FEATURE-ORIENTED
COMPONENT-BASED
DEVELOPMENT OF WHOLE
SOFTWARE PRODUCT FAMILIES
USING ENUMERATIVE
VARIABILITY

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2019

By
Chen Qian
School of Computer Science
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>10</td>
</tr>
<tr>
<td>Declaration</td>
<td>11</td>
</tr>
<tr>
<td>Copyright</td>
<td>12</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>13</td>
</tr>
<tr>
<td>List of Publications</td>
<td>14</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>17</td>
</tr>
<tr>
<td>1.1 Research Problem</td>
<td>17</td>
</tr>
<tr>
<td>1.2 Research Question</td>
<td>21</td>
</tr>
<tr>
<td>1.3 Research Aim and Objectives</td>
<td>22</td>
</tr>
<tr>
<td>1.4 Research Contributions</td>
<td>23</td>
</tr>
<tr>
<td>1.5 Research Methodology</td>
<td>24</td>
</tr>
<tr>
<td>1.6 Thesis Outline</td>
<td>27</td>
</tr>
<tr>
<td><strong>2 Product Family Engineering Vs. Product Line Engineering</strong></td>
<td>29</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>29</td>
</tr>
<tr>
<td>2.2 Framework</td>
<td>30</td>
</tr>
<tr>
<td>2.2.1 Problem Space and Solution Space</td>
<td>30</td>
</tr>
<tr>
<td>2.2.2 PLE Framework</td>
<td>32</td>
</tr>
<tr>
<td>2.2.3 PFE Framework</td>
<td>35</td>
</tr>
<tr>
<td>2.3 Variability</td>
<td>38</td>
</tr>
<tr>
<td>2.3.1 Enumerative Variability</td>
<td>38</td>
</tr>
<tr>
<td>2.3.2 Parametric Variability</td>
<td>39</td>
</tr>
<tr>
<td>2.4 Related Paradigms and Approaches</td>
<td>43</td>
</tr>
<tr>
<td>2.4.1 Existing PLE Paradigms</td>
<td>43</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Existing PLE Approaches</td>
</tr>
<tr>
<td>2.5</td>
<td>Summary</td>
</tr>
<tr>
<td>3</td>
<td>X-MAN and FX-MAN Component Model</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>3.2</td>
<td>Component</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Category 1: Component based on Object</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Category 2: Component based on Architectural Unit</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Category 3: Component based on Encapsulated Component</td>
</tr>
<tr>
<td>3.3</td>
<td>Component Model</td>
</tr>
<tr>
<td>3.3.1</td>
<td>X-MAN Component Model</td>
</tr>
<tr>
<td>3.3.2</td>
<td>FX-MAN Component Model</td>
</tr>
<tr>
<td>3.4</td>
<td>Summary</td>
</tr>
<tr>
<td>4</td>
<td>A Feature-oriented Component-based Approach to Constructing Whole Software Product Families</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>4.2</td>
<td>Family Construction</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Constructing Feature Model</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Modelling and Implementing Features as Components</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Modelling and Implementing Variation Points as Variation Generators</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Applying Family Connectors and Adaptors to Generate Product Variants and Product Families</td>
</tr>
<tr>
<td>4.3</td>
<td>Crosscutting Concern</td>
</tr>
<tr>
<td>4.4</td>
<td>Feature Interaction</td>
</tr>
<tr>
<td>4.5</td>
<td>Summary</td>
</tr>
<tr>
<td>5</td>
<td>Testing Software Product Families Constructed by Enumerative Variability</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>5.2</td>
<td>Domain Unit Testing</td>
</tr>
<tr>
<td>5.3</td>
<td>Domain Integration Testing</td>
</tr>
<tr>
<td>5.4</td>
<td>Domain System Testing</td>
</tr>
<tr>
<td>5.5</td>
<td>Summary</td>
</tr>
</tbody>
</table>
6 Tool Support

6.1 Introduction ............................................. 101
6.2 Web-based Application ................................. 102
6.3 Technologies ............................................. 104
   6.3.1 HTML5 ............................................. 104
   6.3.2 CSS3 .............................................. 105
   6.3.3 JavaScript ......................................... 106
   6.3.4 jQuery ............................................. 108
   6.3.5 jTopo ............................................. 109
6.4 Feature Modelling Tool ................................. 110
6.5 Component Modelling Tool ............................. 111
   6.5.1 Atomic Component Modelling Tool ............... 111
   6.5.2 Composite Component Modelling Tool ........... 112
6.6 Repository ............................................. 115
6.7 Family Modelling Tool .................................. 117
6.8 Product Explorer ....................................... 120
6.9 Testing Tool ............................................ 122
6.10 Other Functions ....................................... 122
6.11 Summary .............................................. 124

7 Case Study: External Car Light Family ................. 125

7.1 Introduction ............................................ 125
7.2 Requirement and Feature Model ....................... 126
7.3 Roadmap of ECL Family Construction ................. 130
   7.3.1 Step 1: Implement Leaves Features ............. 130
   7.3.2 Step 2: Apply Variation Generators ............. 132
   7.3.3 Step 3: Apply Family Composition Connectors .... 133
   7.3.4 Step 4: Address Feature Interactions ............ 135
   7.3.5 Step 5: Add Data Channels and Constraints ...... 137
7.4 Testing of ECL Family and Products .................. 139
   7.4.1 Step 1: Domain Unit Testing of ECL Family ...... 139
   7.4.2 Step 2: Domain Integration Testing of ECL Family 140
   7.4.3 Step 3: Domain System Testing of ECL Family ... 141
   7.4.4 Step 4: Application Testing of ECL products .... 142
7.5 Summary .............................................. 143
8 Evaluation

8.1 Introduction .................................................. 145
8.2 Evaluation by Quality Attributes ................................. 146
  8.2.1 Testability ................................................. 146
  8.2.2 Scalability ............................................... 149
  8.2.3 Maintainability ........................................... 150
  8.2.4 Evolvability .............................................. 151
  8.2.5 Reusability ............................................... 154
8.3 Evaluation by Framework ....................................... 155
  8.3.1 Family Evaluation Framework ............................. 155
  8.3.2 Evaluation Result ...................................... 160
8.4 Summary .................................................... 161

9 Discussion and Conclusion ..................................... 162
  9.1 Discussion .................................................. 162
  9.2 Achievements ............................................... 163
  9.3 Future Work ................................................. 164

Bibliography .................................................... 166

A Screenshots of Our IDE ....................................... 198

Word Count: 50167
List of Tables

1.1 Comparison of Windows 10 editions . . . . . . . . . . . . . . . . . . 19
2.1 Domain artefacts in PLE approaches. . . . . . . . . . . . . . . . . . 43
5.1 Vending machine: Leaf features and their components. . . . . . . . 93
7.1 Leaf features and related components. . . . . . . . . . . . . . . . . 130
8.1 Architecture dimension: aspects and levels. . . . . . . . . . . . . . 156
List of Figures

1.1 Windows 10 family. .................................................. 18
1.2 Research methodology (adapt from [DHTA12]). ................. 25
1.3 Thesis outline (Chapter 2-4). ..................................... 27
1.4 Thesis outline (Chapter 5-9). ..................................... 28

2.1 The PLE framework. .................................................. 32
2.2 The PFE framework. .................................................. 36
2.3 Vending machines: Feature model. .................................. 39
2.4 Negative (parametric) variability in solution space. ............. 40
2.5 Positive (parametric) variability in solution space. ............. 42
2.6 Basic workflow of PLE. .............................................. 51
2.7 Workflow of our approach. ......................................... 52

3.1 Composing generic software components. ......................... 56
3.2 The three main types of components. ............................. 57
3.3 Component life cycle. ............................................... 62
3.4 X-MAN: Component models. ....................................... 63
3.5 X-MAN: Composition connectors. .................................. 64
3.6 X-MAN: Adaptors. .................................................... 64
3.7 X-MAN: Aggregator. ................................................. 65
3.8 X-MAN: Composing components. .................................. 66
3.9 X-MAN: Data channels. ............................................. 67
3.10 FX-MAN: Component model. ...................................... 68
3.11 FX-MAN: Variation generators. .................................... 68
3.12 FX-MAN: Family Sequencer. ...................................... 69
3.13 FX-MAN: Family Selector. ........................................ 70
3.14 FX-MAN: Family adaptors. ....................................... 71

4.1 Redundancy. ......................................................... 74
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Anomalies</td>
<td>75</td>
</tr>
<tr>
<td>4.3</td>
<td>Inconsistency</td>
<td>76</td>
</tr>
<tr>
<td>4.4</td>
<td>Vending machines: Extended feature model</td>
<td>76</td>
</tr>
<tr>
<td>4.5</td>
<td>Vending machines: Component Coffee for feature Coffee</td>
<td>77</td>
</tr>
<tr>
<td>4.6</td>
<td>Vending machines: Variation generators mapped onto variation points.</td>
<td>78</td>
</tr>
<tr>
<td>4.7</td>
<td>Vending machines: Product family architecture</td>
<td>79</td>
</tr>
<tr>
<td>4.8</td>
<td>Login system: Product family</td>
<td>81</td>
</tr>
<tr>
<td>4.9</td>
<td>Addressing homogeneous crosscutting concern</td>
<td>82</td>
</tr>
<tr>
<td>4.10</td>
<td>Vending machines: Product family architecture (with crosscutting feature)</td>
<td>83</td>
</tr>
<tr>
<td>4.11</td>
<td>Two common strategies of addressing feature interactions.</td>
<td>84</td>
</tr>
<tr>
<td>4.12</td>
<td>Hello world: Product family</td>
<td>84</td>
</tr>
<tr>
<td>4.13</td>
<td>Hello world: “Hello World”</td>
<td>85</td>
</tr>
<tr>
<td>4.14</td>
<td>Hello world: “Hi World”</td>
<td>86</td>
</tr>
<tr>
<td>4.15</td>
<td>Hello world: “Bonjour Monde”</td>
<td>86</td>
</tr>
<tr>
<td>4.16</td>
<td>Vending machines: Product family architecture (with crosscutting concern and feature interaction)</td>
<td>87</td>
</tr>
<tr>
<td>5.1</td>
<td>The W-model for software product line testing.</td>
<td>89</td>
</tr>
<tr>
<td>5.2</td>
<td>The W-model for X-MAN CBD process [LTT11]</td>
<td>91</td>
</tr>
<tr>
<td>5.3</td>
<td>Vending machine: Family statechart</td>
<td>97</td>
</tr>
<tr>
<td>5.4</td>
<td>Vending machine: Variability-aware testing case (pseudo code)</td>
<td>98</td>
</tr>
<tr>
<td>5.5</td>
<td>Vending machine: FODA-like family statechart</td>
<td>99</td>
</tr>
<tr>
<td>6.1</td>
<td>Tools in our IDE</td>
<td>102</td>
</tr>
<tr>
<td>6.2</td>
<td>HTML example</td>
<td>104</td>
</tr>
<tr>
<td>6.3</td>
<td>CSS example</td>
<td>105</td>
</tr>
<tr>
<td>6.4</td>
<td>JS example</td>
<td>107</td>
</tr>
<tr>
<td>6.5</td>
<td>jQuery example</td>
<td>109</td>
</tr>
<tr>
<td>6.6</td>
<td>jTopo layers</td>
<td>109</td>
</tr>
<tr>
<td>6.7</td>
<td>Feature model canvas</td>
<td>110</td>
</tr>
<tr>
<td>6.8</td>
<td>Cardinality for ‘OR’ variation point</td>
<td>111</td>
</tr>
<tr>
<td>6.9</td>
<td>Variant Explorer</td>
<td>112</td>
</tr>
<tr>
<td>6.10</td>
<td>Atomic component modelling tool</td>
<td>113</td>
</tr>
<tr>
<td>6.11</td>
<td>Testing result in Chrome DevTools</td>
<td>114</td>
</tr>
<tr>
<td>6.12</td>
<td>Composite component modelling tool</td>
<td>115</td>
</tr>
</tbody>
</table>
Abstract

FEATURE-ORIENTED COMPONENT-BASED DEVELOPMENT OF WHOLE SOFTWARE PRODUCT FAMILIES USING ENUMERATIVE VARIABILITY
Chen Qian
A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy, 2019

A software product family is a cluster of related software systems that are used for similar purposes. In order to construct a software product family, product line engineering (PLE) implements the domain artefacts piece by piece, and builds an ‘assembly line’ that is capable to assemble the artefacts for a product based on its configuration. As a result, there is no executable product to be tested before the final generation. Therefore, a product line with non-trivial size is impossible to be tested due to the everlasting problem of combinatorial explosion of variants. Our approach offers a new possibility to construct a product family by constructing a family architecture that captures and realises all the commonalities and variabilities in a feature-oriented component-based manner. As a proof of concept, we implement our approach by developing a web-based tool, which can visualise the process of family construction and automatically generate any number of family members afterwards. Moreover, in this thesis, we will present the major advantage our approach can bring, which is the support of family-based testing. Finally, we evaluate our work by comparison with existing PLE approaches to show some extra potential advantages of our approach, such as scalability, maintainability and evolvability.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University’s policy on presentation of Theses.
Acknowledgements

I would like to express my deep gratitude to my supervisor, Dr. Kung-Kiu Lau, for his patient guidance, enthusiastic encouragement and useful critiques of this research work. My grateful thanks are also extended to all the members of the Component-based Software Development Research Group who helped me for their valuable technical support.

Finally, I would also like to thank my parents for their support and encouragement throughout my study.
List of Publications


Abstract: Constructing a product family requires the formulation in problem space of a domain model (including a variability model) and its implementation in solution space. Current software product line engineering tools mostly aim to build an ‘assembly line’ for deriving one product at a time by assembling domain artefacts according to the variability model. Therefore, those tools support enumerative variability in problem space, but parametric variability in solution space. In this paper, we present a tool to model and implement enumerative variability in both spaces, and hence construct a whole product family in one go.


Abstract: Feature-Oriented Software Development (FOSD) is widely used in software product line engineering. FOSD constructs product families by incremental feature implementations. In this paper, we introduce a feature-oriented component-based approach, which implements features as an encapsulated components for further family modelling. A case study of elevator systems is also presented to describe the use of our approach.

Abstract: In software product line engineering, in the problem space, variability in a product family is specified in an enumerative manner (by a feature model), i.e., all valid variants are enumerated. However, in the solution space, current approaches use parametric variability (variability parameterised on features occurring in a single product variant) instead. In this paper, we take a closer look at enumerative variability, show how it can also be used in the solution space, and briefly discuss why it may be advantageous to do so.


Abstract: In software product line engineering, the construction of an ADL architecture for a product family is still an outstanding engineering challenge. An ADL architecture for a product family would define the architectures for all the products in the family, allowing engineers to reason at a higher level of abstraction. In this paper, we outline a component model that can be used to define architectures for product families, by incorporating explicit variation points.


Abstract: In system design, the distinction between a logical architecture at design level and the corresponding physical distributed architecture at implementation level is recognised as good practice. In this paper, we show how we can define logical architectures in which variation points can be defined explicitly. Such architectures define families of systems, and should therefore be useful for defining such families in IoT.

