiles, our *VMS* shows its strength in providing stable results.

Cluster-to-cluster Coverage($c2c_{cvg}$)

$c2c_{cvg}$ explores the component-level accuracy and is given as[59]:

$$c2c\left(c_i, c_j\right) = \frac{\left|entities\left(c_i\right) \cap entities\left(c_j\right)\right|}{\max\left(\left|entities\left(c_i\right)\right|, \left|entities\left(c_j\right)\right|\right)} \times 100\%, \tag{3.12}$$

where c_i is a cluster generated by clustering techniques and c_j is the cluster from the ground-truth. The $entities(c)$ represents all candidates in the cluster c. The *archi-*

Table 3.7: Cluster-to-cluster coverage(majority match(50%), moderate match(33%),weak match(10%))

Algorithm	**Prevalyer**			**MobileMedia**		
	Major	Moderate	Weak	Major	Moderate	Weak
ACDC	13.64%	18.18%	54.55%	0.00%	0.00%	30.00%
LIMBO	0.00%	0.00%	80.00%	0.00%	0.00%	28.57%
Bunch	5.26%	15.79%	47.37%	0.00%	14.29%	42.86%
ARC	0.00%	0.00%	20.00%	0.00%	14.29%	57.14%
VMS	16.67%	16.67%	83.33%	0.00%	14.29%	71.43%
W-UE	40.00%	40.00%	60.00%	0.00%	28.57%	71.43%
W-UENM	40.00%	40.00%	60.00%	14.29%	14.29%	71.43%
Algorithm	**ArgoUML**			**BereleyDB**		
ACDC	4.55%	4.55%	22.73%	0.00%	0.00%	7.32%
LIMBO	0.00%	0.00%	66.67%	0.00%	0.00%	14.63%
Bunch	0.00%	3.03%	21.21%	0.00%	0.00%	9.76%
ARC	0.00%	0.00%	22.22%	2.38%	7.14%	45.24%
VMS	0.00%	22.23%	77.78%	2.44%	4.88%	46.34%
W-UE	0.00%	0.00%	11.11%	0.00%	4.17%	54.17%
W-UENM	0.00%	0.00%	11.11%	0.00%	0.00%	50.00%

tecture coverage $c2c_{cvg}$ is a metric that extend the clusters overlap as:$c2c_{cvg}\left(c_1, c_2\right) = \frac{|simC(A_1,A_2)|}{|A_2.C|} \times 100\%$, where $simC\left(A_1, A_2\right) = \{c_i | (c_i \in A_1, \exists c_j \in A_2) \wedge (c2c\left(c_i, c_j\right) > th_{cvg})\}$. A_1 is the recovered architecture and A_2 is the architecture from the ground-truth. The symbol $A_2.C$ represents all clusters in A_2, and th_{cvg} shows the threshold that indicates the bottomline that for the clustering approach must achieve in order to count for similar clustering when comparing to A_2. The detailed performance of our approach and other related approaches are shown in **Tab.**3.7. In addition, the heat map in **Fig.**3.4 indicates *VMS* returns a better performance in terms of weak match($> 10\%$) and majority match($> 50\%$). For example, as for the weak match, *VMS* overwhelmingly performs better than other approaches, including *W-UE*, *W-*

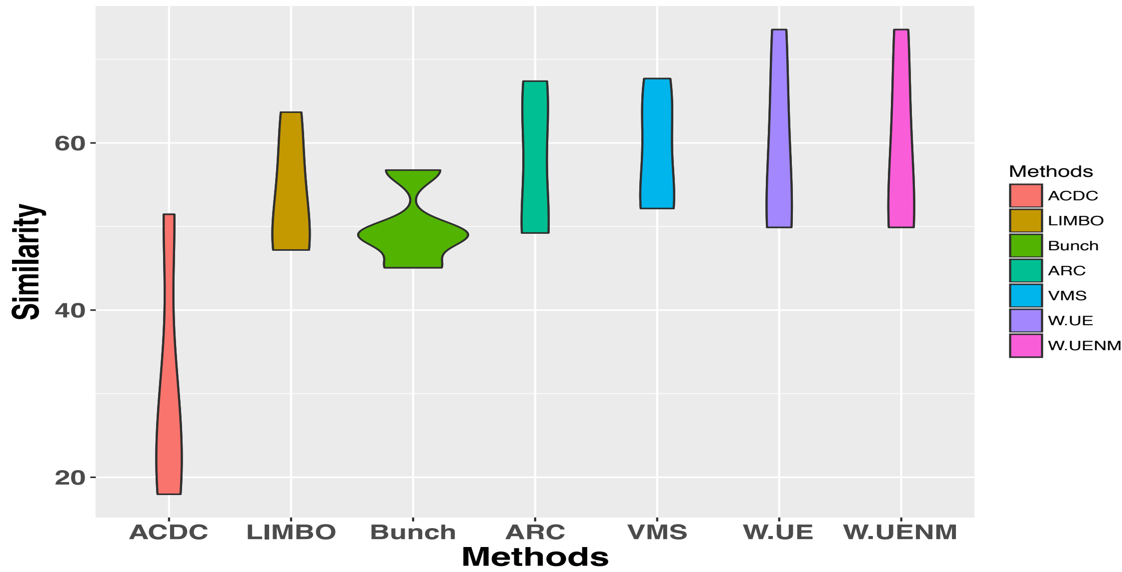


Figure 3.5: The violin plot for a2a Measurement

UENM, *LIMBO* and *ARC*.

Run-time Performance

Run-time performance explores the execution time under the same environment. In this study, all algorithms are run on a MacOS 10.12 with Intel i5 2.6GHz, 8G 1600 MHz DDR3, and targeting on Eclipse 4.5 with JRE 7.

Table 3.8: Runtime Performance in second

Algorithm	Prevayler	MobileMedia	ArgoUML	BerkeleyDB
ACDC	6	2	461	13
LIMBO	5	1	26296	13
Bunch	1	2	252	2
ARC	23	32	1398	64
VMS	1	3	4532	218
W-UE	2	1	303	7
W-UENM	1	1	320	5

As the runtime performance presented in **Tab.**3.8, a potential bottleneck for *VMS* is it requires more resource when processing large-scale systems, which might due to computation for the conflict checking and also computing the complicated model (w_*) to measure module similarity. Specifically, we found that if a system is not large, then the run-time performance of *VMS* behaves compatible with other approaches. Whereas the main bottleneck for *VMS* is for a more complex system, it has to compute and extract all relations between programming elements.

3.7 Discussion

3.7.1 Lessons Learned

In this section, we describe the experience learned from this study and share several empirical understandings in feature model building by answering following research questions. By answering these questions, the strengths and potential weaknesses of *VMS* are also presented.

RQ1*: Is current architecture recovery technique qualified for constructing feature model?*

As the results shown in the previous section, we can conclude that traditional techniques designed for architecture recovery cannot meet the need of feature model construction. The main limitations of these approaches include the following:

1. These approaches do not consider typing issue. To recover the feature model from the system, each feature should be well-typed, which could not be handled in those approaches.

2. These approaches mainly work at a coarse-granularity level, which means some

relations cannot be covered in those approaches. That in return weaken the performance of those approaches.

Another apparent limitation of these approaches is that they cannot ensure all programming elements in a cluster are well-typed, which is solved in our approach. Therefore, our strategy could be a better choice for product line feature model building.

RQ2*: What are the potential limitations of VMS approach?*

Although *VMS* returns a competitive result on four case studies, it still has its limitations. The limitations are two parts: (1) the first limitation is the runtime performance as we described in the previous section; and (2) another limitation is that it is still a coarse-granularity approach. After we carefully check the ground-truth, we found that some programming elements are shared by different features. This can only be resolved using a fine-granularity strategy. Whereas given the goal of building a feature model, a fine-granularity work might be overfit considering we only need to build a feature model to provide an overview of the system, for which our approach is fully qualified. Since, the feature model only requires features are identified and how these features are implemented is out of scope.

3.7.2 Threats to Validity

Construct and Internal Validity. The metrics, including SimilarMoJo, cluster-to-Cluster, and achitecture-to-architecture are broadly adopted in architecture recovery performance collection and have been tested on various target systems. The benchmarks are collected from other researchers' work, which are theoretically acceptable. Nevertheless, they may be incorrect as there is a widely recognized truth that there

is no single "correct" architecture.

External Validity. (1) Even though the size of our subject systems includes two small systems (4K, 8KLOC), a medium-size system (84KLOC) and a large-scale system (120KLOC), due to the small number of cases, the experimental results are not intended to be generalised to all systems. This is mainly because we have to restrict the systems to those with ground truth available. (2) Further in the assessment, we adopt the common architecture recovery performance metrics to testify our approach to reduce the bias of using a self-defined approach. Clearly, the correctness of ground truth can highly influence the performance.

3.8 Chapter Summary

Constructing a feature model and modeling variability are promising and worth investigating in product-line oriented research. In this chapter, we proposed an approach for constructing feature model by investigating variability-aware modules. As our results suggest, traditional methods used in architecture recovery could not reach a stable and competitive performance compared to our variability-aware approach.

Chapter 4

Feature Mining

In the previous chapter, we discussed how to build the feature model from source base by creating type safe variability-aware modules. We noted that traditional approaches in software architecture recovery are not suitable for building the feature model for the following main reasons: (1) software architecture recovery approaches do not consider relations at a fine-granularity level; and (2) architecture recovery approaches could not ensure each feature is well-typed. Given our target is to form a software product line from a legacy system, the next step should be matching features with their implementations. We define this process as *feature mining*. This chapter addresses how to match the features with their implementations.

4.1 Overview

Currently, most works in constructing a product line are primarily concentrated on solutions on analyzing product lines and building product lines from an abstract aspect, for instance, from the architecture or module level, including model checking, refactoring and so forth, to analyze variability in the product line[51, 29, 22]. The

main problem of these coarse-granularity approaches in terms of constructing a product line is that they cannot recover a feature's implementation in a fine granularity manner.

The task of mining features in code base could be considered a clustering problem. The goal of feature mining is mapping features with code, which could be deemed as grouping programming elements into different groups that represent features. Programming elements represent all possible programming units can be found in a program. For example, "class Tool" is a programming element. Therefore, the main problem is giving a quantified presentation for two programming elements rather than merely detecting their relations. For example, given two method invocations *A.act()* and *B.act()*, if *A* also invokes other five functions, but *B* only call method *act()*, *B* should be considered before *A* when checking the related programming element for method *act*. In this chapter, we propose a probability-based approach to address this issue.

To further assess our approach, we developed an Eclipse plug-in tool to obtain code fragments from the code base for the feature concerned, and we compared the performance with three other feature mining approaches, including type check, topology analysis and text comparison, with several case studies.

This chapter is organised as follows. Section 4.2 provides a bird's eye view of the feature mining process. Section 4.3 introduces the underlying model, and two research questions are raised. Our approach is introduced and explained in Section 4.4. We conduct case studies and exhibit our experimental results in Section 4.5 and 4.6 respectively and discuss the results in Section 4.7. We conclude this work in Section 4.8.

4.2 Feature Mining Process Overview

The procedure of detecting potential variants from legacy could be deemed as identifying assets from an application. Particularly, in this chapter, we focus on deriving features' implementation from the source base.

As illustrated in **Fig.**4.1, the whole feature mining process consists of four steps as follows:

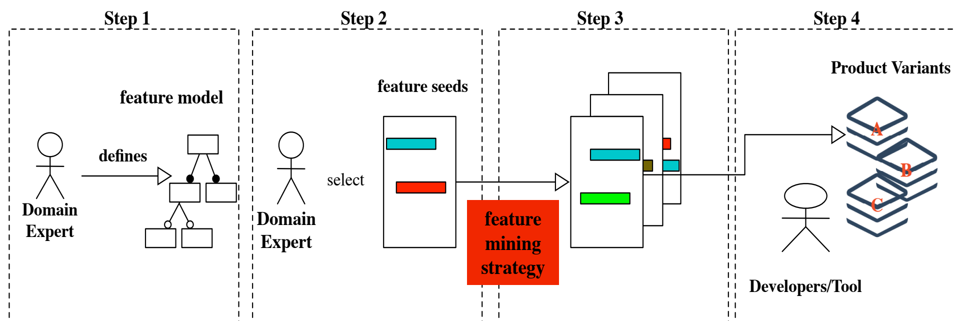


Figure 4.1: Feature mining process overview

1. A domain expert models the product and describes features and their underlying relationships and constraints in a feature model.

2. Moving on, for each feature, developers have to select an initial seed to represent this feature. For example, a method named "Lock" could be used to represent feature *locking*.

3. For each feature, the feature mining strategy expands the known code range for the feature iteratively until some stopping criteria are met.

4. Finally, developers *rewrite* code fragments with different configurations, that is,

using some variants from the variants set to generate members in the product family.

Within this process, we focus on **Step 3**, which is to obtain all code that implements a feature starting with an input seed and a feature relation model defined by the domain expert. As mentioned in **Step 2**, with the selection of seed for each feature, the feature mining task has been transformed to finding all related code fragments based on the given seed.

4.3 Underlying Model

4.3.1 Basis

Programming Elements. To retrieve code fragments that describe variants and their internal relationships, we use a graph-based representation of the system, in which nodes denote programming elements and links stand for dependences. Currently, most source-based tools (such as Suade[83] and Cerberus[33]) merely focus on methods and fields, which may lead to inaccurate results. In our approach, programming elements include local variables, fields, statements, types, methods, classes and interfaces. We denote the set of programming elements in a system as E. Technically, we use abstract syntax tree(AST) nodes to represent programming elements, since using AST nodes as basic units could contribute to finding relations between programming elements at a fine-granularity level.

Among these programming elements, relationship ($R \subseteq E \times E$) indicates how they are linked and impact each other. *Contain* relation shows the hierarchical structure between elements. For instance, import a package or API in a class (`import`

`java.util.Map;`). This relation could be discovered in class import (API import is covered), class instance declaration, enumeration, and inner class. *Reference* denotes a use relation, which could be method invocation, field use and type reference, for example, a relation from a local variable `cfg` to a field `controlFlowGraph` using `this.controlFlowGraph = cfg`, where a field named `controlFlowGraph` is accessed and updated with local variable `cfg`. In addition, *usage* provides an indirect reference between elements, that is, one element might reference another's attributes or functions. This relationship mainly includes `cast`, `instanceof`, super and child class.

Listing 4.1: The data structure of programming elements

```
e:<
 astid ,// the unique identifier of this
 parent_e ,// the parent AST node of this
 relation :{
        <relation_1: target1 >,
        <relation_2: target2 >,
         ....
 },// the relation  target mapping
 feature ,// feature annoated
 ...
>
```

The data structure shown in listing 4.1 indicates that a programming element is stored with following key attributes: (1) *astid:String*: the unique identifier of this AST node; (2) *parent_e:ASTNode*: the parent AST node of this AST node; (3) *relation:Map*: the relation is map, which contains all valid mappings from relations to target AST nodes; and (4) *feature*: represents the features assigned to this AST node.

Feature. In our product-line setting, we require additional domain knowledge

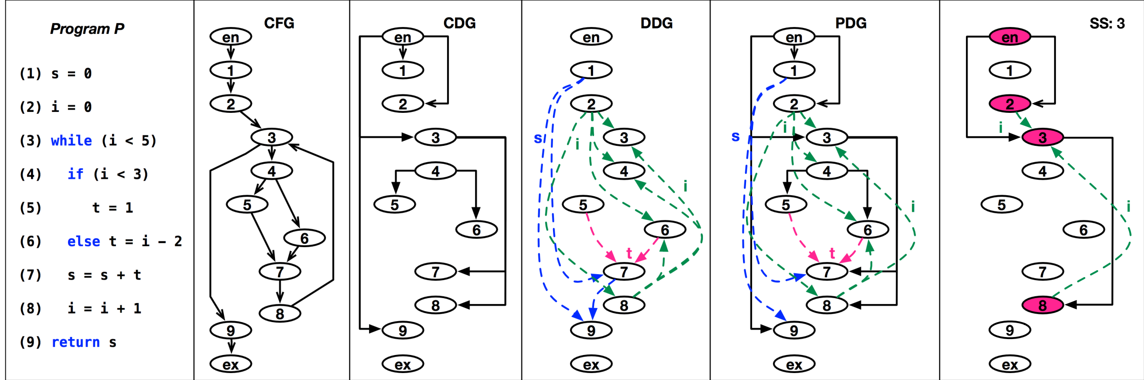


Figure 4.2: Example program P with its control dependency graph, data dependency graph, program dependency graph, and the slicing scope for node 3.

by defining the feature model[78], which describes how features enclosed in products are organized and their underlying dependencies and constraints. The feature model consists of a set of features (F) and relations between these features. Two fundamental relations: *mutual exclusion* and *implications* are frequently used in feature models. *mutual exclusion* ($M \subseteq F \times F$) denotes two features are mutually excluded and code segments belonging to one feature cannot be part of another. Whereas, *implications* ($\Rightarrow \subseteq F \times F$), which initially come from the "if feature f is included in some variants, f's implied feature g must be covered in these variants", is useful in terms of setting seeds, since it would be redundant to provide seeds for an implied feature. Implication is a typical relation between features in a hierarchical relationship.

In the previous chapter, we discussed how to build the feature model from the code base. In this chapter, we will extract the features from the feature model built. For example, for the feature model presented in **Fig.**4.3, the following features can be extracted: *structures*, *options*, *plus*, *neg*, *numbers*, *tostring*, *eval* and *expression*.

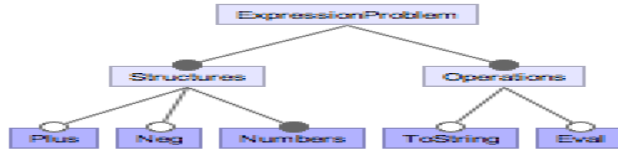


Figure 4.3: An example of feature model

Annotation. Annotation describes the mapping beFtween a feature and programming elements. That is, annotation ($A \subseteq E \times F$) shows programming elements that have been assigned to features during the seed selection. Each programming element could be attached to multiple features during the mining process.

For the relations ($\Rightarrow$ and M) mentioned, we extend the annotation as $A^* = \{(e, f) \,|\, (e, g) \in A, g \Rightarrow^* f\}$ to represent the closure of A with *implication* relation, where $\Rightarrow^*$ is the reflexive closure representation of $\Rightarrow$. In detail, a feature f's $\Rightarrow^*$ relation contains all features that implies f, that is ($g \Rightarrow^* f$), along with f itself. Thereby, A^* relation contains two parts: (1) all elements that are directly annotated to feature f as (e, f); and (2) all elements that are indirectly annotated to feature f using *implications* relation as $\{(e, f) \,|\, (e, g) \in A, g \Rightarrow f\}$.

4.3.2 Modeling Closeness between Element and Feature

RQ1: *How to measure the probability that a programming element should be annotated to a certain feature?*

Considering the whole process of feature mining approach as steps of annotating programming elements to feature, this section starts from raising a question on how to measure the probability that a programming element should be annotated to a

certain feature. Moreover, we introduce the concept of annotation state and feature-element correlation coefficient for modeling this question.