Abstract: Current software product line engineering tools mainly focus on variability in the problem space, and create product families by linking variability models to artefacts in the solution space. In this paper, we present a tool that can be used to define software architectures with explicit variation points, and hence product families, directly in the solution space.
Chapter 1

Introduction

“When the head rope of a net is pulled up, all the meshes open.”

— Chinese Proverb

1.1 Research Problem

Since 1990s, computer knowledge becomes an essential part of modern society. Digitalisation of everything makes a huge impact on human life. As a result, the phenomenal growth of software requirement has been dramatic and spectacular. The productivity and reusability of conventional object-oriented programming (OOP) are no longer able to keep pace with desired software development capability. Some researchers believe that the contradiction will become the fuse of second software crisis [Fit12]. In order to address the problem, several initiatives have been proposed, including component-based development (CBD) [LO02] and product line engineering (PLE) [NCB+07].

The prime asset in PLE is software product families. A software product family is a group of related systems, which usually share a common set of distinguishable characteristics that satisfy the specific needs of a domain. Such a characteristic is called a feature [CE00]. Figure 1.1 shows an example. According to Microsoft, Windows 10 is a product family, which comprises many editions, e.g., Windows 10 Home, Windows 10 Pro, Windows 10 Education, Windows 10 Enterprise and so on1. All these editions are similar because they have the same set of core features, but also vary from each

1https://www.microsoft.com/en-us/windows/compare
other, due to each edition has its own additional features. Table 1.1 compares the difference between Home, Pro, Enterprise and Education editions in several features. The Pro edition contains 4 features, while the Enterprise edition has all. The core features, such as Windows Update and Device Encryption, are also known as the mandatory features, whereas the rest are variable features that specify the variability in the Windows 10 family.

PLE traditionally proceeds in two phases: domain engineering and application engineering [PBvDL05]. It tackles the problem of constructing a product family by using an ‘assembly line’ (product line). In the domain engineering phase, PLE (i) constructs variability and behaviour models and specifications, and (ii) from these, identifies and implements domain artefacts, e.g., a code base [KTA08], which can be used to assemble individual products. In the application engineering phase, PLE (i) creates product variants by configurations, and (ii) accordingly assembles one product at a time using the domain artefacts [BBM05].

Unfortunately, there are many challenges during the life cycle of PLE. The early work on PLE, such as FODA (Feature Oriented Domain Analysis) [KCH+90], did not realise features explicitly in the domain engineering, which caused an intricate n-to-m mapping between features and domain artefacts. As a result, it was infeasible to construct a competent product line that could generate efficient systems without
1.1. RESEARCH PROBLEM

<table>
<thead>
<tr>
<th>Features</th>
<th>Home</th>
<th>Pro</th>
<th>Enterprise</th>
<th>Education</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows Update</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Device Encryption</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Remote Desktop</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>BitLocker</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Direct Access</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Device Guard</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Resilient File System</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Long-Term Servicing Branch</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1: Comparison of Windows 10 editions

manual aid [AK09]. Further work, such as FOSD (feature-oriented software development), makes progress in variability handling by establishing one-to-many mapping between features and domain artefacts. The researchers called such a mapping a clean mapping, due to features can be traced through design and implementation [ASLK10a, SAKS16]. As a result, FOSD approaches have ability to support automated generation of software products, which is a milestone in PLE. Notwithstanding, further steps must be made in order to realise the full potential of FOSD [AK09].

Ideally, the mapping between features and domain artefacts is one-to-one [KA13], which is the ultimate goal of FOSD [AK09]. Previous researches have pointed out the necessity of one-to-one mapping when deciding the variability realisation technique [BC13, BBM05]. Firstly, this significantly simplifies the process of configuration in the application engineering, and avoids the complicated defect diagnosis and fixing situations. For instance, if a variation is annotated by #ifdef statements in every module of the system, one mistake in a module can lead to a number of erroneous products that act in unpredictable ways due to misconfiguration. Secondly, without one-to-one mapping, it is difficult to add new variations, or replace an existing variations, at later stages.

However, one-to-one mapping seems difficult to be achieved. For instance, traditional FOSD approaches implement features in low-level programming languages, e.g., C-based or Java-based languages, but features normally do not align well with the object-oriented or functional decomposition established by the program structure.
Therefore, in many cases, a feature is mapped onto multiple code fragments, each of which refines a class, respectively. So far, FOSD approaches, including annotation-based approaches (e.g., CIDE [KTA08]), aspect-oriented programming (e.g., FeatureC++ [ALRS05]), feature-oriented programming (e.g., FeatureHouse [AKL09]) and delta-oriented programming (e.g., Δ-MontiArc [HKR+11]), stick with the one-to-many mapping.

Another problem of FOSD is the lack of family modelling. The specification of the structure and the behaviour of features and their interactions is critically important that deserves more attentions [AK09]. The value of modelling [CN04] is evident in: (i) better understanding of the engineering situation at hand in order to craft a better family model, (ii) tracing the family design back to the requirements that helps to construct a correct product family, (iii) facilitating quick and frequent changes in high levels of abstraction, and (iv) creation of visualisations of feature implementations. Moreover, without a systematic family modelling process, FOSD approaches cannot verify that a composition of features satisfied the specification of single features thereof, and the requirements of desired products in the family [AK09].

Component-based development (CBD) has been considered as a sensible development paradigm for modelling software product families. FORM (Feature-Oriented Reuse Method) [KLLK03], a previous work on domain analysis, points out: “By designing each selectable feature as a separate component, applications can be derived easily from product-line software. If there is difficulty in establishing this relation, the feature must be refined into specific features so that features can be easily mapped into architectural components.” However, traditional component models fail in this attempt, due to initially the variation has not been considered within the components, i.e., the variation points are not modelled explicitly. As a consequent, in practice, it is impossible to detach the code of a variable feature in a component for further composition. In summary, using traditional component models in FOSD anyway sabotages the mapping between features and components [ARNRS06].

During the domain implementation phase, there are two challenges: feature interactions and crosscutting concerns. The former is defined as: “A feature interaction is a situation in which two or more features exhibit unexpected behaviour that does not occur when the features are used in isolation” [AK09]. It is a complex matter, which has not been addressed completely so far. FOSD adopt refactoring derivatives to insert additional behaviour of a feature that interact with one another [SSdCMdA18]. It looks promising in small family example, but not practical when scaling to massive
numbers of features with numerous interactions, or if the interaction requires some completely different behaviour in comparison with individual features [TAK+14]. On the other hand, crosscutting concerns can be easily resolved by aspect-oriented software development (AOSD), but both FOSD and CBD cannot address it unless heavily violating the modularity of the family model. It is not clear whether integrating AOSD into FOSD can avoid the latent errors.

Finally, the main challenge at present in PLE is how to verify that if a product family functions correctly. Ideally, the testing should be family-based, not application-based, i.e., without generating all products [AK09]. However, so far, none of PLE approaches can achieve that. The reason is clear: the flexibility of feature composition results in exponential number of variants, which takes countless hours and effort to test all of them. Contrariwise, our approach models and implements whole software families, i.e., all valid products in one go, which eventually makes the family-based testing possible.

To summarise, current PLE approaches cannot model and implement whole product families during the domain engineering phase. The quality of software product families highly depends on the mappings between features and domain artefacts, but even FOSD cannot guarantee one-to-one relationship of feature mapping. The family modelling has not received enough attentions. Crosscutting concerns and feature interaction problems are still the two greatest challenges in the domain implementation. Furthermore, the families cannot be tested without generating concrete software products.

1.2 Research Question

This thesis proposes to put forward a feature-oriented component-based approach to construct product families. Integrating CBD into FOSD is not a new concept, as some researches tried before [ZMZY05, TCPB07]. The general idea is to decompose the n-to-m mappings between features and components into one-to-many, even one-to-one, mappings. Thus, the new approach may inherit advantages from both FOSD and CBD. However, the earlier work on feature-oriented component-based development has many limitations.

With respect to the problems we observed, the main research questions are concluded as below:

- Can we identify product family engineering with product line engineering, as
well as enumerative variability with parametric variability?

- Can we define a new component model to realise the enumerative variability?
- Can we use the new component model to define a new approach that constructs a whole product family in a feature-oriented manner?
- Can the proposed approach address feature interactions in the domain implementation?
- Can the proposed approach address crosscutting concerns in the domain implementation?
- Can the product families constructed by proposed approach be tested in a family-based fashion?
- Can we develop a visualised tool to implement the proposed approach?
- Can we exemplify the proposed approach by a use case in industrial setting?
- Can we evaluate our approach and product families using our approach?

1.3 Research Aim and Objectives

The primary aim of the research is to investigate the feasibility of devising an approach for constructing scalable, maintainable and evolvable software product family that consists of all valid products. To achieved the aim, we consider the complementary strengths of FOSD and CBD.

Firstly, we choose FOSD as the fundamental development paradigm of the new approach, due to FOSD facilitates the structure, reuse, and variation of software in a systematic and uniform way [AK09]. On the other hand, feature is the most important asset in FOSD, since it drives the development of every phase of the PLE life cycle. Therefore, FOSD pays more attention to a clean feature mapping than other PLE paradigms.

Secondly, the choice of CBD is thoughtful. A component is an independent executable unit [SGM02], while a feature represents a logical unit of behaviour specified by a set of functional requirements [Bos00]. Components compose with each other to create a systems, while features are a major input for composability design of the
product family \([M^+03]\). Thus, components are suitable as the implementations of features. Furthermore, CBD provides a flexible reuse at the architectural level, which can enhance the reusability of product families.

In regarding to CBD, we decide to adapt FX-MAN \([Ld17, dCTL^+16]\), a component model enforcing algebraic hierarchical compositional construction, to yield composition of variants of components for product family construction. Notably, the original FX-MAN was not at all ideal for our research purpose, but we retain its distinctive essence in the new version. An FX-MAN component is also known as an encapsulated component \([Ld17]\), as it only exposes a provided service, resulting in composition with very loose couplings. This essence is very useful for product family construction and testing.

In summary, to achieve the research aim, this work presents the following objectives:

- Analyse current PLE approaches, especially FOSD approaches, and use the investigation result as related work for our research;
- Define a new version of FX-MAN component model for seamless integration of CBD into FOSD.
- Provide step-by-step instructions of how to use our approach for modelling and implementing software product families. Common PLE problems, such as cross-cutting concern and feature interaction, should be addressed.
- Investigate family-based testing techniques. A product family should be tested at different granularity levels without generating product.
- Develop a toolset for our approach, which should support every step in our approach to model and implement a whole product family.
- Evaluate our approach on an industrial use case, and compare with existing PLE approaches.

1.4 Research Contributions

This research purpose the following contributions:

- The comparison between product family engineering and product line engineering by means of the usage of parametric variability and enumerative variability.
• The presentation of a new version of FX-MAN component model, which supports modelling and implementing enumerative variability.

• A detailed instruction to model and implement product families from scratch, which establish the one-to-one feature mappings. Particularly, we present how to address crosscutting concerns and feature interactions by our approach.

• The denotation of domain testing methods in a family-based fashion. Based on the whole product family constructed by enumerative variability, we can test a product family at different granularity levels, which includes domain unit testing, domain integration testing and domain system testing.

• The development of a toolset, which supports (1) feature modelling, (2) component modelling and implementation, (3) family construction, (4) component/product testing, and (5) product generation and downloading.

• An real-world example for the purpose of proving the feasibility of our approach.

• The evaluation of product families constructed by enumerative variability in terms of quality attributes and a family evaluation framework.

1.5 Research Methodology

Figure 1.2 illustrates the overview of research methodology employed for our research work [DHTA12]. Here, we briefly describe the goal of each stage.

Defining Research Problem

During a research project, researchers always meet theoretical or practical difficulties, and seek solutions for them. Each of the difficulties is a research problem. For example, during the research, we find out that establishing one-to-one mappings between features and components is difficult, thus creating new component model becomes a research problem. Sometimes there are more than one solutions for a research problem, but new problems may emerge after an old one is solved. For instance, after we address the feature mapping problem, we realise that crosscutting concerns are hard to be solved when feature interactions are encapsulated. So, addressing crosscutting concerns becomes another research problem. Therefore, researchers should tackle a
The key to tackle a research problem is to understand its origin and nature. Surveying the related literature helps to define the problem clearly, while discussion with colleagues or someone who has experience in the same research area helps to develop new ideas. If the research problem is ambiguous, researchers must rephrase it.

**Reviewing Earlier Work**

The review includes two parts: (i) concepts and theories, and (ii) previous findings. The review may consume a lot of time, but it is worth. The process of reviewing earlier work helps to explore the research area deeper, hence clarify the research problem. The researchers should be able to know whether similar approaches and solutions have already existed. Additionally, a thorough review helps the researchers to evaluate their work by comparison with others.
CHAPTER 1. INTRODUCTION

Establishing Research Hypothesis

Research hypothesis is a theory in the early form, which means it can be a conjecture, or an imaginative thought. It can be derived directly by observation of the research problem, or developed by extending an existing theory. A valid hypothesis must be definite and precise so that it can explain all the linked facts, whereas an invalid one usually goes against well-known, established knowledge. Notably, a valid hypothesis is not always correct, but it is not useless, as it points out other directions of the investigation and increase the evidence for other theories.

Designing Research

Research design is an overall plan of activities in the research study. It involves establishing investigation strategy, collecting related evidence, analysing data inputs, and specifying methods. An efficient, accurate and reliable research depends on a solid research design. To summarise, it is the stable foundation for the entire research.

Collecting Data

Data is facts and statistics assumed as building blocks of any research. The data collected together is used to validate and evaluate the adopted methods. The major issue occurs in data collection is bias. It is inappropriate to collect particular data, and neglect the disadvantageous part, to satisfy the requirement and make the result looks good.

Analysing Data

Analysing data is an essential process, because the raw data cannot directly reveal the truth for most of the time. Data analysis contains two actions: editing and classification. The former verifies the raw data, i.e., detects errors and omissions, while the latter categorises data into groups based on the common attributes. In the researches, tabulation is always a wise choice to present and compare the results of data analysis.

Interpretation And Report

After the data analysis, researchers need to present the researching findings by means of interpreting the result. Writing the report is last step, and perhaps the most difficult
step for many researchers. A qualified report should explicitly present the research problems, research methods and research findings.

1.6 Thesis Outline

The rest of the thesis has been organised as Figure 1.3 and 1.4 shows:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Research Question</th>
<th>Research Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Chapter 2</strong></td>
<td>Can we identify product family engineering with product line engineering, as well as enumerative variability with parametric variability?</td>
<td>The comparison between product family engineering and product line engineering by means of the usage of parametric variability and enumerative variability.</td>
</tr>
<tr>
<td><strong>Chapter 3</strong></td>
<td>Can we define a new component model to realise the enumerative variability?</td>
<td>The presentation of a new version of FX-MAN component model, which supports modelling and implementing enumerative variability.</td>
</tr>
<tr>
<td><strong>Chapter 4</strong></td>
<td>Can we use the new component model to define a new approach that constructs a whole product family in a feature-oriented manner?</td>
<td>A detailed instruction to model and implement product families from scratch, which establish the one-to-one feature mappings. Particularly, we present how to address crosscutting concerns and feature interactions by our approach.</td>
</tr>
<tr>
<td></td>
<td>Can the proposed approach address feature interactions in the domain implementation?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Can the proposed approach address crosscutting concerns in the domain implementation?</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.3: Thesis outline (Chapter 2-4).
### Chapter 5

**Research Question:** Can the product families constructed by proposed approach be tested in a family-based fashion?

**Research Contribution:** The denotation of domain testing methods in a family-based fashion. Based on the whole product family constructed by enumerative variability, we can test a product family at different granularity levels, which includes domain unit testing, domain integration testing and domain system testing.