Definition 1 (Annotation State)**.** An annotation state of a feature f is defined as a set of elements that have been annotated to f. It is represented by

$$S\left(A^{*}, f, i\right) = \{e|(e, f) \in A^{*}\}$$

Here, i represents a certain annotation iteration. Specifically, the feature mining process for a single feature could be deemed as a series of transformation of annotation states as shown in **Fig.**4.4. In detail, at the beginning, seeds are selected and annotated to a feature. Then, one or more programming elements are annotated to this feature at each iteration, which transforms the current annotation state to its immediate successor, such as from $S\left(A^{*}, f, i\right)$ to $S\left(A^{*}, f, j\right)$ in the example. Thereby, the mining process for a feature f could be regarded as a series of annotation state transformation from $S\left(A^{*}, f, 0\right)$ to $S\left(A^{*}, f, Stp\right)$. The symbol $S\left(A^{*}, f, 0\right)$ represents the initial state in which only seeds are annotated to f. The state $S\left(A^{*}, f, Stp\right)$ is the final state, and it contains all code fragments that have been annotated to f when the mining process stops. For an adjacent transformation, like from annotation state i to j, the feature mining approach will look up all candidate elements which could be annotated to the current feature and annotate those with high likelihood. Therefore, we define the feature-element correlation coefficient to express this likelihood.

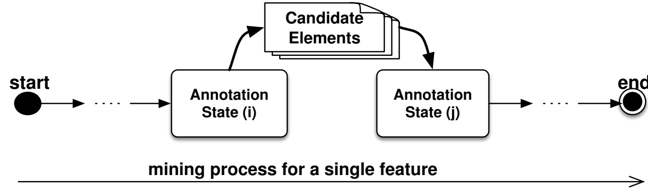


Figure 4.4: The general process of feature annotation

Definition 2 (Feature-Element Correlation Coefficient)**.** A measure of the probability that a programming element e should be annotated to feature f at an annotation state $S(A^*, f, i)$, and is represented in the form of a conditional probability as

$$p(e|S(A^*, f, i))$$

Given the correlation coefficient represented as $p(e|S(A^*, f, i))$, where $S(A^*, f, i)$ is a set of programming elements at ith iteration that has been annotated to f, there should be a method to measure "closeness" between two programming elements. Here, "closeness" indicates the degree that two programming elements belong to the same feature. And this conditional probability representation is designed to simulate this "closeness".

Therefore, the feature-element correlation coefficient could be regarded as the probability required in **RQ1**. To compute this correlation coefficient, another research question, that is how to capture the "closeness" between two programming elements, should be answered first.

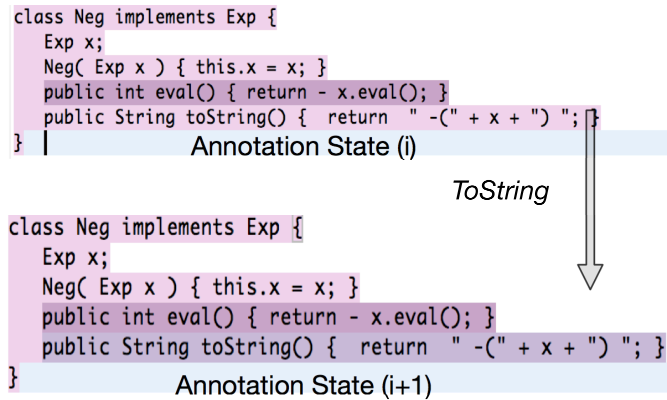


Figure 4.5: The general process of change on annotation state

Example. As shown in **Fig.**4.5, the annotate state changes from i to $i+1$. In this iteration, the programming element "public String toString()..." is annotated to feature "ToString". Therefore, the entire procedure of feature mining could be considered as a series of actions shown in the change on annotation status. At the beginning, seeds are annotated to features manually, which will create an annotation state change from state 0 to state 1. Furthermore, the feature mining approach will automatically make the transformation of annotation state from state 1 to the end of annotation. At last, when the annotation state does not change any more, the feature mining process is finished. Therefore, the main services provided by a feature mining approach should be finding the next programming element to be annotated to a specific feature. Considering the transformation from state i to $i+1$ for a specific feature is a process to infer the next element based on current state i. Therefore, to recommend the next elements to be annotated, an approach which could measure the "distance" between programming elements is essential as the next element is the one which is strongly related to elements at state i.

4.3.3 Modeling Closeness between Elements

RQ2: *How to provide a mathematical representation to capture the "closeness" between two programming elements?*

For the research question (**RQ2**) and the requirements presented above, we introduce two key concepts, slicing scope and binding.

Definition 3 (Slicing Scope)**.** For a programming element, its slicing scope is defined as

$$sscope\,(e) = e \cup \left\{ s | s \stackrel{df}{\to} e \vee s \stackrel{cf}{\to} e, s \in E \right\},$$

where $s \stackrel{df}{\to} e$ represents a data dependency flow from s to e and $s \stackrel{cf}{\to} e$ shows a control dependency flow from s to e.

Example. In the definition of slicing scope, control dependency graph (CDG) is a data structure which describes the control dependencies for operations in a program[74]. In addition, data dependency graph (DDG) shows data flow dependencies between statements [37]. A program dependency graph (PDG) contains all nodes defined in CDG and DDG, with edges in PDG all inherited from CDG and DDG. As shown in **Fig.**4.2, the PDG is used to compute the slicing scope. For instance, for the programming element `i<5` (line: 3), we obtain the slicing scope as a set of coloured nodes $sscope(3) = \{en, 2, 3, 8\}$. The entry en is covered by referencing the control dependency and nodes 2 and 8 are included due to data dependency.

Theoretically, for a given program p, the slicing scope of a programming element e in p returns a program slice with respect to a slicing criterion on that element

[0] Tool JayFX (`http://cs.mcgill.ca/~swevo/jayfx/.`) is integrated in our tool to extract and build CDG and PDG

e. Program slicing[15, 93] is a well-defined program transformation approach with respect to a given slicing criterion. Since the definition of slicing scope follows the same principle in building a program slice, program slicing serves as a theoretic footstone of concepts we defined and their extensions. Besides, it also indicates that approaches for building the program slice could be reused to obtain the slicing scope in our paper. Specifically, we use the program slicing approach defined in [15] to find the slicing scope with a given programming element.

Definition 4 (Binding)**.** For a programming element e, its binding $bind\,(e)$ is defined as all variables and fields defined or used in e's slicing scope as

$$bind(e) = def(e) \cup use(e)$$

Intuitively, our binding definition of a programming element e could represent e in a broad sense, considering all variables or fields, for both use and define, are covered. And their types are covered, since the type information is binding with the programming element. However, it is still insufficient to describe a complex relation between two programming elements (m, n) by just using their bindings $(bind\,(m)\,, bind\,(n))$, such as, method invocation, class inheritance, and method overriding. To resolve this, we further reinforce the binding definition by adding another factor "input context" to describe how a given programming element affects the current element. For example, for a call from method m to n, the call site in m is an "input context" for n. As other methods may also invoke n, each caller brings its unique "input context" to n.

Definition 5 (Context Binding)**.** For a programming element e, its context binding

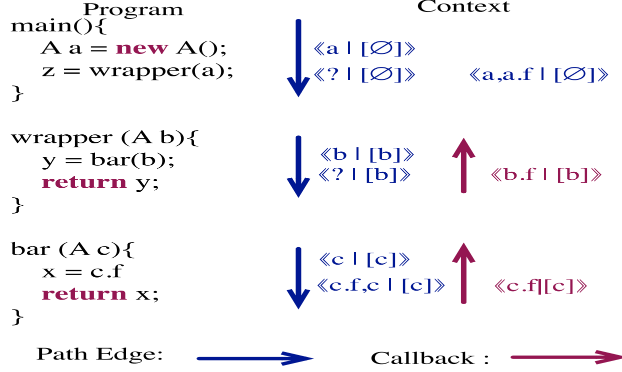


Figure 4.6: Example of context transformation in method invocation

contbind$(e, [c])$ is defined as all variables and fields defined $(def\,(e))$ or used $(use\,(e))$ in e's slicing scope under an input context from programming element c.

In context binding, *context* gives an unique identifier for a programming element at runtime. For example, given two contexts $a.f()$ and $b.f()$ for two different cases: one is function f is invoked by instance a and another is by b. Note that the inherent properties are dependent on object-oriented languages, as different programming languages have their own unique specification for implementing them. For example, multiple inheritances from classes are allowed in Python, but not allowed in Java. Here, we use the language specification defined in Java for illustration.

Method Invocation. Method invocation could bring context change especially when parameters are passed[70] from a call site to its callee. A call site `l`=r_0`.m`$(r_1, ..., r_n)$ will connect the parameters in the call site with arguments in invoked method `m` and `m` receives the *run-time* parameters $(r_1, ..., r_n)$ from this call site. Therefore, the invoked method obtains a unique input context from the call site. To resolve this, we follow Andersen's[5] context-aware analysis and dispatch the

binding of the call site into callee as:

$$contbind(m, [r_1, ..., r_n]) = dispatch\,(p_i = r_i) \rightarrow bind\,(m)\,,$$

where the context $[r_1, r_2, ...r_n]$ is dispatched to method m by mapping all parameters in the call site to the arguments in the callee.

Example. As the example shown in **Fig.**4.6, we use the label ($\ll curbind|[context] \gg$) to mark binding computed at a program point. In this tag, *curbind* represents the context binding collected at this program point, and $[context]$ shows the input context to this method. For example, in **Fig.**4.6, method `main` gives a context $[a]$ to its callee `wrapper`. In `wrapper`, this context $[a]$ is dispatched to the method body of `wrapper` as $a \rightarrow b$. And this initial context $[a]$ will continue be passed to `bar` from the method `wrapper`. Take the method `bar` as an example, its context binding $contbind\,(bar, [a])$, which represents the call path `main`→`wrapper`→`bar`, is $\{a.f, a\}$. Here we use $[a]$ just to provide a simplified representation of the context given by `main`; in practice, we use unique identifiers to encode contexts from different sources.

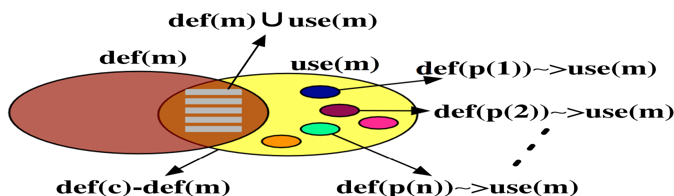


Figure 4.7: Example of context binding in overriding

Overriding. The context of an overriding method m defined in class c, is defined by all classes or interfaces that c inherited from. The context binding of a method

overriding $contbind\,(m)$ in a class c contains two parts: (1) programming elements defined and used in m, and (2) elements defined in c's parent class or interface and used in m. Therefore, a general representation of the context binding of an overriding method could be denoted as:

$$contbind\,(m, [p_1, \cdots, p_n]) = def\,(m) \cup \bigcup_{i=1}^{n} (use(m) \rightsquigarrow def\,(p_i))$$

,where the symbol $\rightsquigarrow$ is used to specify the source of the context. We illustrate this formula using the example in **Fig.**4.7. Based on our definition on binding, it contains two parts: the variables and fields defined and used in m. Particularly, the variables defined in the overriding method could be directly represented as $def\,(m)$. for the variables and fields used in m, they potentially come from three sources: (1) variables defined and used in method m as shown in the overlap area; (2) fields defined in class c as shown by $def\,(c) - def\,(m)$; and (3) all fields inherited from its parent classes and interfaces $p_1, p_2, \cdots p_n$. For example, $def\,(p\,(i)) \rightsquigarrow def\,(m)$ indicates all variables or fields used in m but defined in p_i.

Example. Considering a fragment of overriding given in **Fig.**4.8, class `FlyingCar` is inherited from interface `OperateCar`. `startEngine`'s binding contains two "encryptedValue" in different contexts, one is defined in `FlyingCar.startEngine`, and another in `OperateCar`. Thereby, the context binding of method `startEngine` (*contextbind*(`startEngine`)) should be {`OperateCar.encryptedValue,encryptedValue`}.

```
1 public class FlyingCar implements OperateCar {
2   public int startEngine(int encryptedValue) {
3     OperateCar.super.startEngine(OperateCar.encryptedValue);
4   }
5 }
------------------------------------------------------------
6 public interface OperateCar {
7   int encryptedValue = 1;
8   default public int startEngine(int value) {...}
9 }
```

Figure 4.8: A sample code of overriding

Inheritance. Different from overriding, for inheritance, we are interested in providing a context binding for the inherited class. The context binding of the inherited class c consists of the binding in c and all fields defined in all its parent classes and interfaces that c inherited from. It is defined as:

$$contbind\left(c, [p_1, \cdots, p_n]\right) = bind\left(c\right) \cup \bigcup_{i=1}^{n} def\left(p_i\right),$$

where $def\left(p_i\right)$ represents all fields defined in p_i. The context binding of inherited class contains all fields inherited from p_i, along with all the variables and fields originally defined in c, as $bind\left(c\right)$.

Now, we are able to provide a representation of an annotation status $S\left(A^*, f, i\right)$ of the feature f as

$$S\left(A^*, f, i\right) = \bigcup_{a}^{a \in S\left(A^*, f, i\right)} contextbind(a),$$

which is a collection of context bindings of all programming elements within the annotation status $S\left(A^*, f, i\right)$. However, in our model, $p\left(e|S\left(A^*, f, i\right)\right)$ is an exact value to indicate the probability that a programming element belongs to the feature

f based on the annotation status $S(A^*, f, i)$. Therefore, in the coming section, we will show how our approach works in exploring code fragments for features, and how the condition probability $p(e|S(A^*, f, i))$ serves a major role in the mining process.

4.4 ***StiCProb*** Approach

We first provide an overview of the "STatIc feature location with Conditional Probability" (*StiCProb*) approach, and then all its steps. Specifically, it contains three steps.

1. The first step is to build a database for the system, which contains all programming elements and their underlying relations. This part has been covered in Section 4.3.1.

2. The second step is to build a uniqueness table, in which the major task is to show the uniqueness between two programming elements with a relation.

3. At last, we use the feature-element correlation coefficient defined as an indicator to mine code fragments for features concerned.

The key characteristics of our approach is that it learns the probability from the context of each programming element (step 2) to seek potential elements to annotate. For example, given a call relation from method m to n and another call from j to n, if there are 5 call relations start from m and only 2 call relations start from j, j should be more unique to n in terms of call relation. And *StiCProb* is able to collect this kind of context information to be used for the feature mining.

Knowing that **RQ1** can be answered only if **RQ2** is answered first, this section starts from addressing **RQ2**.

4.4.1 Selecting Seeds

In this chapter, the seeds are collected by using a tool named FLAT^3. FLAT^3 combines the IR and dynamic techniques to provide recommendations. As the architecture shown in **Fig.**4.9, FLAT^3 works as follows: (1) an input query received from the user for the description of feature concerned; (2) users are required to execute a series of actions in the application for the function described in the query, during which scenarios are captured with relations between related programming elements are collected. For example, if a user wants to find the programming elements associated to feature "save" in a text editor, what (s)he has to do is a series of actions, including *open*, *edit* and *save*, which describe the intent of feature "save". During the execution, these actions are captured by FLAT^3. Along with these actions, from the code perspective, they are indirectly represented by a set of method invocations. FLAT^3 keeps trace of the executions, and a list of methods that executed are collected as shown in step 3 to 5. In the previous step (dynamic), programming elements related to the feature concerned are collected, but not ranked. Therefore, FLAT^3 adopts the Lucene library[1] to rank all these candidates as shown in the unit "Information Retrieval Engine". Moving on, it will return the ranked results to developers.

[1] Apache Lucene: available at: `https://lucene.apache.org/core/`

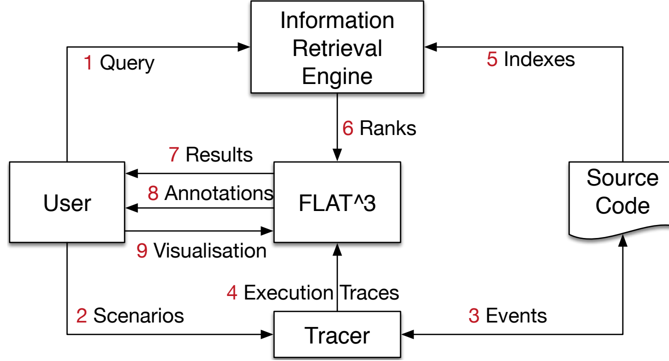


Figure 4.9: The architecture of FLAT^3

4.4.2 Building a Uniqueness Table

We reserve a uniqueness table for the system to describe the cross uniqueness for any two programming element(s,t) under a relation r ($s \xrightarrow{r} t$). In detail, a relation table could be represented as a five-tuple $U\left(E, T, R, P_{forward}, P_{backward}\right)$. For a specific element $u \in U$, it is represented as $u\left(s, t, r, p_{forward}, p_{backward}\right)$, which means there is a relation r from the programming element s to t. The definition of $p_{forward}$ could show the uniqueness of $\underline{s \text{ to } t}$ for relation r. We could define the probability $p_{backward}$ as

$$p_{backward}\left(s \xrightarrow{r} t | (t, f) \in A^*\right) = \frac{contbind\left(t, [s]\right)}{\bigcup_{i \xrightarrow{r} t} contbind(t, [i])},$$

where $\bigcup_{i \xrightarrow{r} t} contbind(t, [i])$ represents a collection of context binding from all programming elements, which have relation r with t.

The uniqueness of $\underline{t \text{ to } s}$ for relation r if s has been annotated to feature f is

represented by $p_{forward}$. Thereby, we define probability $p_{forward}$ as

$$p_{forward}\left(s \xrightarrow{r} t | (s, f) \in A^*\right) = \frac{contbind\,(t, [s])}{contbind\,(s)},$$

where $contbind\,(t, [s])$ represents the context binding of t given a context from s according to our previous definition on *contbind*.

Example. As shown in **Fig.**4.10, for the call from s to t ($s \xrightarrow{call} t$), the forward probability $p_{forward}\left(s \xrightarrow{call} t | (s, f) \in A^*\right)$ describes the uniqueness of t to s. That is, there might be multiple call relations starting from s as shown in the example, the value of $p_{forward}$ indicates the weight of the call from s to t in terms of all method call relations starting from s. On contrast, the backward probability $p_{backward}\left(s \xrightarrow{call} t | (t, f) \in A^*\right)$ depicts the uniqueness of s to t as the weight of the call from s to t referencing all method-call relations that end with t.

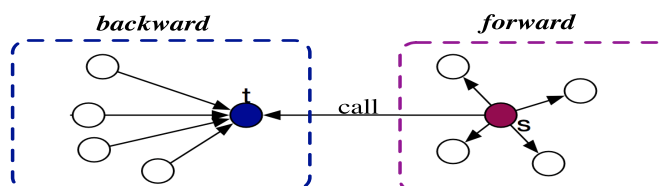


Figure 4.10: An example of a call relation

The strength of forward and backward probability for a relation ($s \xrightarrow{r} t$) from s to t is that they can capture s and r's surrounding information respectively. Specifically, the backward probability $p_{backward}$ could explore the relative context information of t as the left side shown in **Fig.**4.10. And the forward probability $p_{forward}$ could explore the relative context information of s as the right side shown in **Fig.**4.10. Thereby,

with the forward and backward probability, the research question **RQ2** is solved. Moving on, we will answer **RQ1** by introducing the detail of *StiCProb*.