### Chapter 6

**Research Question:** Can we develop a visualised tool to implement the proposed approach?

**Research Contribution:** The development of a toolset, which supports (1) feature modelling, (2) component modelling and implementation, (3) family construction, (4) component/product testing, and (5) product generation and downloading.

### Chapter 7

**Research Question:** Can we exemplify the proposed approach by a use case in industrial setting?

**Research Contribution:** An real-world example for the purpose of proving the feasibility of our approach.

### Chapter 8

**Research Question:** Can we evaluate our approach and product families using our approach?

**Research Contribution:** The evaluation of product families constructed by enumerative variability in terms of quality attributes and a family evaluation framework.

### Chapter 9

**Research Question:** N/A

**Research Contribution:** Discussion, conclusion and future work.

Figure 1.4: Thesis outline (Chapter 5-9).
Chapter 2

Product Family Engineering Vs. Product Line Engineering

“The slightest difference leads to a huge loss.”
— Chinese Proverb

2.1 Introduction

Many researchers have conceded that the terms “product family engineering” (PFE) and “product line engineering” are interchangeable over the past two decades [DNF96]. But in fact, there is a subtle but critical difference.

PFE refers to an engineering methodology that constructs a product family architecture (PFA) describing all members of a family based on commonality as well as planned variabilities. Based on the PFA, new software products can be generated by systematic reuse of system components and structures [vdL02a]. PFE is an initiative against the rising problems in massive software industry, i.e., producing a set of related products at lower costs, in shorter time and with higher quality [Nei86].

By contrary, PLE is based on a so-called ‘product line’, which is an assembly line in essence. The usage of industrial assembly lines can trace back to late 19th century [RD06]. As a manufacturing process, an assembly line greatly improves efficiency with less labour, due to an end-product is assembled through sequentially adding standard pieces, instead of produced from scratch. Likewise, PLE aims at producing every single software product by assembling pre-implemented parts.
Although PFE and PLE both aim at generating significant savings in cost and time when producing nearly identical systems in the family, but their expected purposes are not quite same. From a customer perspective, a product line allows the customer to select the premier product that satisfies all needs, whereas a product family lets the customer to tailor a solution by acquiring and integrating a set of products, even all products, from the family [SB00]. From a developer perspective, there are more technical differences between PFE and PLE. We will discuss them in two aspects: framework and variability.

2.2 Framework

A framework provides a general and reusable development environment to particular problem area. A well-defined framework allows developers to dedicate their effort to the requirement, and implement an application in a standard way. Therefore, having an appropriate framework is important for constructing product families and product lines, as well as generating concrete products. The frameworks of PLE and PFE are similar, both consists of two life-cycle stages for domain engineering and application engineering. But the activities and artefacts involved are not quite same. In this section, we will elaborate the two frameworks and compare the differences.

2.2.1 Problem Space and Solution Space

The term problem space and solution space have been introduced in PLE for a long time. They are used to distinguish and connect various types of artefacts created in the different stages. Due to the diversity of PLE research, there are several definitions or explanations of problem and solution space, for example:

1. Berg and Bishop [BBM05]: “The problem space generally refers to systems’ specifications established during the domain analysis and requirements engineering phases, whereas the solution space refers to the concrete systems created during the architecture, design and implementation phases.”

2. Dhungana et al. [DGRN10]: “the problem space (stakeholder needs or desired features), the solution space (the architecture and the components of the technical solution)”
3. Czarnecki [Cza05]: “Problem space is a set of domain-specific abstractions that can be used to specify the desired system-family member. ... The solution space, on the other hand, consists of implementation-oriented abstractions, which can be instantiated to create implementations of the specifications expressed using the domain-specific abstractions from the problem space.”

4. Beuche and Dalgarno [BD07]: “(Problem space) reflects the desired range of applications (‘product variants’) in the Product Line (the ‘domain’) and their inter-dependencies. ... An associated Solution Space describes the constituent assets of the Product Line (the ‘platform’) and its relation to the problem space.”

5. Apel and Kästner [AK09]: “The problem space comprises concepts that describe the requirements on a software system and its intended behavior. The solution space comprises concepts that define how the requirements are satisfied and how the intended behavior is implemented.”

6. Groher and Voelter [GV09]: “The problem space is concerned with end-user understandable concepts representing the business domain of the product line. The solution space deals with the elements necessary for implementing the solution, typically IT relevant artifacts.”

7. Voelter and Visser [VV11]: “The Problem Space refers to the application domain in which the product line resides. The concepts found in the problem space are typically meaningful to non-programmers as well. The Solution Space refers to the technical space that is used to implement the products.”

By comparing and analysing these definitions, we conclude some common properties of problem and solution space. The artefacts in problem space draft the product family, or define a product variant thereof, at high-level abstraction. Commonality and variability in the problem space are identified and prepared for further implementation. By contrary, solution space comprises constructive artefacts for final assembly, and constructed products for execution and testing.

Based on our definition of problem and solution space, we define the PLE framework and PFE framework as the combinations of the problem and solution space with domain and application engineering, as shown in Figure 2.1 and 2.2.
2.2.2 PLE Framework

Figure 2.1 shows the PLE framework, which contains domain engineering and application engineering. During the two engineering phases, the involved artefacts are classified into problem space and solution space. Ideally, any artefact in the solution space should be traced back to artefacts in the problem space, and vice versa [BBM05]. Here, we will discuss the activities and artefacts in detail.

![Figure 2.1: The PLE framework.](image)

**Domain Engineering**

Domain engineering is the entire process of identifying, modelling and implementing commonalities and variabilities of a product line. It is the key part of PLE. In PLE, it aims to establish the reusable domain artefacts, e.g., feature model, components or code base, for further product generation in the application engineering. The domain engineering consists of four activities that are described as below:

- **Domain Analysis**

  In domain analysis, requirements are analysed and captured, and from these a variability model is defined in the problem space. A number of variability models have been proposed, such as feature models [KCH+90], decision models [W+99], and orthogonal variability models (OVM) [BGL+04]. Feature models are most widely used, as features are a natural way to describe a domain in terms of problem space concepts [Sch03]. Indeed feature models have proved sufficient for defining the overall product portfolio in an industrial setting [SGEL09]. Most PLE approaches use feature models,
although some use other variability models. Afterwards, the feature model is reused for product configuration in the application engineering.

- Domain Design

Domain design specifies the structural and behavioural properties of the features [AK09], e.g., the granularity of components [KAO11] or the extension points in a common framework [PBvDL05]. It is worth noting that a feature should be specified and modelled in isolation to ensure the safe composition of features without considering the implementation [AK09].

Many PLE approaches introduce UML as a standard to model features. Individual features and their relationships are realised by class diagrams [ZHJ04, ABM00, Gom05], state machine diagrams [ABM00, Gom05, BSS+12], activity diagrams [ABM00, CA05, AMAB+06], sequence diagrams [ZHJ04, ABM00, AJTK09], component diagrams [Gom05, Sch10, PHSK07] and so on. However, domain design deserves more attention, not just UML diagrams. Indeed, some PLE approaches use Architecture Description Language (ADL) instead of UML, which can decrease the ambiguity of software architectures, e.g., Koala [VOVDLKM00] and DPD-ADL [ZWZY10]. But most of the ADLs only serve ad hoc domains and lack support with good tools [Pan10]. When features are later selected in application engineering for product design, the correlated model pieces are merged together to model a product.

- Domain Implementation

In domain implementation, the artefacts modelled in the previous stage should be implemented, e.g., components and architecture. Moreover, this stage includes writing documentation, developing a domain-specific language and creating a tool, if any of them is required [Har02].

In PLE, no matter what development paradigms and variability mechanisms are utilised, the main purpose of domain implementation is to create a ‘product line’, which in essence is reusable code assets with feature mappings. The challenges arose in this stage refer to crosscutting concern and feature interaction problem. Features are encapsulated in the cohesive units, i.e., objects, which scattered across the code base, but current methods for addressing crosscutting concern and feature interaction always jeopardise the modularity [KAO11]. In other words, implementing crosscutting features and interacting features as individual features causes a departure from one of the
initial goals of domain implementation, which is to build explicit mapping between features and domain artefacts [AK09].

- **Domain Testing**

Domain testing aims at validating and verifying all the artefacts modelled and implemented earlier [LSR07]. Due to PLE generates products in the application engineering, there is no running application to be tested in domain testing [BPvdL05]. Instead, only off-the-shelf components or integrated chunks composed of common parts can be tested [JHQJ08], whose test case may be reusable for product testing in the end of application engineering. However, it is very difficult to test all domain artefacts completely in a product line (except for trivial cases), because the feature implementations may be scattered across many code snippets in a large code base [PM06]. Thus, many PLE approaches have to bypass the domain testing and focus on application testing [KPRR03].

**Application Engineering**

Application engineering is a process of product-specific development that aims at developing a member of the product line, by assembling feature implementations pre-constructed in the domain engineering. It constituted a significant part of early PLE theory, yet has been considered harmful in practice, especially for software mass customisation [Kru10]. Recent research points out that in the next generation of PLE, application engineering will shrink to almost nothing [KC13], which is exact what our approach has achieved. The application engineering consists of four activities described as below:

- **Application Analysis**

In all PLE approaches, application engineering starts with a product configuration, parameterised on the selected or unselected features. The configuration model is derived from the feature model defined in domain analysis. Invalid product variants, i.e., irrational feature combinations, should be detected in application analysis to prevent illegitimate products are assembled in the following stage.

On the contrary, in our approach, the PFA realises the feature relationships, hence the variability has been enumerated. As a result, the configuration has not to be carried out.
2.2. FRAMEWORK

- Application Design

After a product variant is specified, application design defines the structure and behaviour for the single variant at hand. Notably, the application design does not start from scratch, it is derived from the result of domain design by binding variability specified in application analysis. In addition, particular requirements of a customised product should be considered in this stage. The result of this stage is helpful for application validation.

- Application Implementation

Application implementation covers the selection and assembly of domain artefacts according to the features in the configuration. The result of this stage is an executable software product. However, not every PLE approach supports automated generation. If the feature mapping is not explicit, i.e., the relationship is many-to-many, the final software products need manually writing glue or boilerplate code. The lack of tool support makes the resulting products difficult to be derived, too. Extra functions should be implemented here for customisation.

- Application Testing

Application testing validates and verifies an executable software product against its specification. Traditional software testing techniques can be used in this stage, such as unit testing, integration testing and system testing. Notably, some test cases created from domain testing can be reused to reduce the overall testing effort. However it is impossible to test all single software products in a family.

2.2.3 PFE Framework

Figure 2.2 shows the PFE framework. The domain engineering still exists, but the goals and outcomes of each activity may not be the same. Another major difference is the application engineering, which used to constitute a significant part of early PLE theory, yet has been considered harmful in practice [Kru10]. We do not illustrate the application engineering in Figure 2.2, due to the application analysis and application design are not essential in PFE.
CHAPTER 2. PFE VS. PLE

Domain Engineering

The key artefact in domain engineering is PFA. A well-defined PFA embodies every product’s architecture by explicit variation points, thus any number of products can be extracted without the aid of configuration. In addition, the correctness of an individual product has been determined when the whole family is already tested in the domain engineering. Thereby, the only task remaining in the application engineering is product customisation, i.e., adding special functions or unique features. The four activities in the domain engineering are presented as below:

- **Domain Analysis**

  Except for analysing the domain requirements and capturing commonalities and variabilities, domain analysis in PFE should define a functional model, as well as a feature model. The functional model describes the overall behaviour of the product family, and the behaviour of each variant thereof [KCH+90]. In our approach, we do not create an explicit functional model from the requirements, but when creating the product family we follow the requirements (and the feature model) and the resulting product family actually defines a statechart, i.e., a functional model [QL18b].

- **Domain Design**

  In PFE, domain design aims at structuring a PFA, which can be considered as a ‘normal’ architecture extended with well-defined variation points in practical terms [BBSdS08, DVDH02]. Comparing to the product line, PFA is a higher level structure for a set of
related products [DR08]. In other words, each product member can derive its own architecture from the PFA [DSB05]. However, after years of study on PFA, researchers start to realise that constructing a qualified PFA is not an easy task. In our approach, we create a new component model, which shows the feasibility to construct a PFA as a composition of variants of components [QL17].

- Domain Implementation

Domain implementation aims at implementing the designed PFA. Unfortunately, most of the PFAs are created by ad hoc ADLs, which are only textual and not executable [Pan10], and not even mention to the supporting tools. Moreover, the realisation of feature interactions or crosscutting features anyway leads to additional component dependencies, which eventually sabotage the architecture modularity and integrity [KKB08, OJC+12]. Ideally, the outcome of domain implementation is the whole product family, i.e., all valid and executable software products.

- Domain Testing

Domain testing performs family-based testing [KM03]. It does not need concrete products, only uses domain artefacts. Moreover, invalid feature combinations should be detected as well as the unexpected behaviour in the family.

Application Engineering

As shown in Figure 2.2, the application engineering is excluded from the PFE framework. But, even so, that does not mean that the application engineering is completely useless in PFE. Actually, the reason is simple: most of the activities and artefacts belonging to the application engineering are no longer necessary in PFE. For example, PLE approaches do not model and implement the variation points, hence the relations between features cannot be realised. As a direct result, user has to adopt a product configuration at the application analysis phase, in order to link the feature implementations (usually code fragments) together. But in PFE, the constructed PFA already realises the feature relations as component relations. So, the application analysis may not be needed. On the other hand, the application testing is inevitable in PLE, due to the family cannot be tested in the domain engineering. However, in PFE, the whole family ought to be tested by the end of the domain engineering, i.e., all valid products have been verified. Thereby, the application testing phase is not compulsory in PFE.
There are still a few activities remaining in the application engineering, such as *customisation*. For instance, sometimes a customer wants to add a unique feature on a product for personal reasons. Such modifications should not affect other products in the family, thus must be accomplished in the application engineering.

## 2.3 Variability

Variability modelling is at the heart of product family development [PBvDL05]. In the problem space, variability in a product family is specified by a feature model, and in the solution space, product variants are created according to this variability. Variability defined by a feature model (in problem space) is *enumerative* in nature, as it includes all valid variants. In contrast, in solution space, current PLE approaches use *parametric* variability\(^1\), i.e., variability parameterised on features occurring in a single product variant, and configure code (from a code base) for one product variant at a time.

This divergence is due to the perception that enumerative variability is merely a model whereas parametric variability deals with implementation, i.e., real code. In this section we will discuss the details.