Example. Furthermore, we will use a running example to show how the uniqueness table is built and illustrate why it would be useful to give a more accurate representation of relations between programming elements.

Listing 4.2: Running example of uniqueness table

```
static Exp e
static void evaltest(){
 e = new Num(1);
 System.out.println("eval(1)␣=␣" + e.eval());
 e = new Neg(new Num(1));
 System.out.println("eval(Neg(1))␣=␣" + e.eval());
 e = new Plus(new Num(1), new Num(2));
 System.out.println("eval(1+2)=" + e.eval());
 e = new Neg(new Plus(new Num(1), new Num(2)));
 System.out.println("eval(-(1+2))=" + e.eval());
}
```

As the example shown in listing 4.2, the method *evaltest*() is "related to" the following types: *Num*, *Neg*, *Plus*, *e* and method *eval*. In detail, in method *evaltest*,

1. *e* is defined 4 times, and *e* is accessed 4 times. (8 times in total);

2. Num's constructor $Num()$ is invoked 6 times;

3. Neg's constructor $Neg()$ is invoked 2 times;

4. $Plus$'s constructor $Plus()$ is invoked 2 times;

5. method $eval()$ is invoked 4 times;

Table 4.1: Uniqueness Table: e

	Num	Neg	Plus	e	eval()	....
evaltest()	6	2	2	8	4	....

Therefore, the uniqueness of Num to method $evaltest()$ can be computed by:

$$\frac{6}{6+2+2+8+4} = 27\%. \tag{4.1}$$

As for the uniqueness of $evaltest()$ to Num, it can be computed by looking at the class Num.

The example shown just briefly presents the uniqueness table. However, the subtleness of uniqueness table not well described,

1. First, the example just shows a snippet of a program, but our uniqueness table will check the whole system. The values computed are also based on the information collected from the whole system.

2. Second, as shown in the example, we compare the relation between a method ($evaltest()$) and a type (Num). That is, our approach could compute the relations across types, which is not possible for other approaches.

3. Third, our approach could present the uniqueness of relations with a real value. For example, some approaches count the method invocation from $evaltest()$ to $eval()$ as 1 rather than 4; in our approach, we try to collect the number of occurrence from the program. However, as our approach is still a static analysis strategy, it cannot cope with those cases that can only be decided dynamically.

4. Forth, our approach considers a relation from two directions rather than one and also we could detect the "environment". Normally, to compute the "closeness" between two programming elements, only these two elements are considered, without considering others. In our approach, for two programming elements A and B, we check both A and B's surrounding environments as well rather than only A and B.

5. Finally, for two programming elements, the relations between them could be multiple types rather than one. Our approach could detect all types of relations between two programming elements and compute the relation based on the information collected from the entire project.

4.4.3 *StiCProb*

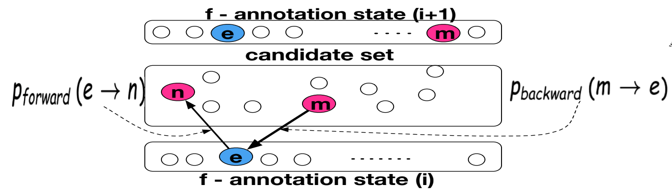


Figure 4.11: Illustration of *StiCProb*

We use **Fig.**4.11 to illustrate the underlying idea of *StiCProb*. As introduced previously (see Section 4.3), the mining process for a feature could be regarded as a series of transformation of annotation states. For a feature f, at the beginning, seeds are selected for this feature, which gives the initial annotation state of f as $S(A^*, f) = seeds$. Iteratively, the annotation state transforms from one state to another and at each transformation one or more programming elements are annotated

to f. For an annotate state transformation, as shown in **Fig.**4.11, the candidate set covers all elements that have direct relations with elements in annotation state (i). And then *StiCProb* categorizes the candidate set into two parts. The first part includes relation starts from an element in the candidate set, like $m \rightarrow e$, with the backward probability $p_{backward}$ to represent the closeness of this relation, which indicates the uniqueness of $\underline{m \text{ to } e}$. The second part includes a relation that starts from elements in the annotation state and ends with an element inside the candidate set, such as $e \rightarrow n$, with the forward probability $p_{forward}$ to describe this relation, which shows the uniqueness of $\underline{n \text{ to } e}$. In other words, *StiCProb* assesses all candidates by giving a probability description of how they are unique to elements in the current annotation state.

The pseudo code of *StiCProb* approach is shown in **Alg.**3. The detailed introduction of *StiCProb* is separated into three components: input, main procedure and output.

Input. The input to algorithm *StiCProb* is the program, containing the followings:

1. Feature model (fm): A feature model is given to show all features required to mine and their underlying relations. Proposing approaches to obtain the feature model is out of scope of this capture. In the case study, we adopt the feature model defined for subject systems from other research works.

2. Feature seeds ($seeds$): The seeds (represented by $seeds$ in algorithm) selected for each feature. For each feature, one or more programming elements are selected as seeds to represent a feature. We leave the discussion on how to

Algorithm 3: StiCProb feature mining approach

```
    Input: seeds, fm, threshold, U
    Output: all annotation states for features Sset in fm
 1  Create a set of annotation states as Sset = ⋃_f^{f∈features} S(A*, f);
 2  Assign seeds to each feature as S(A*, f) = seeds(f);
 3  Create feature set features with all features in fm;
 4  while features not NULL do
 5      for feature f in features do
 6          Create set waitList = ∅;
 7          Create candidate set C(S, f) = ∅ for f;
 8          Add all elements have relations with elements in S(A*, f) to C(S, f) ;
            // initialize C(S, f)
 9          for element m in C(S, f) do
10              if there is a relation r from m to the element e in S(A*, f) then
11                  Let value = p_backward(m -r-> e | (e, f) ∈ A*);
12              else
13                  Let value = p_forward(e -r-> m | (e, f) ∈ A*);
14              if value > threshold then
15                  Add m to waitList;
16          Update S(A*, f) ← S(A*, f) ∪ waitList;
17          if StopCheck(f) is TRUE then
18              Remove f from features;
19  return Sset;
```

select seeds for a feature in section 4.5.1: *Experimental Setting*.

3. *threshold*: It is used to decide whether a programming element could be annotated to a feature.

4. Uniqueness table (U): The uniqueness table U is built for all element-relation tuples (m, n, r), where there is a relation r from m to n.

Main procedure. We will introduce how *StiCProb* contributes to feature mining.

Due to the length of the algorithm, we separate it into four sections and present them seperately.

1. Line $1-3$: For each feature, an annotation state $S\left(A^{*}, f, i\right)$ is created. And a set $Sset$ is used to store all these annotation states. Each annotation state is initialized with seeds for feature f.

2. Line $4-8$: For each feature, a candidate set C is created by adding all elements having relations with the elements in the current annotation state S. And relations could be covered in both directions. That is, an element that either has a relation targeted at an element inside S or has a relation from an element inside S should be covered in C.

3. Line $9-15$: Following the previous step, it iterates over all elements in the candidate set. If there is a relation from an element m within the candidate set to an element e in the annotation state (Line $10-11$), the backward probability $p_{backward}$ is used to capture the relation. For opposite direction, the forward probability is used. In addition, Line $10-13$ is the kernel of *StiCProb*, since it shows how the feature-element correlation coefficient (see **Def.**2) is implemented in our approach. It also gives the answer to research question **RQ1**.

4. Line $16-18$: Line 16 will update the annotation state for each feature by adding the $waitList$, which contains all elements that should be annotated to this feature. The rest (Line $17-18$) checks whether it can stop the current mining process using a function $StopCheck(f)$. The concrete description of

function *StopCheck* is introduced in section 4.4.4: *Stopping Criteria*.

Output. The output returns the set ($Sset$) of all annotation states for all features in the feature model. The feature mining process for all features are finished and each annotation state $S(A^*, f, Stp)$ gives all elements that have been annotated to feature f.

4.4.4 Stopping Criteria

Stopping criteria is shown as the $StopCheck$ function in **Alg.**3. We use the $threshold$ as an indicator to stop the mining process for a feature. For a feature f, if all $value$s (Line $11, 13$) computed in the algorithm are lower than the threshold defined, the mining process is halted. $value$ is computed based on either forward or backward probability to determine whether a candidate element should be annotated to a certain feature.

4.5 Case Studies

4.5.1 Experimental Settings

Defining Feature Model and Selecting Seeds

In principle, a domain expert should be involved in the experiment and contribute to two parts: (1) defining the feature model for the legacy system. and (2) helping to select seeds for each feature. Therefore, the domain knowledge of features and their relation will have a significant impact on the performance of the feature mining approach. Considering our target is to verify *StiCProb*'s performance, it will be wise to reduce all human bias by using the feature model that has been well adapted

and learned by other research works and selecting seeds using automatic tools. In this study, we use the feature model built by other research works and a tool named FLAT^3[90] to obtain the seeds for each feature. This is done to exclude bias, and make the semi-automatic (see Section 4.3) process repeatable.

Other Settings

We list settings for other factors which could influence the performance of feature mining approach as follows.

- **Number of seeds for each feature.** As mentioned we use $FLAT^3$ to provide seeds for each feature. In the experiment, we select the top three items returned by $FLAT^3$.

- **Threshold.** In *StiCProb*, we use a threshold of 0.6. Intuitively, a higher threshold will make the annotation more precise and a lower one will annotate more programming elements. Here, in the experiment, the threshold is set to a median value and its influence on performance is discussed later in section 4.7.

The first setting (number of seeds) is suitable for all approaches, including our *StiCProb* and other three related approaches. However, the second setting for threshold is specific to *StiCProb*. With all these settings, we try to exclude the possible biases in conducting our experiment in selecting seeds and subject systems.

4.5.2 Subject Systems

Note that not all legacy systems are qualified for our experiment, as we need a specific benchmark for the system to be available. The benchmark contains a set

of files that describe how programming elements are mapped to features. Without the benchmark, it would not be possible to assess the performance of our approach. Thereby, we use those systems that have been analyzed and learned in other works to exclude the bias in creating the benchmark on our own. However, this, in return, limits the scope of subject systems. As a result, we carefully select subject systems that have been developed and well-researched by others, from academic and industrial systems. These subject systems are described and listed in Sec. 3.5.

4.5.3 Tools

We have implemented our *StiCProb* and other related approaches with an Eclipse plug-in tool named *Loong* following the feature mining process defined in **Fig.**4.1. We have released the source code, a full tutorial for this tool, and experimental data on the project host page: `http://www.chrisyttang.org/loong/`.

4.6 Experimental Result

4.6.1 Related Approaches

We carefully select 3 related approaches used in [50] for performance comparison.

- **Type System.** Type system [48] is initially designed to bring a type-checking system to product line that ensures all variant products generated are type safe. It has been re-implemented to cope with the feature mining task, and the underlying idea is to look up definition from references. For example, if a type reference is annotated to a feature f, the type declaration of this type should also be annotated to f. The type system checks all these relations,

Table 4.2: StiCProb Performance with $threshold = 0.6$

		Feature Size			Mining Result		
Project	**Feature**	**LOC**	**FR**	**FI**	**IT**	**Recall**	**Precision**
Prevalyer	Censor	105	10	5	3	17%	60%
	Gzip	165	4	4	2	16%	100%
	Monitor	240	19	8	2	17%	82%
	Replication	1487	37	28	26	79%	98%
	Snapshot	263	29	5	9	42%	99%
MobileMedia	CopyMedia	79	18	6	4	43%	95%
	Sorting	85	20	6	4	32%	100%
	Favorites	63	18	6	12	20%	100%
	SMS Transfer	714	26	14	23	91%	49%
	Music	709	38	16	4	39%	90%
	Photo	493	35	13	5	63%	61%
	MediaTransfer	153	4	3	14	97%	94%
Lampiro	Compression	5155	33	20	34	40%	82%
ArgoUML	Cognitive	16319	285	233	127	70%	92%
	Activity	2282	115	80	17	26%	74%
	State	3917	115	88	18	33%	82%
	Collaboration	1579	53	40	40	17%	72%
	Sequence	5379	65	53	98	33%	89%
	Use-Case	2712	59	49	39	19%	70%
	Deployment	3147	57	47	36	22%	67%

such as, from method invocation to declaration, from variable/field access to its declaration, and from type access to its declaration.

- **Topology Analysis.** Originally designed by Robillard[83] and adjusted to the feature mining task in [50], topology analysis explores all structural neighbors, such as caller method and related fields, for a given programming element. It then ranks related programming elements according to two metrics *specificity* and *reinforcement*.

- **Text Comparison.** Text comparison defined in [50] reserves a *vocabulary* list for each feature. It ranks the substring based on a relative weight and its occurrence.

4.6.2 Results

Our definitions for recall and precision are specific to the feature mining procedure. Our framework returns all lines that could belong to the features at the statement level. In a *binary* mapping context, a single statement could either belong to a feature or not. In our framework, precision and recall are defined as:

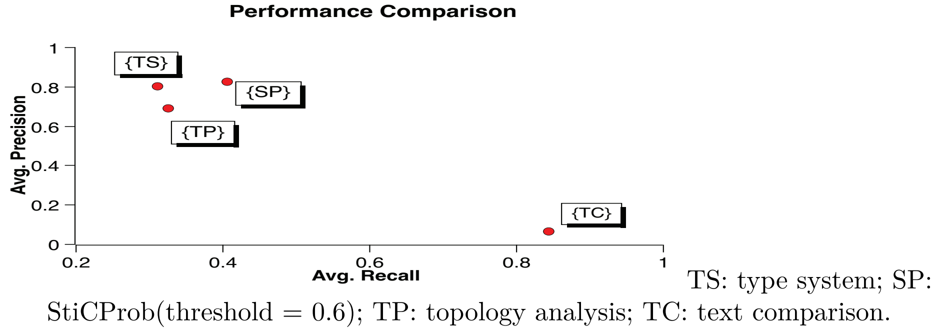
$$Precision = \frac{Correct\,recommendations}{All\,recommendations\,provided} \tag{4.2}$$

$$Recall = \frac{Correct\;Lines of annotated when stop}{Lines\;of\;code\;annotated\;in\;benchmark} \tag{4.3}$$

Another indicator for comparison is *f-measure*, which measures the performance of a model regarding to both precision and recall as:

$$f - measure = \frac{2 * Precision * Recall}{Precision + Recall} \tag{4.4}$$

The experimental result is shown in **Tab.**4.2, where *StiCProb* receives an average precision of 83% and an average recall of 41% on four subject systems with a threshold of 0.6. Furthermore, we compare *SticProb* with other approaches on all four systems with the same experimental settings as depicted in **Fig.**4.12. SticProb generally gives a better result comparing to type system (pre.:80%, recall:22%), topology (pre.:69%, recall:33%), and text comparison (pre. 6%, recall: 84%) average.



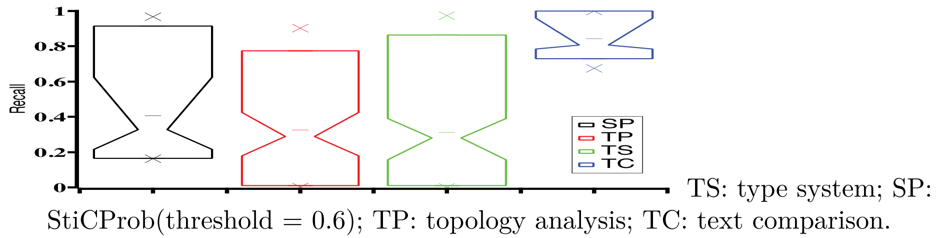
TS: type system; SP: StiCProb(threshold = 0.6); TP: topology analysis; TC: text comparison.

Figure 4.12: Performance Comparison on Subject Systems

For the *f-measure* shown in **Tab.**4.3, *StiCProb* returns a competitive performance in terms of both precision and recall.

Table 4.3: $f - measure$ on all approaches

	SP	TS	TP	TC
$f - measure$	0.55	0.45	0.44	0.12

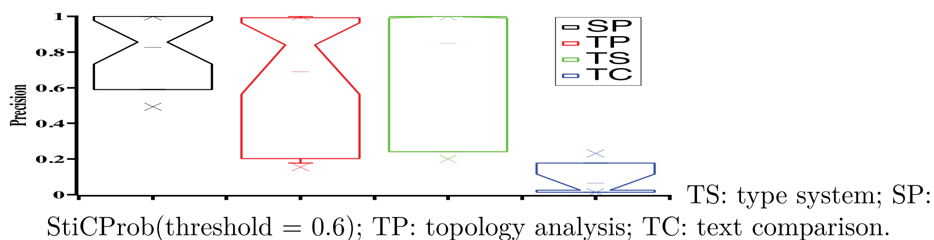


TS: type system; SP: StiCProb(threshold = 0.6); TP: topology analysis; TC: text comparison.

Figure 4.13: Method comparison using notched box plot in recall

In addition, as the notched box plot for recall shown in **Fig.**4.13, at 95% confidence interval of median, *StiCProb* performs better than both type system and topology analysis for most cases. From another aspect, the notched box plot for pre-

cision in **Fig.**4.14 indicates that *StiCProb* works better than both topology analysis and text comparison.



TS: type system; SP: StiCProb(threshold = 0.6); TP: topology analysis; TC: text comparison.

Figure 4.14: Method comparison using notched box plot in precision

Runtime Performance. We also evaluate the performance in terms of run-time. We test all algorithms on a Mac(10.12) machine with Intel i5 2.6GHz, 8G 1600 MHz DDR3, and targeting on Eclipse 4.5 with JRE 7. The result is shown in **Tab.**4.4.

Table 4.4: Runtime Performance (second)

	TS	SP	TP	TC
Pervalyer	1	2	2	71
MobileMedia	2	2	3	21
Lampiro	1	29	13	135
ArgoUML	254	1500	1980	5415

In summary, based on the recall performance, all methods could be ranked as TC (0.77,0.80,0.93) $\gg$ $\boldsymbol{SP}$ **(0.29,0.33,0.53)** $\gg$ TP (0.12,0.32,0.41) $\gg$ TS (0.12,0.21,0.37). Here, we use a three-element tuple $(first, median, third)$ to indicate the first quartile($first$), the median value($median$) and the third quartile($third$) of data.

For precision performance, TS (0.66,0.92,1) $\gg$$\boldsymbol{SP}$ **(0.71,0.85,0.96)**$\gg$ TP (0.42,0.84,0.95) $\gg$ TC (0.02,0.02,0.11). We can conclude that *StiCProb* could return a competitive

and stable performance comparing to others. However, *StriProb* sometimes spends extra time in generating binding and contexts, which could be a potential drawback.

4.7 Discussion

Beyond the default settings, we investigate how the two independent variables, seeds and threshold, influence the performance. Due to space restriction, we provide a general discussion on these factors and put the details on the project page[2].