### 2.3.1 Enumerative Variability

In the domain engineering phase, the standard way to define variability in the problem space is to use a feature model. Figure 2.3 shows a simple example for a family of vending machines. A vending machine can sell tea only, coffee only, or both tea and coffee. The customer can pay in cash, gift voucher, or by card. A machine may give change for cash payments but not for payments in gift vouchers. A machine that does give change may offer a cash-back facility for card payments.

A feature model names all the available features and sub-features, and shows all the valid combinations thereof in the product family [KCH+90]. Features and their sub-features are depicted as parent-child relationships\(^2\), and variability is defined by *variation points*: mandatory, optional, alternative (exclusive ‘or’), and or (inclusive ‘or’); and may be constrained by cross-tree relationships, e.g., ‘feature *Gift Voucher* excludes feature *Change*’ [KOD10]. In Figure 2.3, valid variants include: ‘Coffee, Cash, Change’, ‘Tea, Cash, Change’, ‘Coffee, Tea, Cash, Change’, but not

---

\(^1\)Not to be confused with feature attributes [CSHL13].

\(^2\)Features with child features are *abstract* features; *leaf features* have no child features.
2.3. VARIABILITY

15 Valid Variants:
- Coffee,Cash
- Tea,Cash
- Coffee,Tea,Cash
- Coffee,Gift Voucher
- Tea,Gift Voucher
- Coffee,Card
- Tea,Card
- Tea,Cash,Change
- Coffee,Tea,Cash,Change
- Coffee,Card,Change
- Coffee,Tea,Card,Change
- Coffee,Tea,Cash,Card,Change

Figure 2.3: Vending machines: Feature model.

‘Coffee, Gift Voucher, Change’, due to the ‘Gift Voucher excludes Change’ constraint.

A feature model enumerates all the valid combinations of features, i.e., all the valid product variants. In Figure 2.3, there are a total of 15 valid variants. The variability defined by a feature model is thus enumerative variability. It provides a configuration model for all product variants to be constructed in the solution space.

2.3.2 Parametric Variability

In the application engineering phase, code has to be constructed in the solution space for valid product variants specified by the feature model. Current PLE approaches use parametric variability, i.e., variability parameterised on features occurring in a single product variant, and configure code (from a code base) for one product variant at a time. The behaviour of the features (and hence the product variants) has to be implemented according to their functional requirements.

The general approach to code construction is to start with a code base that consists of code for all the mandatory features (i.e., the base system), with extension points for excluding or including code fragments or modules that implement the variable features.
Then for a chosen valid variant specified by the feature model, i.e., an instance of the feature model, called a configuration for this product, code is constructed from the code base by excluding code for unselected features, or including code for selected features. Approaches based on code exclusion and inclusion are said to adopt negative variability and positive variability respectively [GV07a].

Negative variability and positive variability are illustrated in Figure 2.4 and Figure 2.5 respectively for the vending machine family.

```java
class VendingMachine {
    static void main (String[] args) {
        float price=0;
        //ifdef Coffee && Tea
        if(args[0].Contains("Coffee"))
            //endif
        //ifdef Coffee
        //endif
        if(args[0].Contains("Tea"))
            //endif
        //ifdef Tea
        price+=Tea.price();
    }
}
```

(a) Code base (150% model)

Configuration: Tea=false, Gift Voucher=false, Card=false

(b) Variant configuration

Figure 2.4: Negative (parametric) variability in solution space.

Figure 2.4(a) shows the code base. It includes code fragments for all features, both mandatory and variable, i.e., more than will appear in any valid variant (for this reason this kind of code base is called a 150% model). Code fragments for variable features are placed at extension points annotated by boolean conditions which define feature selection (e.g., #ifdef Coffee) and are to be excluded for non-selected features. Figure 2.4(b) shows a specific configuration, i.e., a valid variant specified by the feature
model, which excludes the features TEA, GIFT VOUCHER and CARD; it is the variant: ‘COFFEE, CASH, CHANGE’. The code for this variant is constructed from the code base in Figure 2.4(a) by excluding code fragments for TEA, GIFT VOUCHER and CARD.

Negative variability is adopted by annotative PLE approaches, which are widely used in industry [BRN+13], with leading commercial tools such as pure::variants [Beu12] and Gears [KC13]. The example in Figure 2.4 is annotated by C preprocessor (cpp) [LAL+10].

PLE approaches based on architecture description languages (ADL), e.g., Koala [ASM03] and xADL 2.0 [DHT05] use boolean connectors to compose components and to exclude unselected ones in a configuration. Therefore, these approaches also adopt negative variability.

In positive variability, the code base is the base model consisting of code for mandatory features only. For a selected configuration, code fragments for the selected variable features are superimposed on the base model. Figure 2.5(b) shows a feature-oriented programming (FOP) [ABKS13a] example. The base model is just the VendingMachine class. Variable features are called layers and are superimposed on the base model. For the configuration in Figure 2.5(a), code fragments for the selected features COFFEE, CASH and CHANGE are superimposed on the base model by successive refinements: the Coffee class extends the VendingMachine class to give a class that is then extended by the Cash class; the resulting class is then extended by the Change class.

However, either negative or positive variability is not flexible enough to construct products in PLE. Thereby, transformational approaches are proposed. This kind of approaches has a lot of similarity to the compositional approaches at the implementation level [VAA13]. Both of them design and implement model or code fragments isolated from the base programs. The difference is that transformational approaches realise positive and negative variability, instead of only one of them. That means, transformational approaches enable to add, modify or remove functionalities to generate a software product. To summarise, transformational approaches are more flexible, while more error-prone, than annotative and compositional ones.

We summarise the relation between PLE approaches and parametric variabilities in Table 2.1. However, neither negative nor positive variability is able to construct a PFA embedding all valid products. Parametric variability is a variability mechanism that offers several implementation techniques to delay the design decisions about behaviour and structure of a software product [VGBS01]. As a natural result, PLE is not aware of any concrete product before configuration in the application engineering, and thus
family-based domain testing is not available.

What we learn from the parametric variability in the PLE context helps us to understand the variability required by PFE. As Figure 2.3 depicts, the variation points not only locate where the variability occurs, but also indicate the relations between features [M+03]. Current PLE approaches implement the features, but not the variation points. Consequently, without a configuration, the variabilities of a product cannot be determined. Therefore, it becomes obvious that PFE needs implementation of the variation points, as well as the features, within the PFAs. As a matter of fact, some ADL-based PLE approaches can model partial variation points in the architecture during the domain design phase, but eventually fail to implement them during the domain implementation phase, due to their underlying component models cannot preserve the compositionality of variants. For example, Koala creates a special construct called
Table 2.1: Domain artefacts in PLE approaches.

<table>
<thead>
<tr>
<th>PLE Approaches</th>
<th>Domain Artefacts</th>
<th>Domain Variability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Domain Analysis</td>
<td>Domain Design</td>
</tr>
<tr>
<td>Annotative approaches</td>
<td>Variability model</td>
<td>150% model</td>
</tr>
<tr>
<td>Compositional approaches</td>
<td>Variability model</td>
<td>Model fragments</td>
</tr>
<tr>
<td>Transformational approaches</td>
<td>Variability model</td>
<td>Model fragments</td>
</tr>
</tbody>
</table>

switch to realise the alternative variation point in the architecture, but finally has to removes the unselected components from the source code according to the configuration [VOVDLKM00].

In Chapter 3, we will present a novel component model that is able to model and implement variation points, and hence construct the enumerative variability for whole product families.

## 2.4 Related Paradigms and Approaches

So far, a number of development paradigms and a considerable amount of PLE approaches have been put forwards. Although none of them can build up a PFA with enumerative variability, some are instructive. In this section, we will discuss those representative paradigms and approaches that enlightening us.

### 2.4.1 Existing PLE Paradigms

A software development paradigm is known as: “a general approach to undertaking software development, a way of thinking about software development, and a metaphor for software development” [AI13]. In this section, we will introduce several development paradigms and analyse their performances.

**Object-oriented Software Development**

Since mid-1990s, object-oriented software development (OOSD) becomes the most popular software development paradigm. It defines a life cycle divided up into four
stages going from abstract descriptions of the problem to designs then to code and testing [HSE90]. The similar life cycles are adopted in PLE for domain engineering and application engineering. OOSD is straightforward and easy to apply. Thereby, many PLE approaches design product lines by UML, and program product lines by object-oriented language [Gom05, ZHJ04, RBSP02, AMS07]. Furthermore, OOSD is flexible because it touches low-level code directly. For instance, the code is implemented not only at capability layer for services and operations but also at operating environment layer for simulations and user interfaces [KLD02, MTS+17]. However, initially, OOSD does not consider variations, i.e., there is no variable object in a program. To identify the variability at design time, annotations must be involved, e.g., stereotypes of UML diagrams [Gom05, ZHJ04], or macros of code files [Beu12, KLD02]. However, a software product line of non-trivial size requires huge amount of annotations for its implementation, which can lead to many problems, such as “#ifdef hell”. As a result, product lines developed by OOSD usually have poor diagnostics and terrible maintenance [LST+06, KA09].

Aspect-oriented Software Development

Since feature model has been proposed for PLE, it has been frequently observed that crosscutting concerns occur between features [ALS08, KAB07]. A crosscutting concern is defined as follows: “a concern that does not align well with the structure established by object-oriented or functional decomposition” [AK09]. For instance, a feature ‘PAUSE’ in a game product line can be triggered at any time when playing a game [FSM16], or a feature ‘CANCEL’ in a vending machine product line can be invoked at any stage during the purchase [CHS+10]. Hence, aspect-oriented software development (AOSD) is put forward to deal with the crosscutting concerns in PLE [MO04, MBJ08, Gri00, ALS06]. However, just like OOSD, handling variability in design time is not the initial goal of AOSD. The variability in the problem space does not always match to an aspect. Moreover, many semantic problems have been found in AOSD [KS04, HNB06, SF06]. For example, if more than one aspect is woven at the same join point, the interference may result in errors. Unfortunately, this situation is quite common in the context of PLE, e.g., sibling features [HLHE13].
2.4. RELATED PARADIGMS AND APPROACHES

Component-based Development

Component-based development (CBD), also known as component-based software engineering (CBSE), aims to compose software systems on demand by off-the-shelf components [HC01, Ld17]. Thus, by using CBD, a system is a composition of components, instead of monolithic entity. A distinguishing property of CBD is reusability. The components should be designed in a fashion that enables them to adapt different applications, e.g., software products in a family. Other properties of CBD may also be useful to PLE, such as replaceability and extensibility. The former indicates a component can be substituted easily and quickly when it is no longer suitable, which improves the maintainability of product families; whereas the latter denotes a component can be further adjusted for additional functionality, which enhances the evolvability of product families. In addition, components are considered as black boxes, which provide extra security for software product families. In consideration of above mentioned benefits, CBD has been used in many PLE approaches, e.g., KobrA [ABM00], Koala [VOVDLKLM00], Koalish [ASM03], COSMOS [DTR+10], LightPL-ACME [SMCB13], etc. However, CBD has a serious issue that limits its usage in PLE. That is, features usually do not align well with the decomposition imposed by component models [AK09], especially when crosscutting concerns exist in a product family [Ape07]. For example, detaching the code of ‘PAUSE’ from a game software and encapsulating it in every other components is not possible in practice [FSM16]. In summary, without a suitable component model, using components to implement features can cause inexplicit mappings between them. We will discuss more details of CBD in Chapter 3.

Feature-oriented Software Development

Feature-oriented software development (FOSD) aims at constructing software product families by incremental feature implementations. The heart of FOSD is feature, which is regarded as the first-class citizen to analyse, design, implement, or evolve a software system [AK09]. A feature represents a logical unit of behaviour specified by a set of functional requirements [Bos00], thus features can be used to explicitly and systematically capture commonalities and variabilities in order to define a product family, and software products thereof, and facilitate reuse at late stages of PLE.

FOSD essentially targets on three distinctive characteristics: structure, reuse, and variation [KA13]. By comparison with other development paradigms, FOSD has made
a great progress by establishing a clean mapping between features and domain artefacts. Ideally, the mapping should be one-to-one, but it is very difficult to be achieved in current FOSD approaches. Nevertheless, the one-to-many mapping is capable of describing which artefacts belongs to which feature, and leads to automatic product generation [KA13, BSR04]. However, as we mentioned in Chapter 1.1, FOSD is not perfect. It requires further steps to realise its full potential.

2.4.2 Existing PLE Approaches

Table 2.1 presents three categories of PLE approaches in terms of variability mechanism. In this section, we explain how the variability works in current PLE approaches.

**Annotative Approaches**

Annotative approaches usually build a single, superimposed model, namely 150% model, to represent all product variants. The variable features are implemented as code fragments with annotations, i.e., boolean feature expressions. Subsequently, in order to generate a product, code fragments corresponding to unselected features have to be removed according to the product configuration. Therefore, annotative approaches only construct negative variability in the solution space. Here we introduce some annotative approaches as following:

- **FORM**

FORM (*Feature-oriented Reuse Method*) [KKL+98, KLD02, Kan98] is an extension of FODA (*Feature-oriented Domain Analysis*) [KCH+90]. FODA provides a systematic method to define features based on the identified commonalities and variabilities in the domain. Moreover, FODA establishes functional models, in the form of STATEMATE statechart and activity-chart [HLN+90, HN96], at the family level, which describe the overall behaviour of the product family, and the behaviour of each product within. However, FODA does not offer the guideline on domain implementation. FORM extends FODA into this space by parametrisation of reusable artefacts using features in the solution space. At code level, FORM devises a macro language in the component specification to map the features onto code fragments [KLD02]. FORM is an inspiring approach in PLE, but it lacks tool support and proof of feasibility in practice.
- **Pure::variants**

Pure::variants is the market leader for software product line and variant management [BPSP04, BD07, Beu12]. In the problem space, it supports feature modelling and configuration with a flexible cross-tree constraints in a tree view. In the solution space, pure::variants allows any kinds of domain artefacts, e.g., code, models and documents, to be annotated by compound feature expressions. After a product variant is determined by configuration, the tool automatically generates a *variant description model* (VDM), which can be regarded as a single problem from the problem space. Eventually, a VDM results in a single concrete software product. Notably, as a commercial tool, pure::variants pays more attention to offering the solutions. That means, the process of constructing software product family in pure::variants is not always feature-oriented.

- **CIDE**

CIDE, acronym of *Coloured Integrated Development Environment*, is proposed to software product lines by refactoring a legacy code base [Käs07, KAK08]. It set up mappings between features and elements of arbitrary *abstract syntax trees* (AST), instead of disordered code fragments [KKB08]. Hence, the refactoring process is feature-oriented [KAK08, KKB08]. CIDE has a unique property that aims to not obfuscate the source code with additional annotations, unlike other annotative approaches. Instead, CIDE applies an editor in which code can be annotated with different background colours. Each colour represent a feature, while the feature name is shown in a tooltip.

If a code fragment is the implementation of multiple features, its background colour is a mix of those features’ symbolic colours, e.g., yellow + blue = green. Briefly, CIDE avoids the “#ifdef hell” and improves the program comprehension [FKA+13]. But the chosen of colour can be problematic, especially when a product family is filled with numerous features.