4.7.1 Seeds

In our feature-mining process, the seeds could strongly influence the performance. In our experiment, we adopted first three items returned by FLAT^3. By increasing the number of seeds, the performance can hardly be improved and sometimes becomes significantly worse. After a careful inspection on seeds, we discovered the following principles, which could be used to guide developers in seeds selection. First, seeds recommended by FLAT^3 might not be correct, which causes the feature mining strategy performs poorly. That is, if the quality of seeds can be improved, the performance may improve. Second, the seeds in coarse granularity could improve the recall, but sometimes at the cost of precision.

In addition, a well-performed approach could be created by combining the returns from different tools. In this study, we highly rely on top seeds returned by FLAT^3. As the experimental experience tells us, FLAT^3 could return incorrect seeds, and even worse the incorrect seeds will lead to poor performance in feature mining. However, another strategy could somewhat prevent this problem, that is combining the results

[2] `http://www.chrisyttang.org/loong/`

by different tools. In the literature review, we listed several related tools beyond FLAT[3]. We could use the top n returns from different tools. Let us assume that we use four different tools, and for each tool we reserve the top 10 recommendations. Then, we rank all forty entities by occurrences. The first 5 items can be the input seeds. We also test this strategy using the project Prevalyer. Unfortunately, we found that it still identifies incorrect seeds. As a result, a better approach is strongly suggested, which is filtering the result with the help of a domain expert.

4.7.2 Threshold

In *StiCProb*, we select a `threshold` of 0.6 as the stopping criteria. That is, for an iteration, if all candidates cannot reach the threshold, the mining process for the current feature will stop. Intuitively, by setting a higher threshold, the precision can reach a higher value, and the recall drops down. However, we found that by increasing the threshold, the precision is not significantly improved. For example, in *Prevalyer*, with a change of threshold from 0.6 to 0.8, the precision merely increases to 85% from 83%. That is mainly due to the use of forward and backward probabilities. And it makes the *threshold* contributes less to the performance since the forward and backward probabilities are directly decided by the structure of the system.

4.7.3 Threats to Validity

Construct and Internal Validity. The measure of performance is dependent on the quality of benchmarks. The benchmarks are selected from systems that have been researched by others. Nevertheless, it is possible that the selected benchmarks might not be entirely accurate. The measurements on recall and precision are based

on line of code, which are intuitively reasonable.

External Validity. (1) Due to the relatively small number of cases selected and the size of subject systems (4-120KLOC), the experimental results could not be generalized to all systems. However, this is mainly because we can only select systems that have already been researched to obtain the benchmarks. In addition, the two systems (Lampiro and MobileMedia) that are initially developed as a product line system might bring bias on performance, considering their architectures could be optimised, like following certain design patterns, and might make feature mining approach performs well. (2) For each system, as seeds are decided using $FLAT^3$, it excludes the bias from selecting seeds by the experimenters. Moreover, the number of seeds and threshold used in our approach could affect the performance, but we have discussed their impact earlier. (3) To assess the performance, we use benchmarks from others' work, which eliminates the bias introduced by providing benchmarks on our own.

4.8 Chapter Summary

Product line engineering has been broadly adopted to developing applications with high customization at a low cost. To reduce the barrier in adopting product lines by migrating legacy software, we provide a novel approach named *SticProb* to extract related code fragments for feature concerned with a tool named *Loong*. *SticProb* uses the conditional probability to direct the feature mining process. Unlike all other approaches, *SticProb* can learn the environment of a programming element before annotating it to a feature. In this way, *SticProb* performs competitively in both precision and recall.

Chapter 5

Reengineering Features into Product Line Variants

With the approach presented in the previous chapter, we are now able to annotate code fragments with features. However, this procedure is only a virtual separation of features rather than separating into executable product variants[47]. Unfortunately, a straightforward approach to generating physically separated product variants could make the variants *invalid* or may not be what users expect. Here, *invalid* means the product variants could not be executed due to errors. In this chapter, we will propose an approach to reengineer annotated legacy system into product variants.

5.1 Overview

To successfully reengineer an annotated legacy application into a product line, three obstacles have to be overcome. First, the reengineering process for a legacy system to product variants could introduce unexpected syntax errors. Since the input configuration will hide irrelevant features and preserve features needed in the annotated legacy system to create a variant, syntax errors may result in *undisciplined annota-*

tion (see Sec.5.2.1 for an example). *Undisciplined annotation* represents the case that only parts of an AST node are annotated[62]. For example, given a `if` statement as `if (cond) stmt`, if only `cond` is annotated to a feature rather than the entire statement, it is considered as *undisciplined annotation*. Such syntactical errors are fatal to the variants and could make them invalid.

Second, product variants generated from an annotated legacy system may be ill-typed: i.e. lacking declarations of classes, methods, variables, or fields for references[35]. For example, a method m is annotated to a certain feature f; however, the access of m is not bound to any features. Therefore, unselecting feature f will make the access of m fail to find the declaration of m (see Sec.5.2.3 for a detailed example). This will introduce a type error, which means some types are unresolved in the system.

Third, the behaviour of some features may be interfered with the given configuration during reengineering. That is, a feature's behaviour in a product variant may be different from what it intends to do in the legacy (see Sec.5.2.2 for an example). The change upon code fragments during reengineering could make features perform unexpectedly and deviate from its intent.

Unfortunately, existing approaches in building software product lines from legacy systems have several limitations. For example, unsuitable for fine-granularity[63, 8, 57], behaviour preservations for feature are not discussed[63, 8, 96, 71, 50, 46], not able to achieve well-typed and cannot exclude syntactical errors [63, 96, 50, 57].

In this chapter, we develop a safe reengineering engine to guarantee syntax correctness, behaviour consistency and well-typed during the reengineering procedure. Unlike some works that target at locating features or concern separation, the ultimate goal of this work is generating valid product variants and also ensuring all

generated variants are syntactically correct (Sec. 5.4), behaviour-preserving (Sec. 5.5), and well-typed (Sec. 5.6). From the perspective of the compiler and type, each feature should not contain compiling errors, should execute correctly and be type safe. From an abstract level, the behaviour of each feature should be preserved.

Compared with other related approaches, our approach has the following advantages:

1. Our approach works directly on the source code, because a source code based approach could reduce the complexity of building a product line. It is worth noting that since the reengineering process from a legacy application to a software product line would be intractable for developers, who often have limited domain knowledge about the system, an aspect-oriented approach could increase the required efforts.

2. Rather than only keeping the product variant well-typed, our approach also explore the feature consistency, which means during the reengineering process, the features' implementation should be consistent with regard to behaviour preserving, syntactical correctness, and well-typing.

Contribution. We propose an adaptive procedure to guarantee feature consistency with the target during the reengineering procedure, which could avoid compiling errors as well as typing inconsistency and preserve features' behaviours.

We will discuss how our approach could overcome these limitations in a later section (Sec. 5.11.1).

Specifically, in this chapter, we make following contributions:

- To our best knowledge, this is the *first* work to describe the issue of feature consistency during reengineering.

- We develop a tool to support the procedure of keeping feature consistency for reengineering legacy applications to product lines.

- We conduct empirical studies on 9 real-world systems and discuss reengineering experiences in migrating a legacy system into a product line.

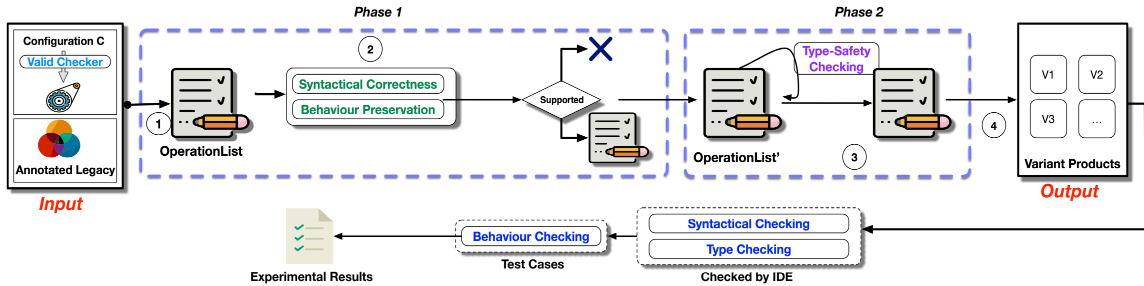


Figure 5.1: The process of transforming annotated legacy to product variants

We implement our method in an Eclipse plugin with over 36 KLOC Java code. For our context, *Syntactical correctness* module is implemented by extending the JDT framework in Eclipse, then the relations between programming elements and type checking system are built by referencing the implementation of tool *CIDE*[46] and a fact extractor *JayFX*[1] is used to build the *behaviour preserving* and *type-safety checking* modules.

Sec.5.2 presents several motivating examples to further illustrate the necessity of a novel approach and how our approach handle these problems. Sec. 5.3 gives an overview of our approach. Sec.5.4 to Sec.5.8 describes the strategy we adopted to

[1] JayFx: available at `http://www.cs.mcgill.ca/~swevo/jayfx`.

avoid syntactic error, inconsistency of features' behaviour and typing errors respect to underlying relations between features respectively. In addition, to evaluate our approach with nine test cases as described in Sec. 5.9. We test our approach with 176 configuration options and 37115 test cases in Sec. 5.10.

5.2 Motivating Examples

The ultimate goal of this chapter's work is reengineering a feature-annotated legacy system to product variants at the code base level with additional auxiliary functions to ensure syntax correctness, behaviour preservation and well-typed. However, this procedure might introduce unexpected errors in the variant products created. In this section, we use three examples with different concerns to motivate the importance and necessity of our work.

5.2.1 Syntax Error Example

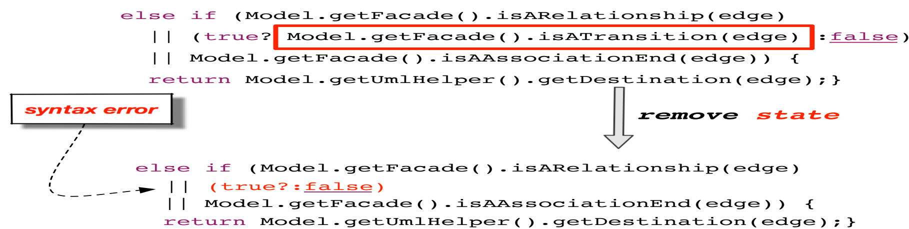


Figure 5.2: A syntax error example

Error. Fig.5.2, a code fragment from ArgoUML product line v1.4[2], shows a fragment of a transformation from an annotated legacy into a variant, where feature *state* is

[2] `http://argouml.tigris.org`.

disabled. The removal of segment `Model.getFacade().isATransition(edge)` will create a syntax error as presented in `true?false`. A correct transformation should be `true? Model.getFacade().isATransition(edge):false` to `false`. This is because, in the ternary expression `a ? b : c`, if `a` is `true`, `b` will be executed; otherwise `c`. In this example, since the condition is always `true` and the removal of positive path will always lead to the negative one, which means the correct transformation should result in `false` statement itself.

Explanation. This kind of error is raised when an AST node is partially annotated and the corresponding actions introduced by a configuration option (i.e. removing feature *state* in this example) will make the product variant incorrect.

5.2.2 Behaviour Inconsistent Error Example

```
Clause 1: class                          Clause 2: public static int
SocketChannel extends                    bytes_received = 0;
BaseChannel

SocketChannel.bytes_received++;   (box1)

if (sockInstream instanceof TlsInputStream == false) { (box2)
   BaseChannel.bytes_received++;
} else if (sockInstream instanceof TlsInputStream == true) {
   BaseChannel.bytes_received =
   SocketChannel.handler.getBytes_received();
}
```

Figure 5.3: A behaviour inconsistent example

Error. A behaviour inconsistent error is shown in Fig.5.3 from Lampiro product line[3], with two segments defined as *box1*, and *box2* respectively. Different background colours show these segments are annotated to three features. The error arises when removing the feature for *box1*.

Explanation. The code fragment in Fig.5.3 gives two clauses:

[3] `http://lampiro.bluendo.com/`.

1. class `SocketChannel` is a subclass of `BaseChannel`;
2. global variable `bytes_received` is defined in class `BaseChannel` and inherited by `SocketChannel`.

Therefore, both *read* and *write* on the global variable `bytes_received` from *box1* and *box2* will access the *same* variable. Then, *box1* will *update* the value of `bytes_received` and the removal of *box1* will make this update invisible and affect the value of `bytes_received` in *box2*.

5.2.3 Type Error Example

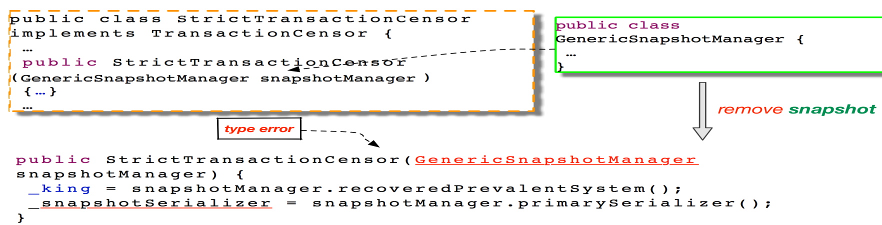


Figure 5.4: A type error example

Error. A type error arises when an object is presented with an unexpected type or the type of the object is not available[46]. For example, in Fig.5.4, class `StrictTransactionCenso` is annotated to feature *Censor* and class `GenericSnapshotManager` is associated with feature *Snapshot*. Therefore the removal of feature *Snapshot* will make a type error in class `StrictTransactionCenso`.

Explanation. If feature *Snapshot* is unselected, then the

class `GenericSnapshotManager` will be removed, which will make the variable "snapshotManager"'s type `GenericSnapshotManager` not found.

The above errors are real-world errors collected by using tool *CIDE* to create product variants. And the tool *CIDE* implements the $\texttt{LJ}^{A}R$ approach, which will be introduced later in Section 5.10. Given the samples above, without a complete approach to inspect the reengineering, it would be easy to introduce errors to the target variants and makes the reengineering process unsuccessful.

In this chapter, we propose a new strategy with a set of constraints to resolve these potential errors, which can be used to assist the process of reengineering an annotated legacy to product variants.

5.3 Configurable AST: Outline and Background

5.3.1 Procedure At A Glance

The reengineering process starts from a virtual feature separation to a series of product variants [47]. Specifically, "virtual feature separation" indicates that the features in the legacy system are represented in annotations, rather than being physically separated into methods, classes, or even packages. For example, background colours could be used to annotate features in the program, and we assign different background colours to show different features in the IDE. However, those implementations are merely distinguished from different colours, not physically separated. Fig.5.1 shows the procedure of our approach.

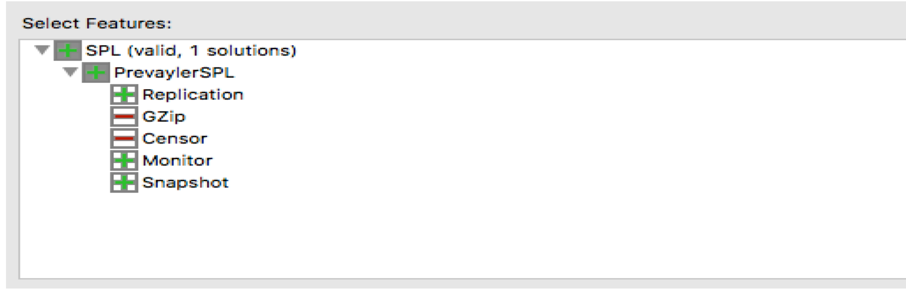


Figure 5.5: The configuration sample

Input: The input for reengineering an annotated legacy to product variants should be a configuration and the annotated legacy. In the configuration, the stakeholder should specify those features that compose a product variant. For example, as shown in Fig. 5.5, a configuration given as `Replication` ∧ ¬`GZip` ∧ ¬`Censor` ∧ `Monitor` ∧ `Snapshot` could be adopted to create a product variant with only features `Replication`, `Monitor` and `Snapshot` selected. Specifically, the input configuration will be verified first and any invalid configuration will be rejected (Sec. 5.9.1).

Output: The output for this procedure is a variant product according to the input configuration.

STEP 1: The configuration will be transformed into a set of operations, as *OperationList* in Fig.5.1, on the annotated legacy (Sec. 5.3.3).

STEP 2: For each operation generated in **STEP 1**, our approach performs a two-phase checking. **STEP 2** shows the first phase and **STEP 3** shows the second phase. In the first phase (**STEP 2**), *syntactical correctness* and *behaviour preservation* module will be active. The *syntactical correctness* module is used to ensure that the process will not introduce syntax errors. The *behaviour preservation*

module ensures behaviours of the features in target variant product are not changed. Each operation will be checked with the *syntactical correctness* module (Sec. 5.4), and *behaviour preservation* module (Sec. 5.5) respectively.

STEP 3: In the second phase, the *type-safety checking* module is used. The *type-safety checking* module, which will be active to remove actions that will lead to typing errors.

STEP 4: All approved and checked actions will be executed on the annotated legacy to create the product variant as the output (Sec.5.9.1).

The experimental results are collected based on the product variants generated by our approach. The performance of our *syntactical correctness* and *type-safety checking* modules are checked with IDE. To access the *behaviour checking* module, unit test cases generated by *evosuite* are used.

5.3.2 Process Modelling

Recall our target is to create an engine that enables safe reengineering from an annotated legacy to product variant. On the one hand, an annotated legacy could be represented by a combination of all features' implementation as:

$$\mathcal{T} = \mathcal{T}_{F_1} + \mathcal{T}_{F_2} + \cdots + \mathcal{T}_{F_n} + \mathcal{T}_{F_{core}}, \tag{5.1}$$

where $\mathcal{T}_{F_i}$ is the implementation of feature F_i and $\mathcal{T}_{F_{core}}$ represents the base architecture. On the other hand, from the implementation perspective, it could be deemed as a combination of AST nodes as:

$$\mathcal{T} = t_0 + t_1 + \cdots + t_n, \tag{5.2}$$

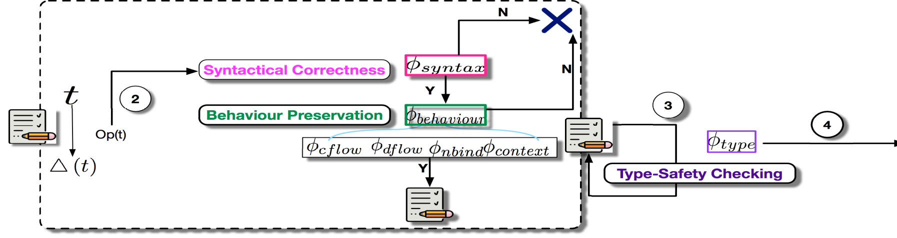


Figure 5.6: The overview of changing process

where t_i represents an AST node and $t_i \neq t_j$ for any $i \neq j$. Therefore, given a configuration $c = \{F_1, F_3\}$[4], the change at modularity perspective could be considered as a projection from $\mathcal{T}$ to $\mathcal{T}_c$, where $\mathcal{T}_c = \mathcal{T}_{F_1} + \mathcal{T}_{F_3}$. However, changes at the code level is represented as the changes on some AST nodes. Technically, the change is denoted by the equation (5.3).

$$\mathcal{T}_c = \Delta(t_0) + \Delta(t_1) + \cdots + \Delta(t_i) + t_j + \cdots + t_n, \tag{5.3}$$

where $\Delta(t_i)$ represents corresponding AST node of t for the given configuration c. As shown in Fig. 5.1, the *OperationList* denotes the change from $t_i\,(i = 0 \cdots n)$ to $\Delta(t_i)\,(i = 0 \cdots n)$.