- **Compositional Approaches**

In compositional approaches, variability in the problem space is designed as model fragments, while the mapping is manually defined based on the structural and behavioural properties of the variability [AK09]. e.g., the granularity of components [KAO11] or the extension points in a common framework [PBvDL05]. The realisation of mappings depends on specific implementation technique and language. In the
domain implementation phase, code fragments are developed isolated from the base programs. In order to compose a concrete product model, aspects are woven in the base model [BGB+06, LMV+07, LKKP06], whereas model fragments are merged together [ABKS13a, SPR04]. Therefore, compositional approaches only construct positive variability in the solution space. Here we introduce some compositional approaches as following:

- **XWeave**

  In AOSD, the term ‘weaving’ describes a process that links aspects with other objects to create an advised object [JHD+04]. An aspect weaver inspects the aspect-oriented program and generates desired object-oriented code with the aspects injected. XWeave is a weaver that supports weaving of both models and meta models [GV07b]. In the context of PLE, XWeave integrates variable pieces of architectural models into base system according to product configurations. The greatest advantage of XWeave is to deal with crosscutting concern. It can easily add same piece of advice to several places, as well as apply different pieces of advice to different places. The former is so-called homogeneous weaving, while the latter is heterogeneous weaving [LH06]. However, like traditional AOSD approaches, XWeave only supporting additive weaving. It cannot modify or remove code in base system, which limits its usage in PLE.

- **Plastic Partial Components**

  A plastic partial component (PPC) is a component with built-in variability points [PDCSG09]. Part of its behaviour is common to a software product family, the rest changes depending on the products. A crosscutting feature is implemented as an aspect, whereas a non-crosscutting one is encapsulated into an entity called feature. Aspects and features are selected and woven into the components at the variability points according to the product configurations. However, there is no variable component in a variant, as common and variable functionality are both encapsulated in the same component. Therefore, the component model adopted by PPC is not truly hierarchical [GOS12].

- **FeatureC++**

  FeatureC++ is a programming language extending C++, which supports feature-oriented programming (FOP) and aspect-oriented programming (AOP) [ALRS05, ALRS06]. A
2.4. RELATED PARADIGMS AND APPROACHES

feature is implemented by a **Mixin Layer**, which contains a group of **Mixins** that implement class fragments. Hence, a Mixin Layer crosscut multiple classes. A set of related Mixins that constitute together a complete class is called a **refinement chain**. In a refinement chain, the start point is called **constant**, the rest are called **refinements**. Refinements are able to refine their constant, i.e., a feature can refine a set of classes by adding and overriding methods. However, the FOP has some well-known issues in crosscutting modularity [MO04, ALRS05]. For instance, FOP can only deals with (1) heterogeneous crosscutting concerns that apply different code fragments to different places [LH06], (2) refinements structure aligning well with their constant structure, (3) refined methods with fixed signature, i.e., no argument list, (4) method extensions with trivial size, i.e., refinements do not crosscut too many classes, and (5) static crosscutting, i.e., aspect woven at design time instead of runtime. Therefore, FeatureC++ implements AOP extensions by using **AspectC++** [SGSP02, SLU05] with the aim of solving the problems mentioned above.

**Transformational Approaches**

At some points, compositional and transformational approaches are difficult to distinguish at the implementation level. Both of them design and implement model or code fragments isolated from the base programs. The difference is that compositional approaches only define positive variability, but transformational approaches define positive and negative variability. That means, transformational approaches enable to add, modify or remove functionalities to generate a software product. To summarise, transformational approaches are more flexible, while more error-prone, than annotative and compositional ones. Here we introduce some transformational approaches as following:

- **DeltaJava**

DeltaJava is a **delta-oriented programming** (DOP) language that provides more flexibility for constructing software product families than FOP does [SBB+10]. Similar to FOP, DOP designs and implements a core module and a collection of isolated modules. These modules are called **delta modules**, which identify the changes to be performed on the core module, in order to generate further products by adding, modifying and removing code. Another progress DeltaJava brings is the solution of optional feature problem [KAR+09]. It occurs when two optional features seem independent at domain level, i.e., in the feature model, but somehow they interact at the implementation level,
i.e., require additional code for their interaction. Usually such a problem is caused by a detected feature interaction. By contrast with FOP, in DeltaJava, a delta module is not just mapped onto one single feature, but it can be mapped onto any combination of features. The application condition of a delta module explicitly illustrates that the code in this delta module is implemented for a specific feature, or a combination of specific features, or a derivative between specific features. Consequently, a product is generated by automatically injecting delta modules to the core module in a linear ordering that respects the partial order induced by the after clauses in the application condition.

- **\(\Delta\)-MontiArc**

*Architectural description language* (ADL) aims at modelling building blocks, e.g., components, interfaces and connectors, in order to construct software architectures with high demands on software quality. *MontiArc* [HRR14] is a well-known ADL developed with a framework called *MontiCore* [KRV10], which offers not only domain-specific language for implementation, but also reuse mechanism at the implementation level. \(\Delta\)-MontiArc [HKR+11], as its name implies, is an integration of delta modelling into MontiArc. Therefore, it can model and manage the variability at the architectural level. The basic idea of \(\Delta\)-MontiArc is similar to other delta-oriented development approaches: (i) building a core architecture assembled with core modules, (ii) specifying a set of delta modules for all possible products, and (iii) determining a subset of delta modules to derive a particular product. Logical expressions in the delta modules define the application order constraints. Notably, delta modelling supports reactive and extractive product line development, as well as proactive one [HRRS11]. So, \(\Delta\)-MontiArc usually chooses a complete product architecture as the core architecture. Other products are generated by modifying the first product.

- **PEoPL**

Projectional editing of product lines (PEoPL) is an approach supporting both annotative and compositional projections, i.e., editable code fragments reflecting two types of variability representations, namely internal and external representation [BPB17]. The former models variable feature artefacts as nodes in a variational abstract syntax tree (AST), whereas the latter implements related code fragments using different development paradigms, such as annotations and feature modules. In the AST, modules can be composed by three operations: (i) addition, (ii) subtraction and (iii) overriding.
Accordingly, the code fragments are added, removed or modified to generate a configured product. Hence, PEOPL can be used to implement delta-oriented software product lines [BF16]. From a purely practical point of view, PEOPL combines the advantages of usages of preprocessor and modular, but it does not overcome the drawbacks of annotative and compositional approaches.

2.5 Summary

For the past decades, PLE has been regarded as an equivalence of PFE in many researches. However, there is a subtle but crucial difference between them. PFE focuses on building an underlying architecture of an organization’s product platform that implement commonalities as well as planned variabilities, whereas PLE emphasises the creation of software ‘assembly lines’ that produces systems one after another in a systematic manner. Simply put, product family architecture (PFA) is at the heart of PFE, but it is not required by PLE.

A PFA is described as a hierarchical architecture that adopts optimised modules as building blocks from both the functional and structural perspectives [ZS06]. It becomes the higher level structure for a set of related products [DR08]. Each product
member can derive its own architecture from the PFA [DSB05].

However, after years of study on PFA, researchers start to realise that constructing a qualified PFA is not an easy task. Several new directions have been suggested. For example, some researches use code-level variability to create program families [Par01] (e.g., pure::variants and CIDE), whilst some other adopt a base model that only captures common features but ignores variations to model product families [TCY93] (e.g., XWeave and DeltaJava). These two methods cannot implement the whole product family with all valid members, thus they have to pay more attentions on the configuration process to ensure a proper configuration is selected [DMvdH01], which lead to the rise of PLE henceforth.

Figure 2.6 shows the basic workflow of PLE. Apparently, PLE is lucid enough to deal with a handful of product requirements, e.g., 10 products from a big family, or all products from a small family. Contrariwise, most of PLE approaches lack evidences that they are truly practical in the context of software industry, due to both developers and users are not aware of all variability in the product line. Consequently, an error might be detected very late, or modifying an artefact might not cover all relevant products.

By contrary, our approach is rather a PFE approach than a PLE approach. It starts
with an feature model including full set of variation points and constraint dependencies. Every feature is specified and modelled in isolation to ensure the safe composition of features without considering the implementation [AK09]. Accordingly, we model and implement components (according to a component model) mapped to features, and an architecture of the product family (also according to the component model) mapped to the feature model. The architecture is a composition of all the product variants in the family, which are in turn compose from components. Strictly speaking, we construct a PFA to define the architecture of the product family and the architecture of each product therein. As a result, product configuration is no longer necessary, as well as the further assembly. Instead we just ‘pick out’ all the products we want. Figure 2.7 shows the workflow of our approach.
Chapter 3

X-MAN and FX-MAN Component Model

“Square tenon does not fit the round mortise.”
— Chinese Proverb

3.1 Introduction

Component-based development (CBD), also known as component-based software engineering (CBSE), is an important branch of software engineering. With CBD, a software system can be designed and implemented by composing loosely coupled independent components, instead of a monolithic entity [HC01, Ld17].

CBD is widely used in software industry, due to it has advantages as follows:

- **Reusability** - A high-quality software component should be designed and implemented in such a way that many different applications can reuse it.

- **Replaceability** - Each component should be quickly replaced by a similar one. Therefore, a component-based system is easier to maintain by substituting the components no longer suitable.

- **Extensibility** - A software component can be further adjusted for additional functionality, or tailored for modified functionality.
3.2. COMPONENT

- **Encapsulation** - A software component interact with others by its interfaces only. Hence a component-based system has better security, due to the local details can be hidden.

- **Independence** - A high-quality software component should be designed for different environments and contexts. Its deployment does not affect the whole system in any way.

In this chapter, we will present (i) the foundational knowledge of CBD, such as component, component models and their life cycles, (ii) a new component model called X-MAN, and (iii) an extension of X-MAN, namely FX-MAN, which can model and implement enumerative variability.

### 3.2 Component

Although the general accepted definition of component has not been proposed [LW07], many researchers define a component from different perspectives:

1. Kruchten [Kru04]: “A nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.”

2. Szyperski et al. [SGM02]: “A software component is an unit of composition with contractually specified interfaces and explicit context dependencies only. An software component can be deployed independently and is subject to composition by third parties.”

3. Pryce and Crane [PC98]: “A component is a unit of distributed program structure that encapsulates its implementation behind a strict interface comprised of services provided by the component to other components in the system and services required by the component and implemented elsewhere.”

4. Fayad and Schmidt [FS97]: “Components are self-contained instances of abstract data types (ADTs) that can be plugged together to form complete applications.”
5. Councill and Heineman [CH01]: “A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.”

According to the component definitions presented above, we can describe a component as an independent, reusable and replaceable system unit that can only be accessed via its services (interfaces), or interact with other components through the connectors, as illustrated in Figure 3.1. We can deposit a created component into the repository, and later retrieve it for building systems. The usage of a repository in Figure 3.1 suggests that the construction of systems is essentially bottom-up (i.e., composition) rather than top-down (i.e., decomposition). Intuitively, the interaction and composition standard of a component is defined by its component model.

![Figure 3.1: Composing generic software components.](image)

We hereby categorise the existing component models into three groups according to what software elements are used to realise the components [Lau14]: (i) use objects as component (Figure 3.2(a)), (ii) use architectural units as component (Figure 3.2(b)), and (iii) use encapsulated components as component (Figure 3.2(c)).
3.2. COMPONENT

3.2.1 Category 1: Component based on Object

Most component-based systems developed by object-oriented programming languages define components as objects, because it is easier to understand and implement. An object resembles the generic component demonstrated in Figure 3.1, but there is a critical and subtle difference. A component defined as an object does not possess, or exhibit, any required service. As a result, the composition mechanism of this type of component relies on method delegation, i.e., direct method call, or message passing [LOW06]. Here we introduce several software components belonging to this category:

**JavaBeans**

A *JavaBean* is a reusable software component implemented in Java, which can be manipulated in visual with an application builder tool [Eng97]. Essentially, a JavaBean is a Java class that encapsulates multiple objects into a single one (the *bean*). Typically, a JavaBean has three characteristics that distinguish itself from a normal Java class:

1. A JavaBean defines and realises the *serializable* interface. Thus, the state of a bean (including its data) can be reliably saved (and restored) somewhere independent of the Java Platform and of the Java virtual machine (JVM).

2. A JavaBean defines *getter/setter methods*, which give the only access to its properties. With the application builder tool, the getters and setters can be generated automatically for the created fields in a JavaBean.

3. A JavaBean has a nullary constructor, i.e., the constructor does not accept any argument.

Thereby, the composition of JavaBeans is emerged from message passing through delegation of events [WCD+01].
Component Object Model (COM) is an object-oriented system introduced by Microsoft, which aims at creating reusable binary software components that can interact [GHL+98]. Therefore, a COM component is independent of its programming language, and independent of its implementation platform. Technically speaking, a COM component can provide more than one interface implemented in interface definition language (IDL). The mandatory interface, called *IUnknown*, allows calling code to get an implementation for a known interface, and implement the memory management mechanism. In conclusion, COM components can be composed by method calls via the interfaces [SMS99].

**OSGi**

OSGi, an acronym of *Open Services Gateway initiative¹*, is an open standards organisation, which publish a set of specifications that define a dynamic component model for Java [HPMS11]. Therefore, an application can be implemented as a composition of OSGi components, which are packaged in *bundles*. The communication between bundles, no matter locally or across the network, occurs via the *services*, which are specified by a Java interface. A bundle can detect new services, and register, remove, adapt services accordingly. Notably, OSGi bundles do not compose, but the services can be composed via direct method call [RVCA07].

### 3.2.2 Category 2: Component based on Architectural Unit

Some applications are designed and structured by software architectures. As the primary element of a software architecture, an architecture unit can be regarded as a component of the system, due to it also has two characteristics: (i) encapsulation and (ii) compositionality. An architecture unit encapsulates the computation unit and data, and the architecture units can communicate and interact. Similarly to a generic component, except that it uses *ports* instead of services. A port specifies an *interaction point* that an architecture unit can communicate with its environment, with other architecture units, or with its internal part [ADG98]. As a result, the composition mechanism of this type of component relies on indirect message passing [LOW06]. Here we introduce several software components belonging to this category:

¹https://www.osgi.org/
ArchJava

ArchJava is a Java extension for architecture, which seamlessly unifies architectural structure and implementation [ACN02b]. In ArchJava, a component is able to communicate with other components at the same level. The communication is only allowed via the explicitly declared ports, i.e., traditional method call is no longer needed in ArchJava. Thereby, a port indicates that a logical communication channel can be established between this component and others. On the other hand, a port declares different kinds of methods inside the computation unit. For instance, a provided method is available to be invoked by any component connected to the port, whereas a required method is provided by other components connected to the port. In conclusion, in ArchJava, a composite component is constituted by a connection of ports of subcomponents [ACN02a].

SOFA

SOFA, an acronym of SOFtware Appliances\(^2\), is a component system that supports hierarchically composed components [BHP06]. A component in SOFA is architecture unit specified by a frame and an architecture. The former defines the both provided and required interfaces, and other properties of the component, while the latter determine the component structure. Consequently, in SOFA, a composite component is composed by subcomponents through connectors. The connector supports several communication styles, such as procedure call, messaging, streaming and blackboard [BP03].