5.3.3 Transforming a Configuration into Operations on AST

Moving on, we will illustrate the approach to translate a given configuration into a series of operations on the AST nodes for the annotated legacy. Basically, the annotated legacy contains all features' implementation and the configuration gives the selected features and unselected features. Therefore, our approach remove all

[4] For brevity, we just show the features selected in the configuration. A complete configuration could contain all features and use labels to distinguish enabled and disabled features.

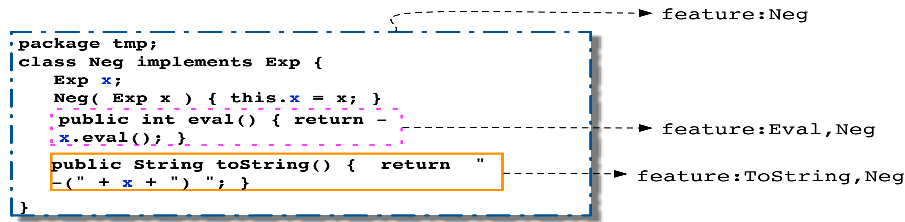


Figure 5.7: The example of code-based reengineering

unselected features' implementation from the annotated legacy. For example, in Fig. 5.7, there are three features associated with the code snippet. If the feature `Neg` is unexpected/unselected in a variant, then the method `eval()` should be removed in the legacy system to create the variant. As a result, our approach removes the corresponding code for the feature `Neg`.

Therefore, $\Delta(t)$ could be "null" means that AST node t is removed in the variant product.

5.3.4 From t to $\Delta(t)$

Recall the process defined in Fig.5.1, in **STEP 2**, each operation (t to $\Delta(t)$) on the annotated legacy will pass through three checking modules. In detail, the operation will only be approved if and only if it passes the following conditions.

1. Syntactical correctness is ensured by ϕ_{syntax} constraints. The detailed description of ϕ_{syntax} is presented in Sec.5.4;

2. To ensure feature behaviours during reengineering, we check the control-flow (implemented by ϕ_{cflow}), data-flow (ϕ_{dflow}), name-binding (ϕ_{nbind}) and context-sensitive relations ($\phi_{context}$) for t represented by $\phi_{behaviour}$ as presented in Sec.5.5;

3. During the transformation, the typing should be assessed by ϕ_{type} as presented in Sec.5.6.

Furthermore, from **Sec.**5.4 to 5.7, we explore corresponding conditions and constraints which should be satisfied when a given AST node t is changed to $\Delta(t)$. The constraint will be described with the following format:

$$\phi_{cons}\left(t \rightarrow \Delta\left(t\right)\right) = \begin{cases} \left\{\left(t', op_{t'}\right) \mid op_t\right\}; & \text{syntax/behaviour} \\ \text{typing-rules}; & \text{type} \end{cases}, \tag{5.4}$$

where *cons* could be *syntax*, *behaviour* and *type*.

The *syntax* and *behaviour* constraint, represented as $(t \rightarrow \Delta\left(t\right))$, shows the additional operations needed to make these constraints satisfied. In detail, op_t means the operations needed on t to become $\Delta\left(t\right)$ under this constraint, which will be used to update *OperationList* in Fig.5.1. It will return a mapping set as $(t', op_{t'})$ for a given condition op_t, where op_t represents the operations on node t. In addition, t' is the affected AST nodes, and $op_{t'}$ is the corresponding operations on t'. As mentioned in Sec.5.3.3, the operation on the AST node should be "removed", since the associated AST nodes should be removed from the system. Therefore, equation 5.4 should be rewritten as

$$\phi_{cons}\left(t \rightarrow \Delta\left(t\right)\right) = \begin{cases} \left\{\left(t', op_{t'}\right) \mid rm\right\}; & \text{syntax/behaviour} \\ \text{typing-rules}; & \text{type} \end{cases}, \tag{5.5}$$

where rm represents "remove". In addition, the return of ϕ_{cons} could be "null", which means it is impossible to satisfy this constraint and this operation (from t to $\Delta(t)$, also removal of t) should be prevented. However, the *type* constraint will

be a set of typing rules to prevent insecure operations that cause type errors, to be discussed in Sec.5.6.

5.4 Configurable AST: Syntactical Correctness

The transformation from a legacy to a product variant could be achieved by taking actions upon AST nodes. In a corpus of a syntactically correct program, the manipulation on AST nodes should be executed safely and the program generated after transformation should also be syntactically correct. To ensure a syntactically correct AST, also known as valid AST fragments, we adopt the *valid AST fragment rules* presented in [73] as shown in **Tab.**5.1. The new rules defined in this work are highlighted.

The main motivation for proposing the valid AST fragment rules strategy is to ensure syntactical correctness. As mentioned in Sec. 5.3.2, the configuration will be translated to a set of operations on AST nodes. These operations may break the structure or change the AST node. The *valid AST fragment rules* will private a set of valid AST fragment rules to construct an AST node without introducing syntactical errors. For example, the syntax error in the motivating example shown in Sec.5.2.1 is due to the remaining AST nodes in `true?Model.getFacade()...:false` are not well-rewriten after the removal of `Model.getFacade().isATransition(edge)`.

In **Tab.**5.1, we extended the "valid AST fragment rules" in [73] by adding auxiliary rules for generating valid fragments. Specifically, it examines the AST node, and considers all possible valid combinations of its children nodes to form a syntactically correct AST node. Specifically,

Table 5.1: Valid AST Fragments Rules

Syntax	Valid AST Fragments
if (s) **then** s′ **else** s″	**If** → s **If** → s,s′ **If** → ¬s,s″(**rewrite**) **If** → s,s′,s″
do{$\overline{\mathrm{s}}$} **while**(s) **while**(s) **do**{$\overline{\mathrm{s}}$}	**While/Do** → s,$\overline{\mathrm{s}}$
for(s;s′;s″):$\overline{\mathrm{s}}$	**For**→s′,s″ **For**→s,s′,s″ **For**→s′,s″:$\overline{\mathrm{s}}$ **For**→s,s′,s″:$\overline{\mathrm{s}}$
switch(v):**case**($\overline{\mathrm{s}}$) {$\overline{\mathrm{s}'}$}	**Switch**→v **Switch**→v,$\overline{\mathrm{s}}$,{$\overline{\mathrm{s}'}$} **Switch**→v,$\overline{\mathrm{s}_0} \in \overline{\mathrm{s}}$, {$\overline{\mathrm{s}'_0}$}(**rewrite**)
try{$\overline{\mathrm{s}}$} **catch**(vd) {$\overline{\mathrm{s}'}$}|**finally**{$\overline{\mathrm{s}''}$}	**Try**→ {$\overline{\mathrm{s}}$} , {$\overline{\mathrm{s}'}$} **Try**→ {$\overline{\mathrm{s}}$} , {$\overline{\mathrm{s}''}$} **Try**→ {$\overline{\mathrm{s}}$} , {$\overline{\mathrm{s}'}$} , {$\overline{\mathrm{s}''}$}
TernaryOp: x **Op** y **Op** z	y z **TernaryOp**→x,y,z
M::=C m($\overline{\mathrm{C\ v}}$) {$\overline{s}$ **return** y; }	M→C m($\overline{\mathrm{C'\ v'}}$) {$\overline{\mathrm{s}'}$ **return** y; } M→C m($\overline{\mathrm{C\ v}}$) {$\overline{s}$ **return** y; }
inheritance C◁D ◁E	C◁D ◁E → C(**rewrite**) C◁D ◁E → D(**rewrite**) C◁D ◁E → E C◁D ◁E → C◁D C◁D ◁E → D◁E C◁D ◁E → C◁E(**rewrite**) C◁D ◁E → C◁D ◁E

1. *if*: the *if* statement contains four valid fragments by combining different components in the *if* statement. Especially, the *if* statement could be reorganised to

if $(\neg s)$ *then* s'';

2. *do ... while .../while ...*: the structure of *do* and *while* statements should not be changed and changing of s or s' will make the entire statement invalid or the behaviour of the statement changed;

3. *for*: as for the valid extension upon *for* statement, the execution condition s' and loop update s'' should be preserved at least. Otherwise, it may lead to a potential problem that the loop is invalid and cause an infinite loop;

4. *switch ... case...*: the valid AST fragments for *switch-case* statement should be the change upon any case. For example, if a *case* is removed in the statement, its corresponding statement should be deleted as well;

5. *try ... catch ... finally ...*: the valid AST fragments for *try-catch-finally* statement could contain the following cases: (1) a valid variant of *try-catch-finally* statement could be one without *finally* branch; (2) remove some *catch* branches should also be a valid fragment; and (3) itself (nothing change);

6. *ternary operation*: the ternary operation should be in a format of *cond ? x : y*, therefore, the valid AST fragments should either be x or y;

7. *method*: as for a method declaration, a valid fragment could be removing some parameters and rewriting the method body correspondingly; and

8. *class inheritance*: direct and indirect inheritance relations between classes and

interfaces could be altered via rewriting the attributes and functions in classes/interfaces as shown in **Tab.** 5.1.

Example. Given an *if* statement, as *if*(s)*then* s′ *else* s″, we can compose four valid AST fragments: (1) a single s, with the *then* statement empty; (2) s,s′: for having if condition and then statement; (3) s,s″: a specific case, in which s″ is the else statement, then it will be rewritten to *if* $\neg s$ *then* s″.

However, not all AST nodes can be expanded to have valid AST fragments, as some are atomic and there is no way to use its children to compose a valid AST with its original type. For example, for the field access $v = x.f$, any combination of children in this AST node will introduce a syntax error.

As a result, we define a syntactical constraint ϕ_{syntax} as:

$$\phi_{syntax} = \begin{cases} \{(par\,(t)\,, Rewrite_{rule}) \mid rm\} & \text{valid actions} \\ \text{null}; & \text{others} \end{cases}, \tag{5.6}$$

where $par\,(t)$ represents the parent AST node of t and $Rewrite_{rule}$ means rewrite the AST node $par\,(t)$ based on the "valid AST fragment" rule. That is, to remove an AST node t, the corresponding operation should be conducted on t's parent node and the operation is based upon the rules in **Tab.**5.1; otherwise, the operation should be rejected.

5.5 Configurable AST: Behaviours Preserving

Operations on an AST node demand the full checking on constraints and dependencies of the AST node being processed since we expect to reengineer an annotated legacy into a product line without introducing any parser errors. However, the brute

force approach for removing or adding an AST node is risky, because it can break the AST structure and affect its behaviours. To preserve the behaviours during the reengineering, we need to check the control flow, data flow, type reference and context-sensitive relations. Specifically, control flow, data flow, and type reference are used to ensure behaviour preserving.

5.5.1 Assumption

In the coming sections (Sec. 5.5.2 to Sec. 5.5.5), we intend to discover the corresponding actions that should be taken, in order to satisfy behaviour preserving, when an action is taken upon an AST node t.

5.5.2 Control Flow Constraint

The constraints with control-flow might not be strict and strongly dependent on the structure of code.

Listing 5.1: Sample Code: Variability-aware Control Flow Constraint

```
if(a < 0)
   b = 1;
else
   b = 2;
c = 2;
```

As shown in the above sample code, the statement $b = 1$ is conditionally dependents on the value of a in $a < 0$, and $b = 1$ depends on $a \geq 0$. The statements $b = 1$ and $b = 2$ are defined as *control dependent branch* and c = *2* is *control independent*[58]. If c is a *control condition*, we define the control-flow constraint as:

$$\phi_{cflow} = \begin{cases} \{(dbranch, rm) \mid rm\}\,; & t = c \\ \{(dbranch, rm) \mid rm\}\,; & t \overset{du}{\rightarrow} c \\ \quad rewrite; & \text{t} \in \text{others} \end{cases}, \tag{5.7}$$

where

$$dbranch = \bigcup_{t'} t \xrightarrow[ctdep]{cflow} t'. \tag{5.8}$$

If t is a *control condition*, the remove operation on t will require the corresponding removal of *dbranch*. The symbol *dbranch* represents all *control dependent branch* under t. In addition, if t has the def-use relation with the condition c, the value used in c is defined in t, the corresponding changes should include removing the affected branches.

Furthermore, we list several representative programming elements affected by the variability-aware control flow constraints.

If-then-else. As the diagram shown in Fig. 5.8, the control flow graph for a basic *if-then-else* statement, the control flow constraint on *if* statement will ensure that the change on the *if* structure will not result in errors. On the left side of Fig. 5.8, it shows the control flow graph and the operations, with corresponding actions represented on the right side.

While/Do. As for *do ... while(cond)* and *while(cond)...* statement, the removal of the condition will make the whole block invalid and our strategy is to remove the entire statement.

For. As the *For* statement's CFG shown in Fig.5.9,

1. Removal of the condition *cond*: it will make the entire *for* loop invalid or lead

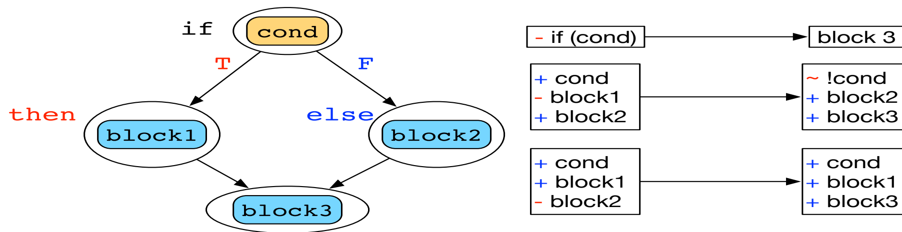


Figure 5.8: If-else statement

to an infinite loop; therefore, to satisfy the control-flow constraint, the entire loop should be removed.

2. Removal of the loop update *update*: it could make the *for* loop into an infinite loop; therefore, the entire loop should also be removed.

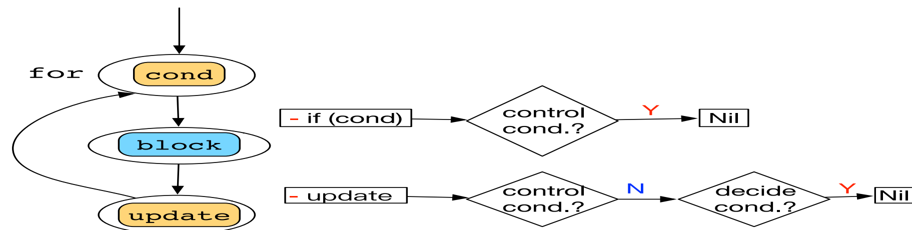


Figure 5.9: For statement

Swith-Case. As the diagram of *switch... case ...* statement shown in Fig. 5.10, it contains two main cases:

1. the removal of any *case*: if one case is removed, then it can be removed safely; and

2. the removal of the *switch* condition *cond*: this will make the entire *switch-case* statement invalid, and the entire statement should be removed.

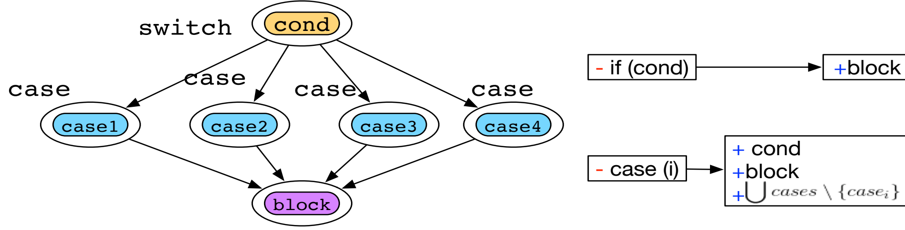


Figure 5.10: Switch statement

Example. For our sample code presented earlier, removing if-condition $a < 0$ will remove the corresponding statement $b = 1$ and the else-branch should also be changed accordingly.

5.5.3 Data Flow Constraint

In data flow analysis, the def-use relation is the main concern, in which the def-use(du) relation represents the link from a definition to its usage. However, the def-use relation is not sufficient to capture pointer information. Therefore, to preserve behaviour during reengineering, a flow-insensitive approach for data-flow analysis is required to capture the transformation[97]. Hence, our flow-insensitive data-flow constraint is computed by:

$$\phi_{dflow} = \begin{cases} \{(\bigcup t'; t, rm) \mid rm\}\,; & t \in def, t' \notin c, t \xrightarrow{du} t' \\ \{null \mid rm\}\,; & t \in def, t' \in c, t \xrightarrow{du} t' \\ \{(\bigcup t'; t, rm) \mid rm\}\,; & t \in use, t' \notin c, t' \xrightarrow{du} t \\ \{(t, rm) \mid rm\}\,; & t \in use, t' \in c, t' \xrightarrow{du} t \\ \{null \mid rm\}\,; & t, t' \in use, t' \in c, t \prec t' \\ \{(t, rm) \mid rm\}\,; & t, t' \in use, t' \in c, t' \prec t \\ \{(\bigcup t'; t, rm) \mid rm\}\,; & t, t' \in use, t' \notin c \end{cases}, \tag{5.9}$$

where $t \in def$ represents that t is a definition and $t \in use$ means that t is an usage. The def-use relation $t \stackrel{du}{\rightarrow} t'$ represents the def-use relation starts from t and ends with t'. The symbol $t' \prec t$ represents t is executed after t'. Concretely, we discuss the following cases.

1. $t \in def, t' \notin c, t \stackrel{du}{\rightarrow} t'$: this case represents that t will be removed and t's access t' will be removed in the variant product. The removal of t and t' will be approved, since t' is the use of t in the program. Removing a variable and its use will not introduce syntax errors.

2. $t \in def, t' \in c, t \stackrel{du}{\rightarrow} t'$: this case means t will be removed and t's access t' will be will be preserved in the variant product. This operation will be rejected, because it will make the declaration of t not available;

3. $t \in use, t' \notin c, t' \stackrel{du}{\rightarrow} t$: same as the first case;

4. $t \in use, t' \in c, t' \stackrel{du}{\rightarrow} t$: the case represents a variable/field's declaration (t') is preserved in the variant product and its use is removed. This operation will be approved due to the declaration is preserved;

5. $t, t' \in use, t' \in c, t \prec t'$: if t is executed before t' and t' should be preserved, then the removal of t will make $\underline{t'}$'s value incorrect. This is because t will change the value that used in t'. Therefore, this operation will be rejected;

6. $t, t' \in use, t' \in c, t' \prec t$: if $t' \prec t$, then the removal of t will not affect the value of t' as $\{(t, rm) \mid rm\}$ show;

7. $t, t' \in use, t' \notin c$: as t' and t are both *use* and t' will be removed, then they will be removed.

Example. Recall our example in Fig.5.3, both `bytes_received` in *box1* and *box2* are *use* of object `bytes_received`. If the feature of *box1* is unselected ($t \notin c$) and *box2* is preserved($t' \in c$) in the configuration, the operation on *box1* will be rejected. Because base on equation 5.9, approval of the operation will change the behaviour of code in *box2*.

5.5.4 Name Binding Constraint

The name binding constraint for a given AST node t could be represented as:

$$\phi_{nbind} = \begin{cases} \{null \mid rm\}; & t \in bind, t' = use(t), t' \in c \\ \{(t; t', rm) \mid rm\}; & t \in bind, t' = use(t), t' \notin c \\ \{(t, rm) \mid rm\}; & t \notin bind, t' = bind(t), t' \in c \\ \{(t; t', rm) \mid rm\}; & t \notin bind, t' = bind(t), t' \notin c \end{cases}, \tag{5.10}$$

where the function *bind(t)* will return the binding of an AST node t. The binding of an AST node could be a variable, a field, a method, an interface, a class, or a type declaration. If t itself is a declaration, *bind(t)* will be t; if t is an access of a declaration, *bind(t)* will give the declaration. And *use(t)* will return all accesses of this binding.

Example. If a class t's declaration should be removed in a configuration, but t's access ts is persevered, then t should be preserved to make ts points to class t correctly.

5.5.5 Context-sensitive Constraint

Context-sensitive analysis distinguishes the calling context from the same caller. Here, we propose a variability-aware context-sensitive constraint, which encapsulates context-sensitive analysis in a variability-aware context. This relation is proposed to distinguish the calling context for methods and ensure that features' behaviours are preserved under each calling context. For example, given a method m and its call sites cs_1 and cs_2, s and cs_1 are annotated to the same feature f_1 and cs_2 belongs to feature f_2. If feature f_1 is unselected and f_2 is selected in a configuration, then by default m and cs_1 will be removed in the variant application. However, our context-sensitive constraint will prevent the removal of method m due to cs_2 is preserved. This can only be achieved by using context-sensitive analysis. Context-insensitive analysis is not qualified for our purpose.

Our variability-aware context-sensitive constraint is defined as equation (5.11).

1. t is a call site and its callee function t' is preserved in the variant product. Operations on t (change t into $\Delta(t)$) can be approved, because t' will not affect the operations on t;

2. t is a call site and its callee function t' is not preserved in the variant product. Operations on t can be approved and removal of t' can also be approved;

3. t is a function and its call site t' is preserved in the variant product. Because t's call site t' is preserved in the variant product, then the function t is present in the variant. Therefore, the operations on t should be blocked;

4. t is a function and its call site t' should not be preserved in the variant. Op-

erations on t can be approved because its call site t' does not present in the variant;

5. t is a parameter in function m, and m is preserved in the variant. As shown in listing 5.2, the removal of parameter will lead to a rewrite of the function;

6. t is a parameter in function m, and m is removed in the variant. Operations on t can be approved because its corresponding function m is removed.

Listing 5.2: Edge constructor in Edge.java

```
public Edge(Vertex the_start,
 Vertex the_end, int aweight){
 start = the_start;
 end = the_end;
 weight = aweight;
}
```

$$\phi_{context} = \begin{cases} \{(t, rm) \mid rm\}; & t \in caller, t' \in c \\ \{(t, t', rm) \mid rm\}; & t \in caller, t' \notin c \\ \{null \mid rm\}; & t \in func, t' \in c \\ \{(t, t', rm) \mid rm\}; & t \in func, t' \notin c \\ \{(m, rewrite) \mid rm\}; & t \in par., m \in fun, m \in c \\ \{(t, rm) \mid rm\}; & t \in par., m \in fun, m \notin c \end{cases}, \tag{5.11}$$

Example. Assuming a method m is annotated to feature f, and it has two callers c_1 and c_2 associated to feature f and f' respectively, then the method m should not be removed in presence of the removal of feature f, since the caller c_2 in f' requires the existance of m.

5.5.6 Putting All Pieces Together

In summary, to preserve the behaviour during the reengineering the transformation should satisfy the following constraints.

$$\phi_{\mathbf{behaviour}} = \phi_{\mathbf{cflow}} \wedge \phi_{\mathbf{dflow}} \wedge \phi_{\mathbf{nbind}} \wedge \phi_{\mathbf{context}}. \tag{5.12}$$

Considering our *behaviour preservation* module will try to guarantee that the operation on an AST node will not interfere features' behaviour in the variant product, an operation on an AST node can be approved if and only if it is possible to satisfy all constraints as presented in equation (5.12).

5.6 Configurable AST: Type Checking

Type constraints are collected from the perspective of preventing ill-typed AST node. To avoid creating ill-typed AST nodes during transformation, we define several rules for type checking any AST node t in the annotated legacy and its mapping AST node tc, if exists, in the product variant under configuration c. As the reengineering procedure from an annotated legacy to a product variant is achieved by a series of operations on AST nodes, tc could be "null", which means t is removed during the reengineering process. Therefore, we mainly do the checking where tc is reachable. Specifically, our type checking rules are inspired by the rules defined in [46, 44, 35]. On the constrast, we are interested in conducting type checking for both t and tc. That is, our type checking ensures that transforming from t to tc will not lead to any typing errors.

5.6.1 $\phi_{\mathbf{T-VAR}}$

The typing of variables is variability-unaware to annotation, with the transformation from t to tc, the type for tc could be extended to C' as C' is a subtype of C. Therefore, it can be described as

$$\frac{\dfrac{t : C \in \Gamma}{\Gamma \vdash t : C} \qquad tc : C \in \Gamma \qquad C <: C'}{\Gamma \vdash tc : C'}$$

5.6.2 $\phi_{\mathbf{T-FIELD}}$

For a field access, including field read and write, $x_0.f_i$ (represents t in our context) is transformed to $xc_0.fc_i$ (tc). It requires to look up the annotation of this access $AT\,(x_0.f_i)$ and the annotation of $AT\,(xc_0.fc_i)$) in $xc_0.fc_i$.

Therefore, we do a three-step mapping:

1. *The target instance xc_0 should be well-typed*: as the process shown in $\phi_{\mathbf{T-VAR}}$;

2. *Get all fields with desired feature $\mathcal{A}'$*: within the field list, the fields annotated to A' should be extracted. Here, function *filter*(A, B) returns those members that, within A, satisfy condition B. Function *fields*(A) returns the fields in A; and

3. *Filter the target field fc_i*: search target field fc_i in the field list $\overline{\mathrm{C}_0'\ f}$, and ensure it is well-typed.

$$\frac{\Gamma \vdash xc_0 : C_0' \qquad AT\,(x.f_i) = \mathcal{A} \qquad \mathcal{A} \Rightarrow \mathcal{A}' \qquad filter\,(fields\,(C_0')\,,\mathcal{A}') = \overline{C'\ f} \qquad AT\,(C_i', fc_i) \Rightarrow \mathcal{A}'}{\Gamma \vdash xc_0.fc_i : C_i'}$$

5.6.3 $\phi_{\mathbf{T-INVK}}$

Type checking for method invocations $xc_0.mc\,(\overline{yc})$ (tc in our context) is similar to checking a normal method invocation. In detail, it contains the following steps.

1. Method mc's type is filtered and obtained by *mtype* function;
2. The invocation parameters of ms should match the parameter declarations $(\overline{yc})$ and this mapping should also support the *override* function $(\mathcal{A} \Rightarrow \left(AT\,(\overline{yc}) \equiv AT\,\left(\overline{D'}\right)\right))$; and
3. The type of parameters should be checked as well.

$$\frac{\begin{array}{c}\Gamma \vdash xc_0 : C_0' \qquad AT\,(x.m(\overline{y})) = \mathcal{A} \qquad \mathcal{A} \Rightarrow \mathcal{A}' \qquad mtype\,(m, C_0, \mathcal{A}) = \overline{D} \rightarrow C \\ mtype\,(mc, C_0', \mathcal{A}') = \overline{D'} \rightarrow C' \qquad \Gamma \vdash \overline{y} : \overline{C} \qquad \overline{C} <: \overline{D} \\ \Gamma \vdash \overline{yc} : \overline{C'} \qquad \overline{C'} <: \overline{D'} \qquad \mathcal{A}' \Rightarrow \left(AT(\overline{yc}) \equiv AT(\overline{D'})\right) \qquad AT\,(\overline{yc}) \Rightarrow \mathcal{A}'\end{array}}{\Gamma \vdash xc_0.mc\,(\overline{yc}) : C'}$$

5.6.4 $\phi_{\mathbf{T-NEW}}$

Type checking (from new $C\,(\overline{y})$ to new $(C_c\,(\overline{yc})))$ for typing object creation contains the following steps:

1. The target class C_c should be annotated to $\mathcal{A}'$;

2. The provided parameters for C_c's constructor should be matched with expected parameters as $\mathcal{A}' \Rightarrow \big(AT(\overline{yc}) \equiv AT(\overline{D_c\ f_c})\big)$; and

3. In addition, the types of parameter and constructor are checked.

$$\frac{\begin{array}{c} AT(\text{new } C(\overline{y})) = \mathcal{A} \\ \mathcal{A} \Rightarrow \mathcal{A}' \qquad AT(\text{new } C_c(\overline{yc})) = \mathcal{A}' \qquad \text{filter}(fields(C_c), \mathcal{A}') = \overline{D_c\ f_c} \qquad \Gamma \vdash \overline{y} : \overline{C} \\ \overline{y} : \overline{C} \vdash \overline{yc} : \overline{C_c} \qquad \overline{C_c} <: \overline{D_c} \qquad \mathcal{A}' \Rightarrow AT\,(C_c) \qquad \mathcal{A}' \Rightarrow \big(AT(\overline{yc}) \equiv AT(\overline{D_c\ f_c})\big) \end{array}}{\Gamma \vdash \text{new } C_c\,(\overline{yc}) : C'}$$

5.6.5 $\phi_{\textbf{CAST}}$

For casting, we check two cases in terms of changes:

Case (1) $(C)\,x \rightarrow (C_c)\,x$: It means that the casting target has been changed to (C_c). The expression of T-UCAST(left) and T-DCAST(right) can be represented as follows ($\underline{AT(C_c) = \mathcal{A}}$ and $\underline{AT(C_c x) = \mathcal{B}}$):

$$\frac{\begin{array}{c} \Gamma \vdash x : C \\ C <: C_c \qquad \mathcal{B} \Rightarrow \mathcal{A} \end{array}}{\Gamma \vdash (C_c)\,x : C_c} \qquad \frac{\begin{array}{c} C_c <: C \qquad C_c \neq C \\ \Gamma \vdash x : C \qquad \mathcal{B} \Rightarrow \mathcal{A} \end{array}}{\Gamma \vdash (C_c)\,x : C_c}$$

Case (2) $(C)\,x \rightarrow (C)\,xc$: It means that the casting object has been changed to xc. The change that occurs to a variable should be type checked to confirm it is still compatible with the original type C for T-UCAST(top) and T-DCAST(bottom) as follows:

$$\frac{\begin{array}{c} \Gamma \vdash x : D \qquad D <: C \\ AT\,((C)xc) \Rightarrow AT\,(C) \end{array}}{\Gamma \vdash (C)\,xc : (C)} \qquad \frac{\begin{array}{c} \Gamma \vdash x : D \\ C <: D \qquad C \neq D \\ AT\,((C)xc) \Rightarrow AT\,(C) \end{array}}{\Gamma \vdash (C)\,xc : (C)}$$

5.6.6 $\phi_{\textbf{METHOD}}(M_c$ **OK in** $C_c)$

The method typing constraints check the change on the M to (M_c) with the following checking rules:

1. The *override* function gives a method's parameter type $\overline{C}$, return type C $(\overline{C} \rightarrow C)$ and a super class D. If M_c is an overriding function, it should be checked in $\text{override}\big(mc, D_c, \overline{C_c} \rightarrow (C_c)_0, \mathcal{A}'\big)$;

2. The class that corresponds to the return type of M_c should be checked as $(C_c)_0$; and

3. Each parameter in method M_c should be well-type as $AT\left(\overline{C_c\ xc}\right) \Rightarrow AT\left(\overline{C_c}\right)$;

$$\frac{\begin{array}{c} M_c = (C_c)_0\ mc\left(\overline{C_c\ x_c}\right)\{\text{return } tc_0;\} \\ AT\,(M) = \mathcal{A} \qquad AT\,(M_c) = \mathcal{A}' \qquad \mathcal{A} \Rightarrow \mathcal{A}' \\ AT\left(\overline{C_c\ xc}\right) \Rightarrow \mathcal{A}' \qquad filter\left(\overline{C_c\ xc}, \mathcal{A}'\right) = \overline{D_c\ yc} \qquad \overline{yc} : \overline{D_c}, this : C_c \vdash tc_0 : (E_c)_0 \\ (E_c)_0 <: (C_c)_0 \qquad CT\,(C_c) = \text{class } C_c \text{ extends } D_c\{....\} \\ \text{override}\big(mc, D_c, \overline{C_c} \rightarrow (C_c)_0, \mathcal{A}'\big) \qquad \mathcal{A}' \Rightarrow AT\,((C_c)_0) \qquad AT\left(\overline{C_c\ xc}\right) \Rightarrow AT\left(\overline{C_c}\right) \end{array}}{M_c \text{ OK in } C_c}$$

5.6.7 $\phi_{\textbf{CLASS}}$

Type checking for the transformation from a well-typed class declaration (C OK) to class C_c should be considered as the checking upon all AST nodes within C and C_c. It contains the following steps:

1. Checks the type of C_c as its super class is present;

2. C_c and its super class/interface should be well-typed;

3. C_c's annotation should be compatible with C's annotation;

4. The typing rules specify that super constructor (of $(C_c)_{super}$) call receives exactly the parameters from constructor (of C_c);

5. As rule (K.2[46]) specifies in the field assignment($this.f = f$), the constructor parameters must match the field declared;

6. All types associated with the constructors' parameters must be present; and

7. In addition, methods, fields, and constructors within the class should be checked as well.

$$\frac{\begin{array}{c} K = C\left(\overline{\text{D g}}, \overline{\text{C f}}\right)\{super\left(\overline{g'}\right); \overline{\text{this.f=f;}}\} \\ \overline{M} \text{ OK in C} \qquad K' = C_c\left(\overline{\text{D}' \; gc}, \overline{C_c \; fc}\right)\{super\left(\overline{gc'}\right); \overline{this.fc = fc}\} \\ \overline{\text{M}_C} \text{ OK in } C_c \qquad AT(C) = \mathcal{A} \qquad AT\left(C_c\right) = \mathcal{A}' \\ \mathcal{A} \Rightarrow \mathcal{A}' \qquad filter\left(fields\left(D\right), \mathcal{A}\right) = \overline{\text{D } g''} \qquad \text{filter}(fields\left(D'\right), \mathcal{A}') = \overline{\text{D}' \; gc''} \\ \mathcal{A} \Rightarrow AT(D) \qquad AT\left(\overline{\text{C f}}\right) \equiv AT\left(\overline{\text{this.f=f}}\right) \equiv AT\left(\overline{\text{C f'}}\right) \\ \mathcal{A}' \Rightarrow AT(D') \qquad AT\left(\overline{C_c \; fc}\right) \equiv AT\left(\overline{\text{this.fc=fc}}\right) \equiv AT\left(\overline{\text{C_c fc'}}\right) \\ \mathcal{A} \Rightarrow \left(AT(\overline{\text{D g}}) \equiv AT(\overline{g'}) \equiv AT(\overline{\text{D } g''})\right) \\ \mathcal{A}' \Rightarrow \left(AT(\overline{\text{D}' \; gc}) \equiv AT(\overline{gc'}) \equiv AT(\overline{\text{D}' \; gc''})\right) \\ AT\left(\overline{\text{C_c fc}}\right) \Rightarrow AT\left(C_c\right) \qquad AT\left(\overline{\text{D_c gc}}\right) \Rightarrow AT\left(D_c\right) \\ AT\left(\overline{C_c \; fc'}\right) \Rightarrow \mathcal{A}' \qquad AT\left(\overline{M_c}\right) \Rightarrow \mathcal{A}' \qquad AT\left(\overline{D' \; gc}\right) \Rightarrow \mathcal{A}' \\ AT\left(\overline{C_c \; fc}\right) \Rightarrow \mathcal{A}' \qquad AT(\overline{gc'}) \Rightarrow \mathcal{A}' \qquad AT(\overline{this.fc = fc}) \Rightarrow \mathcal{A}' \end{array}}{\text{class} C_c \text{extends} D_c \{\overline{C_c \; f'_c}; K_c \; \overline{M_c}\} \text{ OK}}$$

5.6.8 Putting All Pieces Together

In summary, type checking constraint $\phi_{\mathbf{type}}$ is defined as a specific checking based on types of t and tc.

$$\begin{aligned} \phi_{type}(t, tc) &= \bigcap_{tsub}^{TYPE} \phi_{tsub}(t, tc); \\ TYPE &= \{\text{T-V,T-F,T-I,T-N,CAST,METHOD,CLASS}\}; \end{aligned} \tag{5.13}$$

which means the type checking constraint only checks typing errors related to t and tc. Here, the type checking is conducted on both t and tc to ensure that the AST nodes in the legacy and the variant products are well-typed.

5.7 Configurable AST: Feature-effect Constraints

The feature effect constraints are collected based on the constraints and dependencies covered in a feature model. However, this type of constraint has already been covered in the given configuration. That is, a given configuration is "valid" if and only if it satisfies the constraints from the feature model. In feature model analysis and transformation, the problem could be transformed into a boolean satisfaction problem and then it can be resolved by a satisfiable problem solver. As shown in the process displayed in Fig.5.1, we add a validity check to check whether the input configuration is correct, and the invalid configuration will be rejected. Hence, this checking will resolve feature effect constraints.

As for the valid configurations, the constraints are automatically resolved during constraints checking as described in **Sec.**5.5 to **Sec.** 5.6, since these constraints also consider the side-effect from feature annotations.

5.8 Configurable AST: Algorithm

5.8.1 Putting all pieces together

To achieve our goal of generating product variants respect to syntax correctness, behaviour preserving and well-typed, the constraints required to be satisfied is shown in equation (5.14). That is a reengineering action will be approved if and only if it meets the requirements from the constraints.

$$\phi_{\mathbf{full}} = \phi_{\mathbf{syntax}} \wedge \phi_{\mathbf{behaviour}} \wedge \phi_{\mathbf{type}} \wedge \phi_{\mathbf{feature}} \tag{5.14}$$

5.8.2 From Annotated Legacy to Product Line

Algorithm 4: *Annotation2Variants*

Input: annotated AST,nodetofeature, configuration c
Output: $variant_c$

1 **if** *c is invalid* **then**
2   **return** error in configuration c;
3 Create an action list *action_list*;
4 Extract actions from c, store in *actions*;
5 *action_list* ← *actions*;
6 **forall** $act \in actions$ **do**
7   **if** $\phi_{syntax,behaviour,feature}(act)$ *not pass* **then**
8   delte *act* in *action_list* and continue;
9   Add $\Delta(act)$ to *action_list*;
10 **forall** $act \in action_list$ **do**
11   **if** $\phi_{type}(act)$ *not pass* **then**
12   delete *act* in *action_list*;
13 Execute *action_list*;
14 **return** $variant_c$;

To further present the overall constraint ϕ_{full} (**Eq.**(5.14)) as a practical approach, we define the procedure in **Alg.**4.