Koala

In Koala, a component is an architecture unit without implementation [VOVDLKM00]. That means, the components are defined in an ADL-like language, which are only used to describe the software architecture. The components can connect to each other via their interfaces, or compose method calls through connectors. One of distinctive feature of Koala is that the diversity can be handled by connectors. Therefore, Koala is a promising approach for modelling software product families in ad hoc domain [ASM03].

\(^2\)http://sofa.ow2.org/index.html
3.2.3 Category 3: Component based on Encapsulated Component

A component belonging to this category encapsulates the computation unit with only provided services. Thus, the encapsulated component does not possess functional dependencies with other components. As a result, such a component can be replaced easily. Moreover, in a project, encapsulated components can be independently developed by different groups. Finally, these created components can be composed by *exogenous connectors* [LVW05], which are able to coordinate the control flow between components. So far, the component defined by X-MAN component model is the only one belonging to this category [HKW+12, LT12]. More details of X-MAN will be discussed in next section.

3.3 Component Model

Component model has been proposed to describe the individual components, and define the specific communication and interaction between them, and a set of composition standards for system construction [LH02]. Thereby, a well-defined component model must operate on two levels [CH01]:

1. A component model defines the semantics and syntax of components, e.g., the hierarchy of components, the static relationship between components, the representation and responsibilities, etc. Hence, with component model, developers can understand how to construct an individual component.

2. A component model defines the composition mechanism in order to establish assembled or integrated connections between components.

It is worth noting that one of the major differences between component and other forms of packaged software (e.g., function, library) is that a component is always compliant to a component model [BBB+00]. Therefore, it becomes obvious that understanding what desirable characteristics a component should have directly provides the requisite support for defining a component model. We hereby conclude our study of past researches with a resulting set of desiderata [HC01, SGM02, Mey03, BDH+98]:

- *Components should be pre-exist before constructing the system.* Thus, we should use a repository to deposit components in design phase, and retrieve components in deployment phase.
Components should be developed in an independent manner. Thereby, in design phase, we should use a builder tool to construct new components. The builder should support for depositing and retrieving components.

Components should be deployed in an independent manner. Therefore, in deployment phase, we should use an assembler tool to retrieve components from a repository, and if necessary, compile them into binary code, and finally assemble them into a system.

Components should be able to be copied and instantiated. Hence, we can copy a component in design phase, and create one or more instances of a component in the deployment phase. Notably, normally when a component is retrieved from the repository, we actually create an instance of the component.

Components should be composable. That means, we can generate a composite component by composing subcomponents in design phase and/or deployment phase.

Over the last two decades, many component models have been put forwards, whereas some of them are adopted in software industry [VCdA+16]. However, most component models cannot produce a component whose life cycle meets the desiderata listed above. Figure 3.3 shows five categories that cover all major existing component models.

Category 1 shows that some component models do not support repository in design phase. As a direct consequence, the components cannot be retrieved from a repository and assembled into systems in the deployment phase. But we can compose such components in the design time, and instantiate the composite component in the run-time phase. This category includes all simple Acme-like ADLs [GMW00] and POJOs, such as ArchJava [AGST04] and UML 2.0 [PP05].

Category 2 describes that new components can be deposited into a repository, but cannot be retrieved in both design and deployment phases. Compositions of components are possible, but they cannot be retrieved neither. Thus, the component instance is created directly from the repository in the run-time phase. This category includes EJB [MSD03], OSGi [HPMS11], .Net [ES14], COM [GHL+98], CCM [Bar01] and Fractal [BCL+06].

Category 3 expresses that new components can be deposited into a repository in the design phase, and they can be retrieved from the repository only in the deployment
CHAPTER 3. X-MAN AND FX-MAN COMPONENT MODEL

<table>
<thead>
<tr>
<th>Component Models</th>
<th>Builder</th>
<th>Repository</th>
<th>Assembler</th>
<th>RTE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Category 1:</strong> Design without Repository</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Acme−like ADLs, UML2.0, PECOS)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Category 2:</strong> Design with Deposit−only Repository</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>(EJB, OSGi, Fractal, COM, .NET, CCM)</td>
<td></td>
<td></td>
<td>InsA</td>
<td>InsB</td>
</tr>
<tr>
<td><strong>Category 3:</strong> Deployment with Repository</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>(JavaBeans, Web Services)</td>
<td></td>
<td></td>
<td>InsA</td>
<td>InsB</td>
</tr>
<tr>
<td><strong>Category 4:</strong> Design with Repository</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>(Koala, SOFA, KobrA, ProCom)</td>
<td></td>
<td></td>
<td>InsA</td>
<td>InsB</td>
</tr>
<tr>
<td><strong>Category 5:</strong> Design and Deploy with Repository</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>(X-MAN, FX-MAN)</td>
<td></td>
<td></td>
<td>InsA</td>
<td>InsB</td>
</tr>
</tbody>
</table>

Figure 3.3: Component life cycle.

...
3.3. COMPONENT MODEL

ProCom [BCC+10] and so forth.

Category 5 illustrates that new components, and compositions of components, can be deposited into or retrieved from a repository in the design phase. Moreover, they can be retrieved from the repository in the deployment phase. Composition is also possible in the deployment phase. As a result, components, including composite ones, are same as the deployed components. So far, X-MAN [HKW+12, LT12] is the only member in this category.

3.3.1 X-MAN Component Model

X-MAN component model describes the semantic and syntax of encapsulated components, and defines an exogenous composition between these components [HKW+12, Ld17]. An X-MAN component may be atomic (Figure 3.4(a)) or composite (Figure 3.4(b)).

![Figure 3.4: X-MAN: Component models.](attachment:image)

An atomic component encapsulates a set of methods into a provided service (an input-output function) with a purpose that different components can access. Therefore, the service is the only portion visible and accessible from the outside of a component. A composite component consists of subcomponents (atomic or composite) composed by exogenous composition connectors, which coordinate control flows between subcomponents. Each kind of composition connectors has a fixed semantic that can form a statement. Notably, X-MAN component model constructs algebraic and hierarchical components, i.e., the composition of X-MAN components is a bigger X-MAN component, while the composition of services (of X-MAN components) is another sole service (of a bigger X-MAN component).

As Figure 3.5 shows, a sequencer SEQ defines sequencing, while a selector SEL defines branching. In Figure 3.5(a), composite component \( C_0 \) is composed by subcomponents \( C_1, C_2, \ldots, C_n \) with a sequencer. Likewise, \( S_0 \), as the service of \( C_0 \), is
the composition of services $S_1$, $S_2$, ..., $S_n$, each of which is exposed by a subcomponent. From a composite component, we can derive a statechart\(^3\). The right part of Figure 3.5(a) shows $S_1$, $S_2$, ..., $S_n$ are executed sequentially when $S_0$ is invoked. Similarly, in Figure 3.5(b), a selector selects one service to call, depending on which selection condition is satisfied.

Except for the composition connectors, X-MAN applies *adaptors* to an individual component to adapt the control it receives. As Figure 3.6 shows, a *guard* adaptor $GUD$ allows control to reach a component only if a condition is satisfied, while a *loop* adaptor $LOP$ repeats control to a component a fixed number of times. In Figure 3.6(a), $C_0$ is composed by a guard adaptor and $C_1$. The statechart of $S_0$ shows that service $S_1$ in component $C_1$ is only invoked when the guard condition is satisfied. Similarly, in Figure 3.6(b), the loop adaptor keeps calling $S_1$ until the condition is no longer fulfilled.

Additionally, in order to handle the variability caused by ‘or’ variation generator (see next section), X-MAN defines an *aggregator* connector to aggregate subcompo-

---

\(^3\)In this thesis, we use *STATEMATE* statechart [HN96], as FODA does.
nents. That means, according to the arguments, the aggregator connector allows control to reach more than one subcomponents if the corresponding condition is met. In Figure 3.7, component $C_0$ is aggregated by its subcomponents $C_1$, $C_2$, ..., $C_n$. Service $S_0$ parses the arguments, and iteratively call the services in subcomponents. Thus, the statement of aggregator is considered as a mix of selection and iteration, as depicted by the statechart on the right. For example, if the input of $S_0$ is an array $[c_2, c_1, c_n]$, $S_2$, $S_n$ and $S_1$ will be called sequentially.

In the design phase, an atomic component is designed and created from scratch, while a composite component is designed and constructed by components retrieved from a repository. The key question at this stage is granularity: how many methods should be put in a component? The design problem should be considered with reuse. Ideally, every component should be designed as a potential reusable unit. Atomic components are the smallest reusable unit that can be reused in composite component, while composite components can be reused in bigger ones.

Figure 3.8 shows the process of composing X-MAN components. $C_1$, $C_2$ and $C_3$ are atomic components that expose services $S_1$, $S_2$ and $S_3$, respectively. $S_1$ and $S_3$ invoke multiple methods, whereas $S_2$ only invokes one. After they are created, we deposit these three components into the repository (Step 1). Then, in order to construct composite components, we retrieve the atomic components from the repository and create four instances of them: $C_{1,1}$ and $C_{1,2}$ instantiated from $C_1$, $C_{2,1}$ instantiated from $C_2$, and $C_{3,1}$ instantiated from $C_3$ (Step 2). After that, we can use composition connectors to compose the component instances. The resulting components are $C_4$ and $C_5$, each of which exposes a provided service. Likewise, the composite components are deposited into the repository (Step 3). Finally, in the deployment phase, we retrieve and instantiate $C_4$ and $C_5$ (Step 4), with the aim of further development, such as constructing a product family. Notably, in this example, we reuse the component $C_1$ to model $C_4$ and $C_5$. Apparently, service is essentially an input/output function of a component’s
methods. It is regarded as the entry point into the component, which must be called at the start of the component.

The exogenous connectors and adaptors coordinate the control flow, while the \textit{data channels} coordinate the data flow. After a composite component is structured, we need to add data channels between X-MAN components in order to define the direction of each data flow. Such a channel links input and output of services. Figure 3.9 demonstrate two types of data channels within a composite component \(C_0\): \textit{horizontal data routing} and \textit{vertical data routing} [LT12]. The former is between two individual components, such as the two subcomponents \(C_1\) and \(C_2\), which indicates a component passes the outcome data to another (\(o_1 \rightarrow i_2\)). The latter is data propagation between the services of a composite component and its subcomponents, such as \(C_0\) and \(C_1/C_2\), which illustrates the data received by the composite component is passed to the first invoked subcomponent (\(i_0 \rightarrow i_1\)), whereas the outcome data of last invoked subcomponent becomes the output of the composite component (\(o_2 \rightarrow o_0\)).

So far, it becomes obvious that the communication between X-MAN components only allows arguments to pass directly, with the purpose for invocation or a return of control [Kin84]. In most of the cases, the passed data is very simple, e.g., variable or array, which leads to \textit{data couplings} between X-MAN components. Data coupling is
3.3. COMPONENT MODEL

Figure 3.9: X-MAN: Data channels.

de facto loosest and best coupling in software engineering, considering the communication inside a system is inevitable [MM88]. However, an X-MAN component may pass a data structure (e.g., object) as parameter for complex systems, which results in the second loosest coupling, as known as stamp coupling [HM95]. The loose coupling is a very important character of X-MAN component model. We will discuss the details in the context of family-based testing (see Chapter 5).

3.3.2 FX-MAN Component Model

FX-MAN component model is an extension of X-MAN component model. A distinguishing property of FX-MAN is the composition of variants of components. The variability is handled by explicitly realising full set of variation points. The overall structure of FX-MAN is demonstrated in Figure 3.10.

As the basic building blocks of FX-MAN, X-MAN components, no matter atomic or composite, are at the lowest level of composition. In PFE, they are mapped to the leaf features specified in the feature model. The next level of composition offers variation generators (Figure 3.11), which model variation points. In Figure 2.3, we introduce the three types of variation points defined in the feature model, each of them specifies multiple variants (where Fs denote features): (i) 2 for optional($F$); (ii) $n$ for alternative($F_1,...,F_n$); (iii) $(2^n - 1)$ for or($F_1,...,F_n$). Accordingly, a variation generator must generate multiple variants: it takes (a set of) sets of components as input and produce (a set of) permuted sets of components, i.e., variants. We have implemented variation generators for the full range of standard variation points, viz. optional, alternative and or (respectively OPT, ALT and OR in Figure 3.11). Variation generators are thus also algebraic and hierarchical. Whilst optional and alternative are
trivial to implement, the \textit{or} variation point requires the generation of the power set of its input set (of component sets), and hence the aggregation of component sets. We use the \textit{AGG} connector (Figure 3.7) for this aggregation.

\begin{align*}
\{\{F_1\}, \emptyset\} & \quad \{\{F_2\}, \{F_3\}\} & \quad \{\{F_4\}, \{F_5\}, \{F_4,F_5\}\}
\end{align*}

(a) OPT \quad (b) ALT \quad (c) OR

So at the next level of composition in FX-MAN component model (Figure 3.10), family composition takes place by means of family composition operators, also defined as connectors. A family composition operator is applied to multiple input component sets to yield a set of product variants, i.e., a (sub)family of products. These operators are defined in terms of the component composition operators: a \textit{family composition connector} forms the Cartesian product of its input sets, and composes components in each element of the Cartesian product using the corresponding component composition...
connector. Notably, which family composition operator to use is a design choice in order to satisfy the requirements.

For example, Figure 3.12 shows a family composed by two component sets and a family composition connector $F$-$SEQ$, whereas Figure 3.13 shows a family constructed via an $F$-$SEL$. $F$-$SEQ$ and $F$-$SEL$ apply the corresponding X-MAN composition connector $SEQ$ and $SEL$ to components in each element of the Cartesian product, respectively. The statechart on the right is derived from the FX-MAN component, which expresses the behaviour of the family, and the behaviours of the products thereof. Notably, every component set refers to a state set, i.e., a maximal set of states (e.g., $S_{12}$ and $S_{34}$) that the family can be in, or in other words, every state in a set that a product can be in. Hence, The family statechart can be considered as a superposition of all products’ statecharts. This property is very useful in family-based testing (see Chapter 5).

FX-MAN also applies family adaptors to an individual component set to adapt the control received by its member components. Figure 3.14(a) shows that the family adaptor $F$-$GUD$ applies the corresponding X-MAN adaptor $GUD$ to components in the set, and the corresponding if-statement in the family statechart and every product thereof. Figure 3.14(b) shows that the family adaptor $F$-$LOP$ provides the corresponding X-MAN adaptor $LOP$, and the while-loop in the family statechart and every product thereof.
Figure 3.13: FX-MAN: Family Selector.

It is worth noting that the result of a family composition is also a family, i.e., a component set, so this level of composition is also algebraic, which means it can be further composed into a family of families. In other words, an FX-MAN component can contain nested variation generators, family composition connectors and family adaptors.

In an FX-MAN component, every family connector results in a (sub-)family, in which all members are enumerated. However, in many scenarios, not everyone is valid or wanted. Therefore, we design a special construct, called family filter, to filter out some results. For each family composition connector, we put a family filter, which aims to parse constraints as strings, detect invalid component sets produced by the family connector, and remove them immediately. The format of constraints is generally defined as:

\[
\text{SOURCE RULES}[\text{requires}|\text{excludes}] \text{ TARGET} \quad (3.1)
\]

It suggests that a component must exist with another, or two components cannot co-exist. Moreover, a family filter is used to solving the feature interaction problem by defining the interaction rules. This matter will be discussed in Chapter 4.4.