- **Line 1:2**: Check whether the input configuration is valid;
- **Line 3:5**: The manipulating of an annotated legacy will be transformed to a set of actions, which could be extracted in configuration c. And then store these actions in *actions* and *action_list* respectively;
- **Line 6:9**: The *syntax*, *feature* and *behaviour* constraints are checked in this step. If it is not possible to satisfy these constraints, then it will skip this action; otherwise, corresponding actions are added back to *action_list*. Here, the corresponding actions could be additional actions taken to satisfy these constraints;
- **Line 10:12**: The *type* constraint is checked; if it could not be satisfied, then delete this action; and
- **Line 13:14**: Based on all valid actions, the variant $variant_c$ is created.

In summary, our approach tries to prevent unexpected problems generated from syntax, behaviour, feature and typing. Our approach avoids generating any syntax errors by referencing "valid AST fragment rules", which guides the actions taken upon AST nodes are acceptable iff they follow the rules; otherwise, this action should not be allowed. As for behaviour and typing errors, we consider the following constraints: control flow, data flow, name binding, context-sensitive relations, and type checking. The feature related issue is resolved by feature-effect constraint.

5.9 Case Studies

5.9.1 Experimental Settings

Input Configurations

Generating and testing all valid configurations require extremely large effort as the number of features increases. For example, a feature model with 27 features will create 2^{27} combinations, although the constrains will reduce this, still it will require a lot of effort. Therefore, in this chapter, we adopt a specific pair-wise testing for product line as described in [75] to reduce the test cases required and try to cover each configuration option at least once.

Generate Variants

After all checking are conducted and the *OperationList* is updated correspondingly, our engine will generate the product variant based on *OperationList* as shown in **STEP 4** of Fig.5.1. The Eclipse Java development tools (JDT) provides APIs to rewrite ASTs and generate code fragments based on the given AST.

5.9.2 Subject Systems

As the focus of this work is to generate product variant from an annotated legacy application, we first find suitable benchmarks. The benchmark contains a set of files that describe how programming elements are mapped to features. Without the benchmark, it would not be possible to assess the performance of our approach. Thereby, we use those systems that have been analysed and learned in other works to exclude the bias in creating the benchmark on our own. And the benchmarks are selected based on following criteria:

1. A benchmark system should be well-analysed by other research works;
2. Along with the benchmark system, the feature model, which shows all features and relations between them, should also be available, since, we will check whether a configuration is valid by referencing the feature model.
3. In addition, mapping between features and their implementation should also be available.

By referencing the selection criteria, we carefully select 9 different subject systems that have been developed and well-researched by others, from academic and industrial systems. Four of these systems are early described in Sec. 3.5.

- **Graph Product Line.** Lopez-Herrejon et. al. developed a Graph Product Line to illustrate the problem in product line design [64] with 1350 LOC in Java.
- **HSQLDB.** Hyper SQL Database[5] is a relational database written in Java. It offers a small, fast multithreaded and transactional database engine with in-memory and disk-based tables and supports embedded and server modes.
- **MobileRSS.** It is an open-source project with portable RSS reader for mobile phones on Java ME platform. The core features include: *MIDP1.0*, *MIDP2.0*, *CLDC1.1*, *JSR 75* and *JSR 238*. Other features supported include logging, testing, memory capacity and compatibility features.

[5] HyperSQL: available at:`http://hsqldb.org`.

- **Sudo.** With 1975 LOC, Sudo is designed as a student project containing five features: *States*,*Undo*, *Solver*, *Generator* and *Variable Size*. It also contains the following relations: $Generator \rightarrow Solver$, $Solver \rightarrow Undo$, and $Undo \rightarrow States$.

- **Lampiro[6].** An open-source instant-messaging client with 44584 LOC. Here feature *Compression* without dependency is selected, as others are affected by limited code fragments or cannot be deemed as debugging features. Here, a debugging feature represents a feature that provides an "invokable" service to end-users rather than assisting the workflow. Some other small features have already been tested in other cases.

5.10 Experimental Result

We experimentally investigate the following research questions in order to test our approach:

[**RQ1**]: *Can our approach successfully ensure the syntactic correctness and well-typed in the variants created?*

[**RQ2**]: *Can our approach successfully ensure the feature behaviours are well preserved during reengineering?*

RQ1: Syntactical and Type Checking Results

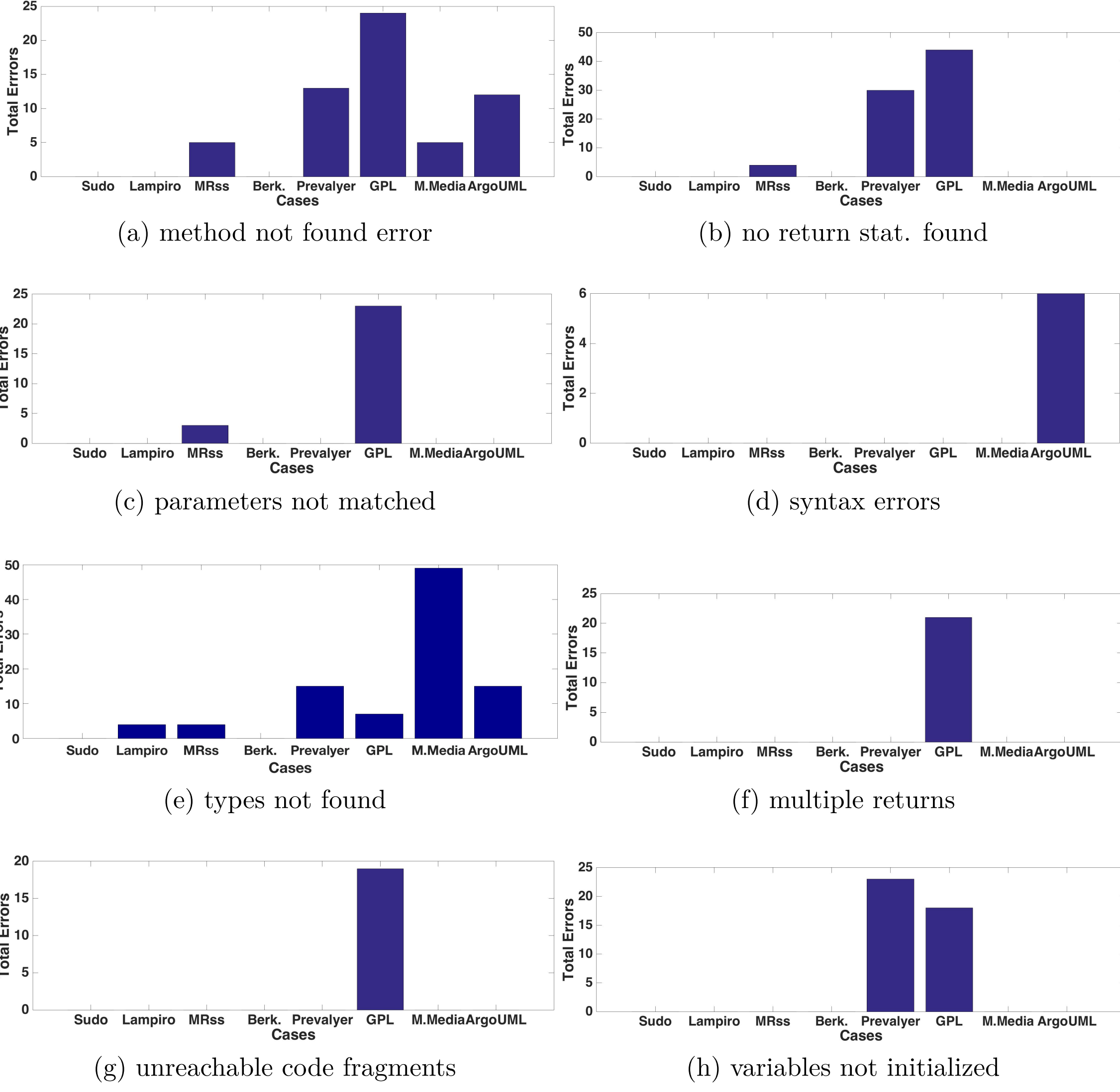
We compare our approach `CAST`, with $\texttt{LJ}^{AR}$ as shown in **Tab.**5.2 with two metrics: (1) **#valid** to show the percentage of valid variants created for configurations tested; and

[6] Lampiro v9.6.0 available at `https://code.google.com/archive/p/lampiro/`

Table 5.2: Error collection and statistics

System	#TestV	#Valid		Errors	
		LJ^{AR}	CAST	LJ^{AR}	CAST
GPL	32	**0%**	75%	**6.1±3.0**	0.4±0.8
BerkeleyDB	14	100%	100%	0	0
HSQLDB	14	60%	100%	0.9±1.0	0
MobileRss	18	78%	100%	1.2±1.7	0
Sudo	6	100%	100%	0	0
Prevalyer	30	**10%**	100%	**2.7±1.6**	0
MobileMedia	12	**8%**	100%	**4.5±3.1**	0
Lampiro	11	90%	90%	0.4±1.2	0.3±0.1
ArgoUML	22	**31%**	73%	**1.5±1.5**	0.4±0.7
Avg./Overall	17	**31%**	88%	10.1±4.8	0.1±0.5

(2) **error** in a format of "mean±std" to show the mean value(mean) and standard deviation(std) of the number of errors.



(a) method not found error

(b) no return stat. found

(c) parameters not matched

(d) syntax errors

(e) types not found

(f) multiple returns

(g) unreachable code fragments

(h) variables not initialized

Figure 5.11: Error types of LJ^{AR} approach

From the performance of $\mathbf{LJ}^{AR}$ and CAST shown in **Tab.**5.2,

1. For most cases, LJ^{AR} can only generate valid product variants for 22% (2/9) of all systems. Whereas CAST could reach 67% (6/9) on average;

2. For the performance of # valid configuration, LJ^{AR} could only reach 31%, whereas CAST reaches 88%;

3. For 44% cases, LJ^{AR}'s performance is less than 50%. Whereas, all CAST's performance is larger than 50%; and

4. The error of LJ^{AR} ranges a lot (10.1±4.8), which means the performance is unstable. However, our approach seems to give a more stable performance (0.1±0.6).

The errors found in LJ^{AR} as shown in Fig.5.11 could be grouped into the following types:

1. Fig.5.11-*(a)*: methods are not implemented, representing two cases:

   (a) methods defined in parent classes are not found in the current class;

   (b) method's invocations are found, whereas its declaration is removed;

2. Fig.5.11-*(b)*: no return statements;

3. Fig.5.11-*(c)*: parameters are not matching. The parameter not match problem is generated in the case that an access of a method m_{access} is preserved, where its declaration m has been removed. Therefore, the method's access will lead to a "parameters are not matching" problem, where m's overriding method m' exists;

4. Fig.5.11-*(d)*: syntax error. Additionally, some syntax errors could be generated when using LJ^{AR}.

5. Fig.5.11-*(e)*: types are not found. Types are used without declaration;

6. Fig.5.11-*(f)*: unreachable code error. The unreachable code error arises when the code fragment appears after "return" statement;

7. Fig.5.11-*(g)*: multiple returns. The "multiple returns" problem arises when a series of "return" statements appear;

8. Fig.5.11-*(h)*: variables/fields/instances are not initialised;

Our approach could resolve these problems by applying additional constraints during transformation. Specifically, these problems could be resolved partially in our approach.

```
public Vertex getStart() {
    if (true)
        return this;
    else
        return null;
}
```


Figure 5.12: No Return Error

```
EdgeIfc addEdge(Vertex start, Vertex end, int weight){
   ...
   if(true)
       return (EdgeIfc)start;
   //GN
       Neighbor e = new Neighbor(end,weight);
       addEdge(start,e);
       if(true)
           return e;
   // GEN
       Edge theEdge = new
   Edge(start,end,weight);
       ...
}
```


Figure 5.13: Unreachable Code Error Example

1. Fig.5.11-*(a)*: as for the methods are not implemented issue:

(a) for the case that methods are defined in the parent classes/interfaces but could not be found in the current class, it could be assessed and prevented by constraint $\phi_{\mathbf{METHOD}}(\Delta(M)$ OK in $C_0')$, in which the rule "override($\Delta(m)$,D,C'$\rightarrow$ C_0',$\mathcal{A}'$)" will do the checking. In detail, it will check the method that is overridden;

(b) for the second case, it is resolved by both **name binding constraint** and ϕ_{T-INVK} constraint for method invocation;

2. Fig.5.11-*(b)*: as for no return problem, it occurs in a variant of GPL as shown in Fig.5.12. The removal of this code snippet will make our algorithm check "this" statement and resolve this to type *Vertex*. Since type *Vertex* is preserved during reengineering, our algorithm is fraud as this removal is safe;

3. Fig.5.11-*(c)*: the parameters are not matched problem is also an issue, which is part of method problem, and it is resolved by the **name binding constraint** in our approach. Therefore, if a method invocation existed in a variant, its method declaration with matching parameter set should also be preserved in the variant by using this rule;

4. Fig.5.11-*(d)*: to prevent syntax errors, our valid AST fragments rules will avoid unsafe actions and generate a valid AST based on the rules;

5. Fig.5.11-*(e)*: type not found problem is handled by the type checking constraint in our approach;

6. Fig.5.11-*(f)*: the essence of "multiple returns" problem is same as the unreachable code error, since it can be considered as all *return* statements after the

first *return* are unreachable;

7. Fig.5.11-*(g)*: the unreachable code error is introduced by inconsistent annotation for the benchmark. For example, in the case that *if* condition expression is annotated, but *then* branch is not annotated, which contains a *return* statement. As a result, removal of a feature could make some code fragments unreachable, which is a false-positive case. As the code snippet shown in Fig.5.13(found in project GPL product line), the removal of feature with pink colour could make the rest of the code in the function unreachable. Fortunately, the inconsistent annotation could be resolved in our approach, since we define a set of valid AST fragment rules for preventing syntax problem and unsafe actions;

8. Fig.5.11-*(h)*: for variables/fields/instances are not initialised issues, it could be checked by the data flow constraint in our approach;

9. In addition, we found another issue "method not found for casting". This problem arises in a format of "((A')a).m()", which means an object in type A is casted to A', and the method $m()$ has been removed in type A'. For example, in the MobileMedia product line, we found an error statement "(PhotoViewScreen)this.getCurrentScreen()", where the method $PhotoViewScreen$ has been removed in class PhotoViewScreen.

In summary, our approach could resolve the following issues that cannot be fully solved by other approaches:

1. The syntax errors introduced by inconsistent annotation: the "inconsistent

annotation" is defined as a use of variable/field v is annotated to a feature f, but its definition is not annotated to f. The operation upon v could be risky since it may generate potential typing errors. This could be solved by *data flow* and *name binding* constraint; and

2. The syntax errors introduced by partial AST node annotation: the partial AST node annotation could also be found in the benchmark. For example, in a `try ... catch ...` statement, if and only if `try` block is annotated to feature f, but not the *catch* part. The actions on `try` block could lead to a potential error that "no try statement found for the `catch`". This could be handled by valid AST rules in our approach;

Unfortunately, there is still an issue that cannot be resolved by applying our approach since we do not enforce the structure to check all requisite components, which might lead to "no return" statement error.

RQ2: Behaviour Preservation Checking Results

Besides the syntax and type checking for transformation from an annotated legacy to product variants as shown in Fig.5.1 and the previous section, more importantly, we also check whether the feature's behaviour is consistent during the transformation by running some test cases. In our study, the checking is achieved by test cases, which are automatically generated by *evosuite* framework[7]. Specifically, the statistics of test cases and pass rates are available in **Tab.**5.3. In **Tab.**5.3, "#Method, #Class, #Package" show the basic information of the subject project, including the total number of methods, the number of classes, and the count of packages.

[7] evosuite:`http://www.evosuite.org`.

Table 5.3: Behaviour preserving test and performance

Cases	#Method	#Class	#Pack.	$\mathbf{LR}^{AR}$		
				#Test(Pass)	**Avg.Test**	**Mean±Std**
GPL	89	14	1	512(512)	100%	1 ± 0
BerkeleyDB	3116	268	16	3276(3013)	85%	0.85 ± 0.26
HSQLDB	4186	292	14	3184(2630)	80%	0.8 ± 0.29
MobileRss	928	96	14	13703(13537)	87%	0.87 ± 0.30
Sudo	121	25	1	597(542)	79%	0.79 ± 0.38
Prevalyer	571	127	21	9856(9593)	84%	0.84 ± 0.33
MobileMedia	274	50	9	2960(2960)	100%	1 ± 0
Lampiro	1482	188	12	378(90)	7%	0.07 ± 0.42
ArgoUML	9740	1665	80	2649(2027)	73%	0.73 ± 0.42
Average	2279	303	19	4124(3878)	77%	0.77 ± 0.29
Cases	**#Method**	**#Class**	**#Pack.**	**CAST**		
				#Test(Pass)	**Avg.Test**	**Mean±Std**
GPL	89	14	1	512(512)	100%	1 ± 0
BerkeleyDB	3116	268	16	3276(3248)	92%	0.92 ± 0.19
HSQLDB	4186	292	14	3184(2958)	90%	0.9 ± 0.24
MobileRss	928	96	14	13703(13645)	95%	0.95 ± 0.07
Sudo	121	25	1	597(560)	88%	0.88 ± 0.30
Prevalyer	571	127	21	9856(9797)	95%	0.95 ± 0.09
MobileMedia	274	50	9	2960(2960)	100%	1 ± 0
Lampiro	1482	188	12	378(350)	87%	0.87 ± 0.31
ArgoUML	9740	1665	80	2649(2481)	85%	0.85 ± 0.21
Average	2279	303	19	4124(4067)	93%	0.93 ± 0.15

Our approach could reach a precision of 93% on average, which means that on average for 93% test cases, the program could output the values expected. This suggests that our approach could successfully preserve features' behaviour during the reengineering procedure. As the result shown, our performance is better than LJ^{AR} approach (77%) for the subject systems explored.

5.11 Discussion

5.11.1 Issues Studied

To further illustrate our research purpose and discuss our findings, we try to fill the gap and provide direction to future work below.

I1: What are the limitations of current research?

The limitations of current work motivate our work. That is, to compose or reengineer features' implementation, the constraints and dependencies arise not only from the feature model but also from the code base. Therefore, a qualified approach should combine the constraints from the feature model and the code base, which has not been well considered in current studies. For example, given an annotated legacy with three features a,b, and c. To create a variant with a configuration $c = a \wedge b \wedge ! \, c$, by merely preserving code for features a and b are not sufficient, as some code fragments from c is also required, even if no relation could be detected from the feature model.

I2: What are the potential limitations of our work?

1. The test cases for configurations are generated based on the all-pair testing strategy, which might not be suitable for all configurations. This means that not all valid configurations are tested and the actual performance might be different from the results shown in **Tab.**5.3; and

2. Our approach tries to create valid AST fragments; however, this may violate the feature annotation. That is, we try to generate syntactically correct, typing safe variant and preserving features needed at the large extent; however, sometimes it will violate the essences that features trying to express. Unfortunately, this

dilemma could not be solved without the domain experts, since only they know what the features are trying to express.

I3: How do the benchmarks influence the performance?

In the experiment, we found inconsistent feature annotation problem in the benchmark. For example, in the annotated legacy, a method m is annotated to feature f; however, m's invocation is not annotated. But, our approach could cope with this inconsistent annotation issue by checking all types of constraints and try to generate valid ASTs based on valid AST rules. Whereas LJ^{AR} could not solve this. However, the feature inconsistent annotation issue could be resolved by our approach.

I4: Behaviour preserving and behaviour tolerance issue.

Behaviour preserving is another critical concern in our approach. Our approach uses test cases to automatically check whether the behaviours are persevered during the transformation. As the testing results partially reflect that the behaviours could be kept during transformation, our approach could be a qualified approach for generating variants from annotated legacies. However, in some cases methods' structures might be changed with certain configuration.

For example, a method $m(a, b, c)$ might be changed to $m'(a, b)$ for c is associated with a feature not covered in the configuration. For this change, the statements, rewrite and read argument a and b, are left in the variant. As the code snippet shown in `Edge` class in *Graph Product Line* product line, the last argument `aweight` is annotated to a feature different from the rest; therefore, removal of this feature could result in the creation of new constructor `Edge` with only two arguments: `Vertex the_start` and `Vertex the_end`. We can ensure there is no interaction with argu-

ment c; otherwise c should be kept in the variant. Whereas, the potential risk could be we try to keep a feature's implementation consistent with respect to the programming logic, which might change the original intention of the feature.

l5: Can our approach resolve the limitations in other works?

As presented and discussed in the results,

1. the issues with the syntax errors introduced by inconsistent annotation and errors introduced by partial AST node annotation are resolved by the syntactical correctness module;

2. as demonstrated by the test cases, our *behaviour preserving* module could preserve the feature behaviours during the reengineering; and

3. the type checking module ensures that all types used in the variant applications are well-resolved;

In summary, comparing to related approach, our strategy could perform better in terms of reducing syntax errors[47], and we also contribute the behaviour preserving and typing safety, which have not been discussed in other research works[63, 8, 96, 71, 50, 46, 47, 57].

5.11.2 Threats to Validity

The validity issues may exist in following aspects: (1) limited case studies: even though the experiments are conducted based on nine systems with size ranging from 1K to 120KLOC, the performance of our approach still could not be extended to all applications; (2) single language: in this work, our case studies are all in Java. The performance bottleneck and scalability have not been tested in other languages; (3)

the configurations are created using all-pairs testing and invalid configurations are removed. Since not all configurations are tested, this might affect the performance of our approach; and (4) the tools could also affect the performance. CIDE as a prototype tool of LJ^{AR} might not strictly follow the definitions and rules in LJ^{AR}, which might impact LJ^{AR}'s performance.

5.12 Chapter Summary

The reengineering of legacy systems requires features' implementation and behaviours be well preserved. To ensure this, in this work, we propose a feature persistent approach for migrating an annotated legacy into a product line. As demonstrated in the experimental results, our approach could preserve feature's implementation and behaviour during the transformation from an annotated legacy.

Chapter 6

Conclusion

Software product lines are difficult to build compared to developing a single system, considering that several issues should be resolved, including domain requirements, feature model, feature interaction and variability. Building a software product line from an existing system is a helpful and practical step in terms of constructing the software product line by reusing software artefacts. In this way, the legacy applications could be reused as a product line, which will extend the life cycle of the application, supporting the extension of the system and contributing to software ecosystem.

This thesis has presented the migrating procedure from the code base of a legacy application to a software product line. This work is intended to bridge the gap between a normal system and a software product line. If successful, it can improve the construction of a product line and make it easier to develop a software product line rather than building every artifact in a product line. And it improves the current research in software product line engineering in following ways:

1. It can provide a guide on how to build a software product line by reusing

existing resources, and it can also be extended to legacy understanding. That is, without the help of documentation, domain experts and developers can start the learning process of a new system by mainly referencing the source code;

2. How to conduct the type checking on a software product line, and a legacy system, and how to reengineer a legacy system into a software product line? This part will make the product line created are well-typed. Specifically, in Chapter 3, when we build the feature model, our work ensures that each feature is well-typed; and in Chapter 5, when the features are reengineered to product variants, we guarantee that each product variant is well-typed;

3. To map features with their implementations, in Chapter 4, we proposed a novel approach based on conditional probability. With this approach, the performance has been strongly improved, since it can extract and compute the relation between programming element at a fine-granularity;

4. We distinguish the feature model and software architecture, we argue that the approaches in the software architecture domain are not suitable for building feature models; and

5. Our tools provide a complete process to build product variants from the legacy source and assist developers in understanding the functions, features, relations and other underlying constraints to build the product line.

Section 6.1 reviews the most important contributions of this thesis and discusses how these contributions solve the problems in practice. Section 6.2 gives some directions and perspectives for future work.

6.1 Summary of Contribution

In this thesis, we try to compose a software product line by reengineering a legacy application. We explore the path from a legacy application to a software product line, and the solution on generating well-typed product line application and customised product variants. Specifically, we make the following contributions in building a product line from a legacy application:

1. We build a type checking system to ensure the product variants created are well-typed;
2. We develop a variability-aware module system to build the feature model from the code base;
3. Based on the feature model built from previous step, we develop an approach using conditional probability to locate features in the code base;
4. With features and their implementations extracted. We further create a tool to extract all valid configurations for the product line;
5. With the help of the configuration tool, we propose an approach to reengineering the legacy system to product variants; and
6. To support all these processes, we develop several prototype tools.

6.2 Future Work

We have provided a series of approaches for transformation a legacy application into a software product line. Our approach provides a general procedure for building the

product line from a legacy application.

Transferring our study and experience to other works should be a promising direction. Nowadays, software products are often provided as services in the cloud; therefore, building services for cloud could be an interesting direction to consider. Composing modules for cloud applications with modularity concern by systematically reusing existing legacy could save the development effort in terms of building cloud applications.

Also, feature model in a software product line is essential, since it represents all the features in the system and how they are organised in the software product line. However, it is still unknown whether there exists a better representation comparing to the feature model. Some research efforts are required to check attributes that are essential but cannot be captured by the feature model.

Furthermore, working on presenting features and underlying relations should be an open challenge for developers. Currently, features are presented and organised using the feature model. Although in this thesis we build the mapping between feature and code fragments, there is still a long way to go for presenting features and visualising them. In addition, more effort are needed to visualise the features. In our approach and other approaches, features are annotated directly in the code. To view a features' implementation, normally a developer has to switch between different files. However, a better practice could be building a tool to index all code fragments grouped by features.

Finally, there are many related challenges from building the software product line and feature annotation, especially for software evolutions. For example, adding additional features to the existing software product, removing features from the

product line or updating features may introduce potential defects to the feature itself, including whether a feature's consistent is preserved. In the future, we intend to build tools and models to develop an integrated system for building adaptable software product lines, in which developers are allowed to make changes to the feature model, and the system will automatically transform the code base.

References

[1] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire. Reverse Engineering Architectural Feature Models. In *5th European Conference of Software Architecture (ECSA)*, volume 6983, pages 220–235, 2011.

[2] B. Adams, W. De Meuter, H. Tromp, and A. Hassan. Can we refactor conditional compilation into aspects? In *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD '09, pages 243–254, 2009.

[3] R. Al-Msie'Deen. Mining feature models from the object-oriented source code of a collection of software product variants, 2013.

[4] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 201–210, 2006.

[5] L. Andersen. *Program analysis and specialization for the C programming language*. Thesis, 1994.

[6] P. Andritsos and V. Tzerpos. Information-theoretic software clustering. *IEEE Transactions on Software Engineering*, 31(2):150–165, 2005.

[7] G. Antoniol and Y. G. Gueheneuc. Feature identification: An epidemiological metaphor. *IEEE Transactions on Software Engineering*, 32(9):627–641, Sept 2006.

[8] S. Apel, C. Kästner, and D. Batory. Program refactoring using functional aspects. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08, pages 161–170, 2008.

[9] D. Batory. Feature models, grammars, and propositional formulas, 2005.

[10] J. Bayer, J. Girard, M. Wurther, J. DeBaud, and M. Apel. Transitioning legacy assets to a product line architecture. *SIGSOFT Softw. Eng. Notes*, 24(6):446–463, 1999.

[11] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 7:1–7:8, 2013.

[12] J. Bergey, L. Brien, and D. Smith. Mining existing assets for software product lines, 2000.

[13] S. Beydeda, M. Book, and V. Gruhn. *Model-driven software development*, volume 15. Springer, 2005.

[14] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding, 1993.

[15] S. Blazy, A. Maroneze, and D. Pichardie. Verified validation of program slicing, 2015.

[16] J. Bohnet, S. Voigt, and J. Doellner. Locating and understanding features of complex software systems by synchronizing time-, collaboration- and code-focused views on execution traces. In *2008 16th IEEE International Conference on Program Comprehension*, pages 268–271, June 2008.

[17] M. Bruntink, A. van Deursen, M. D'Hondt, and T. Tourwé. Simple crosscutting concerns are not so simple: Analysing variability in large-scale idioms-based implementations. In *Proceedings of the 6th International Conference on Aspect-oriented Software Development*, AOSD '07, pages 199–211, 2007.

[18] S. Buhne, K. Lauenroth, and K. Pohl. Modelling requirements variability across product lines. In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 41–50, Aug 2005.

[19] L. Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 266–277, 1997.

[20] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*, pages 241–247, 2000.

[21] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13–17, 1990.

[22] A. Classen, P. Heymans, P. Y. Schobbens, A. Legay, and J. F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 335–344, 2010.

[23] B. Cleary, C. Exton, J. Buckley, and M. English. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empirical Software Engineering*, pages 93–130, 2009.

[24] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello. Investigating the use of lexical information for software system clustering. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 35–44, 2011.

[25] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, 35(5):684–702, 2009.

[26] M. V. Couto, M. T. Valente, and E. Figueiredo. Extracting software product lines: A case study using conditional compilation. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 191–200.

[27] K. Czarnecki and U. Eisenecker. *Generative programming : methods, tools, and applications*. Boston Mass. : Addison Wesley., 2000.

[28] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[29] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints, 2006.

[30] J. M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Clelang-Huang, and P. Heymans. Feature model extraction from large collections of informal product descriptions, 2013.

[31] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[32] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

[33] M. Eaddy, A. V. Aho, G. Antoniol, and Y. G. Gueheneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 53–62.

[34] T. Eisenbarth, R. Koschke, and D. Simon. Derivation of feature component maps by means of concept analysis. In *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, pages 176–179, 2001.

[35] S. Erdweg, O. Bračevac, E. Kuci, M. Krebs, and M. Mezini. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 880–897, 2015.

[36] M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.*, 21(1):6:1–6:27, 2011.

[37] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[38] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor, and F. Dantas. Evolving software product lines with aspects: An empirical study on design stability. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 261–270, 2008.

[39] R. Filman, T. Elrad, S. Clarke, and M. Akşit. *Aspect-oriented Software Development*. Addison-Wesley Professional, 2004.

[40] W. B. Frakes and Kyo Kang. Software reuse research: status and future. *IEEE Transactions on Software Engineering*, 31(7):529–536, 2005.

[41] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou. Variability in software systems-a systematic literature review. 2013.

[42] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and C. Yuanfang. Enhancing architectural recovery using concerns. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 552–555, 2011.

[43] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *2009 IEEE 31st International Conference on Software Engineering*, pages 232–242, 2009.

[44] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[45] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, 1990.

[46] C. Kastner and S. Apel. Type-checking software product lines - a formal approach. In *Proceedings of the 2008 23rd IEEE/ACM International Conference*

on Automated Software Engineering, ASE '08, pages 258–267, Washington, DC, USA, 2008. IEEE Computer Society.

[47] C. Kästner, S. Apel, and M. Kuhlemann. A model of refactoring physically and virtually separated features. In *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, GPCE '09, pages 157–166, 2009.

[48] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.*, 21(3):14:1–14:39, 2012.

[49] C. Kastner, A. Dreiling, and K. Ostermann. Variability mining: Consistent semi-automatic detection of product-line features. *IEEE Trans. Softw. Eng.*, 40(1):67–82, January 2014.

[50] C. Kästner, A. Dreiling, and K. Ostermann. Variability mining: Consistent semi-automatic detection of product-line features. *IEEE Transactions on Software Engineering*, 40(1):67–82, 2014.

[51] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 805–824, 2011.

[52] C. Kästner, K. Ostermann, and S. Erdweg. A variability-aware module system. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 773–792, 2012.

[53] R. Khatchadourian and H. Masuhara. Automated refactoring of legacy java software to default methods. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 82–93, 2017.

[54] J. Kim, D. Batory, D. Dig, and M. Azanza. Improving refactoring speed by 10x. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1145–1156, 2016.

[55] K. Kobayashi, M. Kamimura, K. Kato, K. Yano, and A. Matsuo. Feature-gathering dependency-based software clustering using dedication and modularity. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 462–471. IEEE, 2012.

[56] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. A case study in refactoring a legacy component for reuse in a product line. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 369–378, Sept 2005.

[57] M. Kuhlemann, D. Batory, and S. Apel. *Refactoring Feature Modules*, pages 106–115. Springer Berlin Heidelberg, 2009.

[58] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 46–57, 1992.

[59] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural change in open-source software systems. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 235–245. IEEE Press, 2015.

[60] K. Lee, K. C. Kang, and J. Lee. Concepts and guidelines of feature modeling for product line software engineering. In *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools*, pages 62–77, 2002.

[61] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer. Morpheus: Variability-aware refactoring in the wild. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 380–391, 2015.

[62] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 81–91, 2013.

[63] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 112–121, 2006.

[64] R. E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, GCSE '01, pages 10–24, 2001.

[65] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, and R. Kroeger. Comparing software architecture recovery techniques using accurate dependencies, 2015.

[66] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 50–59, 1999.

[67] O. Maqbool and H. A. Babri. The weighted combined algorithm: a linkage algorithm for software clustering. In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, pages 15–24, 2004.

[68] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 214–223, 2004.

[69] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage, 2011.

[70] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[71] M. Mortensen, S. Ghosh, and J. Bieman. Aspect-oriented refactoring of legacy applications: An evaluation. *IEEE Transactions on Software Engineering*, 38(1):118–140, 2012.

[72] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Where do configuration constraints stem from? an extraction approach and an empirical study. *IEEE Transactions on Software Engineering*, 41(8):820–841, Aug 2015.

[73] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 858–868, 2015.

[74] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, Berlin ; New York, 1999. Includes bibliographical references and index.

[75] S. Oster, F. Markert, and P. Ritter. *Automated Incremental Pairwise Testing of Software Product Lines*, pages 196–210. Springer Berlin Heidelberg, 2010.

[76] J. L. Overbey, F. Behrang, and M. Hafiz. A foundation for refactoring c with macros. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 75–85, 2014.

[77] M. Petrenko and V. Rajlich. Variable granularity for improving precision of impact analysis, 2009.

[78] K. Pohl, G. Böckle, and F. Linden. Software product line engineering foundations, principles, and techniques, 2005.

[79] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *15th IEEE International Conference on Program Comprehension*, pages 37–48, 2007.

[80] K. Praditwong, M. Harman, and X. Yao. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 37(2):264–282, 2011.

[81] M. Glinz R. Stoiber. Modeling and managing tacit product line requirements knowledge. In *Managing Requirements Knowledge (MARK), 2009 Second International Workshop on*, pages 60–64, Sept 2009.

[82] A. Reynolds, M. Fiuczynski, and R. Grimm. On the feasibility of an aosd approach to linux kernel extensions. In *Proceedings of the 2008 AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, ACP4IS '08, pages 8:1–8:7, 2008.

[83] M. P. Robillard. Topology analysis of software dependencies. *ACM Trans. Softw. Eng. Methodol.*, 17(4):1–36, 2008.

[84] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1), 2007.

[85] M. R. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 406–416, May 2002.

[86] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *14th IEEE International Conference on Program Comprehension*, pages 84–88, 2006.

[87] K. Sartipi. Alborz: a query-based tool for software architecture recovery. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 115–116, 2001.

[88] K. Sartipi. Software architecture recovery based on pattern matching. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 293–296, 2003.

[89] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird. Recommending random walks. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 15–24, 2007.

[90] T. Savage, M. Revelle, and D. Poshyvanyk. Flat^3: feature location and textual tracing tool. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 255–258, 2010.

[91] Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proceedings of the 14th International Conference on Software Product Lines*, pages 77–91, 2010.

[92] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 461–470, 2011.

[93] J. Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):1–41, 2012.

[94] D. Smith, L. Brien, and J. Bergey. Mining components for a software architecture and a product line: the options analysis for reengineering (oar) method, 2001.

[95] M. Steyvers and T. Griffiths. Probabilistic topic models. *Handbook of latent semantic analysis*, 427(7):424–440, 2007.

[96] Y. Tang and H. Leung. Sticprob: A novel feature mining approach using conditional probability. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 45–55, 2017.

[97] R. Thiessen and O. Lhoták. Context transformations for pointer analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 263–277, 2017.

[98] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, 2014.

[99] M. Trifu. Using dataflow information for concern identification in object-oriented software systems. In *2008 12th European Conference on Software Maintenance and Reengineering*, pages 193–202, 2008.

[100] V. Tzerpo. The orphan adoption problem in architecture maintenance. In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, WCRE '97, pages 76–82, 1997.

[101] V. Tzerpos and R. C. Holt. Acdc: An algorithm for comprehension-driven clustering, 2000.

[102] M. T. Valente, V. Borges, and L. Passos. A semi-automatic approach for extracting software product lines. *IEEE Transactions on Software Engineering*, 38(4):737–754, 2012.

[103] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. Variational data structures: Exploring tradeoffs in computing with variability. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 213–226, 2014.

[104] Z. Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pages 194–203, 2004.

[105] N. Wilde and M. C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[106] W. E. Wong, S. S. Gokhale, J. R. Horgan, and K. S. Trivedi. Locating program features using execution slices. In *Application-Specific Systems and Software Engineering and Technology, 1999. ASSET '99. Proceedings. 1999 IEEE Symposium on*, pages 194–203, 1999.

[107] G. Xu, A. Rountev, and M. Sridharan. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 98–122, 2009.

[108] Y. Xue. Reengineering legacy software products into software product line based on automatic variability analysis. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1114–1117, 2011.

[109] Y. Xue, Z. Xing, and S. Jarzabek. Understanding feature evolution in a family of product variants. In *2010 17th Working Conference on Reverse Engineering*, pages 109–118, 2010.

[110] Y. Yang, X. Peng, and W. Zhao. Domain feature model recovery from multiple applications using data access semantics and formal concept analysis. In *2009 16th Working Conference on Reverse Engineering*, pages 215–224. IEEE, 2009.

[111] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. S. do Prado Leite. Reverse engineering goal models from legacy code. In *13th*

IEEE International Conference on Requirements Engineering, pages 363–372, 2005.

[112] W. Zhihua and V. Tzerpos. An optimal algorithm for mojo distance. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 227–235, 2003.