Finally, after an FX-MAN component is structured, we need to add data channels to coordinate the data flows, just like what we do for X-MAN composite components.
3.4 Summary

As a matter of fact, either X-MAN or FX-MAN component model is not a new concept. Similar encapsulated components have been proposed before [LT12, dCTL^16, QL17]. However, during the course of our study, we found out that the old version of X-MAN and FX-MAN component model are not suitable for PFE. For instance, the lack of service makes the code generation difficult, whereas multiple services in one component result in problematic compositionality. Hence, in this chapter, we present new versions of X-MAN and FX-MAN, which aims at not only construct enumerative variability, but also define one-to-one mappings between features and components, as well as variation points and variation generators. More details will be discussed in next chapter.
Chapter 4

A Feature-oriented Component-based Approach to Constructing Whole Software Product Families

“Give a man a fish, and you feed him for a day. Teach a man to fish, and you feed him for a lifetime.”

— Chinese Proverb

4.1 Introduction

In order to construct whole product families, our approach aims at modelling and implementing enumerative variability as they specified in the feature model in a feature-oriented, component-based manner. In the previous section, we present the X-MAN and FX-MAN component model, and the feasibility of using them to establish an architecture for a product family. In this chapter, we demonstrate, step by step, how to construct a PFA by using FX-MAN component within the PFE framework (Figure 2.2). Furthermore, we will show how to address the critical problems occurred in PLE, i.e., the feature interaction problem and the crosscutting concern.
4.2 Family Construction

In Chapter 2.3, we use a vending machines family to express the enumerative variability specified in the problem space, as well as the parametric variability realised in the solution space. In this chapter, we will extend the vending machine family by adding an optional feature, namely ‘CANCEL’, and accordingly demonstrate the complete process of constructing a product family step-by-step from scratch by using our feature-oriented component-based approach. During the family construction, we will show how to address feature interactions and crosscutting concerns.

4.2.1 Constructing Feature Model

As a matter of fact, a number of variability model have been proposed for domain analysis. However, not all of them are suitable for our approach. For example, FODA feature model does not define the ‘or’ variation point, so it is impossible to select a subset of a feature group [KCH+90]. FeatuRSEB feature model does not define any cross-tree constraint, hence it cannot present complex feature relations [GFd98]. FORM feature model not only defines variation points, but also defines compositions and generalisations for father-child relationships, which restricts the further realisation of product lines [KKL+98]. Cardinality-based feature model uses cardinalities to replace variation points, which results in an open-world domain [CHE04]. PLUSS approach binds the feature model with use cases, which only facilitates use case driven development [EBB05]. Benavides et al. feature model includes feature attributes in the feature model, which may be useful, but definitely increases the task difficulty in product family implementation [BMAC05]. KobrA uses textual decision model to describe variabilities in the problem space, which causes the lack of concrete templates or standards to represent the framework design [ABM00]. Orthogonal Variability Model (OVM) does emphasise the variation points, but neglects the importance of mandatory features [MP14]. Therefore, our feature model must satisfy several conditions:

1. The feature model must be easy to understand, for both developers and customers.

2. The feature model must cover all product configurations, i.e., every combination of features can be instantiated from the feature model.
3. The feature model must support complex feature relations, such as cross-tree constraints, in order to filter out invalid products.

4. Every feature in the feature model must represent a logical unit of behaviour specified by a set of functional requirements.

Based on the conditions, we consider that Figure 4.4 is an appropriate feature model. The notations are comprehensible, and apply valid product variants, while every feature has its own functionalities. The same kind of feature model is also adopted by some trending commercial tools, such as pure::variants and FeatureIDE. However, so far, our feature model does not support non-functional requirements, customer-specific functions and feature interactions. These properties can be implemented in the application engineering, whereas we decide to construct a feature model purely used for family-level construction.

In order to construct a correct PFA, we must guarantee the feature mode is well-formed. Because our PFE approach is feature-oriented, even a tiny mistake in the feature model can lead to huge errors in domain design and implementation. There are three kinds of mistakes commonly occurred when constructing feature models [vdML04]:

- **Redundancy.** A feature model contains redundancy, if at least one semantic information is modelled in a multiple way. Figure 4.1 shows 4 examples of redundancy:

(a) $F_2$ is a mandatory feature, which must be selected, regardless of the constraints.

(b) $F_1$ and $F_2$ are alternative features, thus they anyway exclude each other.

(c) A mandatory feature, whose predecessors are all mandatory.

(d) An optional feature, whose descendants are all optional.
4.2. FAMILY CONSTRUCTION

Technically, the redundancy issue does not effect the correctness of domain implementation, but it does costs extra time and effort during the family construction. We should prevent this problem, unless there is a business reason.

- **Anomalies.** A feature model contains anomalies, if the choice of possible product configurations is restricted by senseless modelled information.

![Anomalies Diagram](image)

Figure 4.2: Anomalies.

Figure 4.2 shows 4 examples of anomalies:

(a) $F_1$ is optional, but every product has to include it.

(b) $F_1$ and $F_2$ should be able to coexist, but they cannot because of the constraint.

(c) $F_2$ becomes mandatory, while $F_1$ becomes a *dead feature*.

(d) $F_2$ always exists, while $F_1$ becomes optional.

Anomalies are more severe, due to they may lead to erroneous product lines or PFAs. The semantic of variation points are changed, as well as the family behaviour.

- **Inconsistency.** A feature model contains inconsistency, if at least one contradiction can never be fulfilled.

Figure 4.3 shows 4 examples of inconsistency:

(a) $F_1$ excludes $F_2$, while $F_2$ must be included in every product.

(b) $F_1$ excludes $F_2$, while they cannot be selected at the same time.

(c) If both $F_1$ and $F_2$ are unselected, mandatory feature $F_0$ is void.

(d) $F_3$ requires $F_2$, but excludes $F_1$, while both $F_1$ and $F_2$ are mandatory.

\footnote{A dead feature cannot appear in any product in the family [TBK09]}
Inconsistency is not a tolerable issue in PFE. It sabotages the validation of product family, and hence the correctness and quality of any product within cannot be guaranteed any more.

Figure 4.4 shows a well-formed feature model without any aforementioned problem. It describes the enumerative variability in the problem space of vending machine family. Notably, we add an optional feature \textsc{Cancel}, which will be defined as a cross-cutting feature in Chapter 4.3. The feature model enumerates all the valid combinations of features, i.e., all the valid product variants. In Figure 2.3, there are a total of 30 valid variants, including but not limited to: ‘\textsc{Coffee, Cancel, Cash, Change}’, ‘\textsc{Tea, Cancel, Cash, Change}’, ‘\textsc{Coffee, Tea, Cancel, Cash, Change}’, but not ‘\textsc{Coffee, Cancel, Gift Voucher, Change}’, due to the ‘\textsc{Gift Voucher excludes Change}’ constraint.

According to the requirement, in this family, a vending machine provides tea or coffee, or both. If the order is not cancelled in the next step, a vending machine supports one of three payment methods: cash, gift voucher, or card. Some machines with cash payment gives change, whereas some do not. If a machine accepts the gift vouchers, it is not designed for change return. On the other hand, if a machine accepts cards,
it may offer a cash-back facility.

### 4.2.2 Modelling and Implementing Features as Components

The first step is creating X-MAN components to model and implement leaf features. In a product variant, abstract features aggregate behaviour corresponding to leaf features, so it is sufficient to have a component for every leaf feature, i.e., 1-to-1 mapping between leaf features and components. Such a component may be atomic or composite, depending on the complexity of the corresponding feature.

For example, as Figure 4.5 shows, Coffee is a composite component mapped onto the leaf feature \textit{Coffee} in the feature model. The component is composed by three atomic components: Latte, Mocha and Cream. On the right of Figure 4.5, the statechart expresses the behaviour of \textit{Coffee}, which is realised by two composition connectors (\textit{sequencer} and \textit{selector}) and an adaptor (\textit{guard}). Accordingly, if a customer want coffee, he can choose either mocha or latte, with or without cream on top.

![Statechart for Coffee](image)

Figure 4.5: Vending machines: Component Coffee for feature COFFEE.

### 4.2.3 Modelling and Implementing Variation Points as Variation Generators

The second step is using variation generators to model and implement variation points specified in the feature model. In the vending machine example, the variation generators are applied to the X-MAN components created before, as shown in Figure 4.6\textsuperscript{2}. \textbf{OR}(Coffee, Tea) generates the power set \{\{Coffee\}, \{Tea\}, \{Coffee, Tea\}\}. This contains the aggregate \{Coffee, Tea\}, which is implemented using the aggregation connector \textit{AGG}. \textbf{OPT}(Cancel), \textbf{ALT}(Cash, Gift, Card) and \textbf{OPT}(Change) generate \{\emptyset, \{Cancel\}\}, \{\{Cash\}, \{Gift\}, \{Card\}\} and \{\emptyset, \{Change\}\}, respectively.

\textsuperscript{2}We will abbreviate component GiftVoucher to Gift from now on.
In each (sub-)family, a set of component sets generated by a variation generator provides a set of feature sets in product variants. For example, in Figure 4.6, \(\text{OR}(\text{Coffee, Tea})\) provides: (i) \{Coffee\} in vending machines that sell only \textit{Coffee}; (ii) \{Tea\} in vending machines that sell only \textit{Tea}; and (iii) \{Coffee, Tea\} in vending machines that sell both \textit{Coffee} and \textit{Tea}. At the next level of variation points, each application of a variation generator will result in larger component sets that provide larger feature sets in product variants. Eventually, beyond levels with variation points, variation generators will have generated full feature sets of all possible final product variants. At this stage, however, features in feature sets remain separate, as separate components. These features/components need to be composed appropriately to yield the final product variants. This composition is performed at the next level up by family composition connectors and adaptors.

### 4.2.4 Applying Family Connectors and Adaptors to Generate Product Variants and Product Families

The third step aims at family composition. Which family composition operator to use is a design choice in order to satisfy the requirements. For instance, the process of purchase from a vending machine starts with a selection of beverage, then the payment for the order, and finally, if possible, ends with the change. Thus, we can use a simple family sequencer to compose the variants of components generated before (Figure 4.6). The PFA of the vending machine family is structured as Figure 4.7 shows.

Notably, there is a feature constraint in the vending machine family: if a vending
machine receives \textit{Gift Voucher}, it does not give the \textit{Change}. Simply put, components \textit{Gift} and \textit{Change} cannot coexist. The constraint described as

\begin{align}
\text{SOURCE(‘Gift’) RULES(‘excludes’) TARGET(‘Change’)}
\end{align}

is defined in the family filter on top of the family sequencer, which can rule out the invalid products at the moment after the family composition.

After all, we must add data channels between components, connectors and adapters. For the simplicity, we omit the data channels in Figure 4.7.

\section*{4.3 Crosscutting Concern}

In software engineering, we use the term \textit{‘concern’} to describe a part of the system divided based on the functionality. Generally, a concern specifies single functionality for primary requirements. But sometimes concerns demonstrate more functionalities for secondary requirements by affecting other concerns. Such a concern is called a \textit{crosscutting concern}. In the context of PFE, a product family is decomposed into features, each of which represents specific functionality. The crosscutting concerns have been frequently observed occurring between features [ALS08, KAB07]. Thus, a feature that has an effect on others is called a \textit{crosscutting feature}. For instance, in the vending machines family, if the new optional feature \textit{Cancel} allows the order to be cancelled at any time in the purchase process, it becomes a crosscutting feature.

There are two types of crosscutting concerns in PFE. Different features crosscutting
in different positions is called heterogeneous crosscutting concern, whereas the same feature crosscutting in different positions is called homogeneous crosscutting concern [LH06]. The former has been addressed by contemporary FOSD approaches by encapsulating various class fragments into different refinements [KA13]. However, using the same fashion for homogeneous crosscutting concerns anyway leads to tangling issues within a code base. If a feature effects a great deal of existing implementation units, it causes excessive method shadowing [ALRS06]. In other words, it is infeasible to expose extension points at all positions, and henceforth override tons of methods, in which most of additive code is redundant [ALRS05, KAO11].

Our FX-MAN component model provides a chance to address the crosscutting concerns [QL18b]. Heterogeneous crosscutting concern is no longer existing, due to the product family we construct does not have a base model that provides multiple positions for crosscutting. Homogeneous crosscutting concern can be solved by applying family composition connectors to proper positions in the PFA. Therefore the control flow can be coordinated and certain components can be invoked multiple times in a system. We will use a simple example to elaborate.

Figure 4.8 depicts a small family that only contains two products. The simple product (Figure 4.9(a)) is composed by two components, called SignIn and SignOut, which ask the user sign in using his credential, and subsequently sign out the system. Another product (Figure 4.9(b)) has an additional Syslog, which records every system event. That means, the behaviour of this product can be described as: (1) user signs in, (2) write log file, (3) user signs out and (4) write log file. Clearly, the Syslog is an implementation of a crosscutting feature.

The F-LOP starts a loop by receiving a ‘start’ message, whereas the F-SEQ illustrates that Syslog must be invoked after either SignIn or SignOut is executed. The F-SEL controls which one between SignIn and SignOut should be invoked. Therefore, in the first iteration, the argument of F-SEL is “sign in”, which invokes SignIn, and then Syslog is invoked. In the second iteration, because the argument of F-SEL becomes “sign out” (passed from SignIn in the first iteration), SignOut is invoked this time. After that, Syslog is executed again. The loop is terminated, due to the SignOut outputs a ‘stop’ message and sends it to F-LOP.

The correction of family behaviour can be verified in its products, which are formed as X-MAN components, i.e., Figure 4.9(a) and 4.9(b). Thus, we can generate state-charts accordingly. By comparison, it becomes obvious that SignOut is invoked twice

---

3Red arrows represent data flows.
4.4 Feature Interaction

During the construction of product families, the behaviour of features that interacts with one another needs particular attention. This is known as feature interaction: “A feature interaction is a situation in which two or more features exhibit unexpected behaviour that does not occur when the features are used in isolation” [AK09]. For example, the vending machine family (Figure 2.3) has an optional feature called Change, which counts change back to a customer if the machine accepts Cash. However, if the machine accepts Card instead of Cash, then the Change becomes a cash-back facility. That means, the behaviour of Change is changed in the presence of Card. In conclusion, a feature interaction takes place between Card and Change.
Thus, the feature interaction actually consists of two challenges: (i) detecting feature interactions at design phase, and (ii) solving feature interactions at implementation level. The second challenge is also defined as optional feature problem [KAR+09]. As the name implies, it usually occurs when two optional (non-mandatory) features seem independent at domain level, i.e., in the feature model, but somehow they interact at the implementation level, i.e., require additional or modified code for their combination. In this section, we only discuss how to address the second challenge when constructing a PFA with components.

The problem of feature interaction has lasted for decades, but no perfect solution has been found yet. At present, there are three major strategies [SSdCMdA18]: (1) conditional compilation, (2) micromodularisation, and (3) multiple implementations per feature.

Conditional compilation implements feature interactions by the same way that annotative approaches implement variable features, but conditional compilation uses nested directives (e.g., #ifdef $A$ and $B$) to annotate additional code fragments related to the interactions [KA08, MP02, SFR08]. Unfortunately, this strategy results in an exponential growth of the annotations and severe complexity of the code base, which limits its usage in PLE.
4.4. FEATURE INTERACTION

Micromodularisation, as the name indicates, aims at modularising feature interactions by extracting the additional code into small separate modules, namely derivative modules [KAO11]. The derivatives are assembled in conjunction with core modules to generate products with feature interactions, as shown in Figure 4.11(a). Therefore, this strategy is also known as refactoring derivatives [LBL06]. Although micromodularisation is widely used by FOSD approaches [LLHE17, TC13, LBN05], it is not flawless. For example, if too many features are involved in the interactions, the size of derivatives can be very small, whilst the number of derivatives can be huge, which requires massive effort to design and manage. In addition, micromodularisation is not capable to handle feature interactions that only remove methods, or requires completely different functionalities [AK09].

By comparison with the other two, multiple implementations per feature is a straightforward and flexible strategy. The general idea is to implement different domain artefacts for a feature. The choice of these implementations depends on the selection of other features. This strategy enables a feature interaction to modify or remove existing functionality, as well as add new one. For example, in Figure 4.11(b), module $A$ and $A'$ can be completely different, while they both are implementations of the same feature. $A'$ only exists when feature interaction occurs. This strategy is widely adopted by transformational approaches [SBB+10, SD10]. On the other hand, it has a defect

Figure 4.10: Vending machines: Product family architecture (with crosscutting feature).
that causes code replication. If a product family is implemented without a systematic reuse mechanism, it can take countless effort to realise frequently occurred feature interactions.

By weighing costs and benefits of the three strategies, we choose multiple implementations per feature to address feature interactions in our PFA. In fact, it does not seem overly cumbersome, because our approach is based on an algebraic and hierarchical component model, in which components can be easily reused and tailored at any granularity level [Ld17, dCTL+16], especially with the aid of our visualised tool [QL18a].

Now, we use an example to demonstrate how to solve the feature interactions. Figure 4.12 shows an family of ‘HelloWorld’. Both Hello and World are optional components, and mapped onto optional feature HELLO and WORLD, respectively. These two components have similar functionality, which is to print their names, i.e., ‘Hello’ and ‘World’. Now, we assume that feature HELLO and WORLD interact in a manner described below.
4.4. FEATURE INTERACTION

Adding functionality

When feature HELLO and WORLD are both selected, the system prints a phrase “Hello World”, with a space in the middle. Thus, printing the space is an extra behaviour needed to be realised.

```
print("Hello ");
print("World");
```

Figure 4.13: Hello world: “Hello World”.

Figure 4.13 provides one of the solutions. We implement a new component, namely HelloSpace, for feature HELLO. This component is constructed by reusing original component Hello, and a newly created one called Space that can print a space. As a result, HelloSpace prints “Hello ” (a space appended to ‘Hello”). Then we establish the interaction rule in the nearest family filter:

$$Hello \rightarrow HelloSpace, \text{World} \rightarrow |$$  \hspace{1cm} (4.2)

The rule defines that if Hello and World are both detected in the composition composed by an F-SEQ, HelloSpace replaces Hello automatically when the product is derived, whilst World has no substitution.

Removing and modifying functionality

When feature HELLO and WORLD are both selected, the system prints a phrase “Hi World”, instead of “Hello World”. In other words, the second implementation of feature HELLO is printing out the word “Hi” and a space.

Through the analysis and comparison of the behaviours of HELLO, we notice that the letter ‘H’ is printed first anyway. Thus, it is a common functionality that can be reused. Accordingly, we design the Hello as a composition of component H (prints ‘H’) and ello (prints “ello”). Subsequently, component H is reused to compose the new component HiSpace, which takes the place of component Hello whenever the
CHAPTER 4. AN APPROACH TO CONSTRUCTING WHOLE FAMILIES

print("Hi ");
print("World");

Figure 4.14: Hello world: “Hi World”.

following interaction rule is satisfied:

\[ \text{Hello} \rightarrow \text{HiSpace}, \text{World} \rightarrow | \quad (4.3) \]

Changing to completely different functionality

When feature HELLO and WORLD are both selected, the system prints a French phrase “Bonjour Monde”. This time, there is no reusable part, so we have to create complete new components Bonjour and Monde for feature HELLO and WORLD, respectively.

The interaction rule is stated as following:

\[ \text{Hello} \rightarrow \text{Bonjour}, \text{World} \rightarrow \text{Monde} \quad (4.4) \]

The final product family is illustrated in Figure 4.15.

Figure 4.15: Hello world: “Bonjour Monde”.
4.5 Summary

Figure 4.16 shows the full PFA of vending machines family. This example is not very large, i.e., consists of 30 products in total, but it is sufficient to demonstrate our feature-oriented component-based approach. All valid variants and products are composed while the variation generators and family composition connectors are applied. In other words, we successfully construct the enumerative variability in the solution space (i.e., PFA), as they are specified in the problem space (i.e., feature model). Henceforth, we do not need to assemble a product on the basis of its configuration, instead we directly derive any numbers of vending machine system from the whole family. By comparison with traditional PLE approaches, our approach is a lot more efficient to produce massive amount of systems, and equally efficient to produce a few.

Figure 4.16: Vending machines: Product family architecture (with crosscutting concern and feature interaction).
Chapter 5

Testing Software Product Families
Constructed by Enumerative Variability

“Real gold does not fear the test of fire.”

— Chinese Proverb

5.1 Introduction

Testing is the first step in investigating and evaluating the quality of software [Kan06]. It spots the errors and flaws occurred in the development phase, and ensures that the system’s performance satisfies the customer’s requirement. Thereby, having a testing is imperative in software engineering, no matter what kind of development paradigm is used. As a new branch of software engineering, PFE needs appropriate testing models and techniques.

According to the PLE life cycle, early work on testing software product lines put forward the W-model [JHQJ08, LKL12], as depicted in Figure 5.1.

Ideally, for every stage in domain and application engineering, a related testing should be performed. Similar to the traditional single system testing, the software product line testing starts from bottom to top crossing all granularity levels. The domain unit testing focus on the basic building blocks of family models, i.e., the elementary units of feature implementations, such as low-level components and code fragment.
5.1. INTRODUCTION

Figure 5.1: The W-model for software product line testing.

The domain integration testing targets on the integration of all possible combinations of all variations. At last, the domain system testing, also known as domain functional testing [LUV09] or domain conformance testing [MVDH03], is used to examine the compliance of all software products in the family with their functional requirements.

On the other hand, due to the software products assembled by the product line are independent executable systems, the application engineering actually adopts the traditional software testing. Therefore, it has not aroused general interest in the field of product family testing. In light of this, plus our approach constructs whole product family directly instead of an ‘assembly line’ (product line), i.e., application engineering is not required, we will not discuss individual product testing in detail.

There are two main kinds of testing approaches [TAK+14]: product-based testing and family-based testing. The former operates tests on generated products. It might rely on all products, or a subset thereof, depends on different testing strategies. The latter does not need concrete product, only uses domain artefacts. Moreover, it should be able to detect invalid feature combinations.

Previous researches on product line testing have proposed several testing strategies [JHQJ08, RE12]:

- **Brute Force Strategy.** It aims at testing every artefacts in the domain engineering, including unit testing, integration testing and system testing for all possible variants at all levels. Indeed, this strategy ensures the correctness of product line family before any concrete product is generated, but it is only practical when the product family consists of few features. Otherwise, the oversized test suite cannot be handled in polynomial time. There is no special tactics in this area.
• **Pure Application Strategy.** It works like exhaustive testing, as to test all possible feature combinations, i.e., concrete software products. Thereby, everything are tested in the application engineering, which can be considered as a full set of product-based testing. This strategy is impractical due to (i) it cannot eliminate redundant artefacts, and (ii) effective results can only be conducted after all products are tested. Some testing techniques are useful here to detect errors, such as type checking [BS12, AKL08] and model checking [BH13, ASLK10a].

• **Sample Application Strategy.** It is the most used testing strategy in current PLE approaches. The idea is to test minimal number of products that can cover all artefacts pre-constructed in the domain. In comparison with the previous two strategies, it markedly reduces the number of tests. Several coverage criteria have been proposed, such as pair-wise [OMR10] and t-wise coverage [PSK +10]. However, the redundancy problem existed in pure application strategy is still not addressed. Furthermore, the premise of selecting useful samples is that all relevant artefacts must be identified clearly, which happens to be the greatest challenge in many PLE approaches.

• **Commonality and Reuse Strategy.** It tests the common parts at domain level, whereas the variable parts at the application level. Therefore, the redundancy problem is solved by reusing testing artefacts systematically. Another area where this strategy could potentially bring advantages is family-based testing, while the other three comply with product-based testing. However, it must aware all variability in the family. Several variability-aware testing can be used in this area, e.g., family-based type checking [ASLK10b], syntax checking [KVRE +12] and model checking [CHS +10]. Nevertheless, the aforementioned tests are only credible for a whole software product family, i.e., a closed-world that define a fixed set of variants.

In a nutshell, no matter what testing strategy is used, product-based testing has to face the fact that the number of possible product variants grows exponentially causes such thorough testing infeasible. Contrariwise, family-based testing does not have the exponential number of combinations problem, but the correlated family must be constructed with a high requirement for modelling and implementing variability.

In this chapter, we present how to perform a family-based testing, with commonality and reuse strategy, on software product families constructed using enumerative variability.
5.2 Domain Unit Testing

In Chapter 4, we have presented the process of constructing a PFA for vending machine family. The first step is implementing every leaf feature as an X-MAN component, which becomes the bottom-level building block of the PFA. Thus, domain unit testing is intended to check that every X-MAN component complies with its design specification and meets the needs of the customer [Wal02].

A leaf feature must be realised as an X-MAN component, either atomic or composite. Therefore, we can use the X-MAN CBD process to develop and test a feature implementation, before the further PFA construction. As Figure 5.2 shows, the X-MAN CBD process has been proposed in form of another W-model (do not confuse with Figure 5.1), which defines one V for the (sub)component development process, and one V for the system (composite component) development process, and conjoins the two processes into a single CBD process [LTT11]. Based on the domain requirement, the first V process identifies, creates, tests and deposits components, whereas the second V process composes the repository components into a bigger composite component, as well as verifies and validates each composition and resulting component.

![Figure 5.2: The W-model for X-MAN CBD process [LTT11].](image)

The resulting component is atomic or composite, depending on the complexity of the feature functionality. For example, in the vending machine family, feature cancel has only one function, which is ending the purchase if the customer decides to cancel the order. Thereby, we can create an atomic component Cancel that encapsulates all the computation. Therefore, we can test its code directly. On the other hand, if a feature’s function is complicated, or parts of the function are possibly reused by other
features, we should create a composite component for it. Figure 4.5 shows a composite component Coffee mapped onto feature COFFEE.

To summarise, domain unit testing guarantees all the X-MAN components are functional correct, before they are selected and deployed as the bottom-level components of a PFA.

### 5.3 Domain Integration Testing

As Figure 5.1 shows, domain integration testing is a mid-level testing where individual domain artefacts are combined and tested as a group, in order to detect the defects in the interfaces and interactions between integrated modules (code units, components, etc.). However, in PLE, usually a feature is implemented as multiple code fragments scattered across a number of objects, or several features shares a piece of code. As a result, a product line may consist of massive numbers of domain artefacts, and intricate dependencies among them. Simply put, it is impossible to cover all combinations of domain artefacts in current PLE approaches [JHQJ08].

Our feature-oriented component-based approach is capable of realising enumerative variability in the PFA, as they are specified in the feature model. In other words, every valid variant of features should correspond to one and only one variant of components. Therefore, in order to certify that all valid products are identified in the PFA, we only need to examine two mapping relations as follows:

- The mapping between leaf features and bottom-level components.
- The mapping between variation points (in feature model) and variation generators (in FX-MAN).

Firstly, we compare the leaf features and bottom-level components. A leaf feature at least has one corresponding X-MAN component as its implementation. Therefore, if the amount of bottom-level components is smaller than the amount of leaf features, we know that the product family is incomplete. On the other hand, if any feature interaction is detected in the requirement, the number of bottom-level components should be larger than the number of leaf features, due to we use the multiple implementations strategy to handling the feature interactions, otherwise the product family is functional incorrect. Moreover, according to the interaction rules established in the family filers, we can calculate the desired amount of X-MAN components. For instance, as listed in
Table 5.1, there are 7 leaf features in the vending machine feature model (Figure 4.4) and 8 bottom-level components in the vending machine PFA (Figure 4.16). Notably, feature \texttt{Change} is mapped onto two components.

<table>
<thead>
<tr>
<th>Leaf Feature</th>
<th>Component Name</th>
<th>Component Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Coffee}</td>
<td>Coffee</td>
<td>Composite</td>
</tr>
<tr>
<td>\texttt{Tea}</td>
<td>Tea</td>
<td>Composite</td>
</tr>
<tr>
<td>\texttt{Cancel}</td>
<td>Cancel</td>
<td>Atomic</td>
</tr>
<tr>
<td>\texttt{Cash}</td>
<td>Cash</td>
<td>Composite</td>
</tr>
<tr>
<td>\texttt{Gift Voucher}</td>
<td>Gift</td>
<td>Atomic</td>
</tr>
<tr>
<td>\texttt{Card}</td>
<td>Card</td>
<td>Composite</td>
</tr>
<tr>
<td>\texttt{Change}</td>
<td>Change</td>
<td>Atomic</td>
</tr>
<tr>
<td></td>
<td>Cashback</td>
<td>Atomic</td>
</tr>
</tbody>
</table>

Table 5.1: Vending machine: Leaf features and their components.

Secondly, we compare the variation points and variation generators. The former not only denotes the variability, but also reveals the relationships between siblings features. However, these feature relationships are not, or at least not completely, realised in PLE. Contrariwise, our PFA defines a set of variation generators that models and implements the full set of variation points, as well as realises the feature relationships as component relationships. For example, the vending machine feature model (Figure 4.4) contains 4 variation points (two ‘optional’, an ‘alternative’ and an ‘or’), whilst its PFA has 4 variation generators (two \texttt{OPT}, an \texttt{ALT} and an \texttt{OR}) embodied at the same positions. Notably, the variable features can exist at the different levels of a feature model, thus the variation generators can be nested in a family model. Therefore, in order to simply calculate the product variants, we should examine the variation generators from bottom to top. Extra components caused by feature interactions, e.g., \texttt{CashBack} in Figure 4.16, must not be connected to any variation generators, due to they are only considered during family composition. As a matter of fact, the feature model and the PFA should be structurally isomorphic.

So far, we have confirmed that all possible combinations of all variations can be generated from the product family. However, we still need to examine the interrelations among components inside the combinations. How closely components are interrelated in software architectures is defined as coupling [DWT15]. In software testing, coupling is used to measure the degree of connection and the amount of interaction between modules [Kin84, Mar02]. It is called coupling-based integration.
testing [JO96, OAA00]. The higher the coupling, the more likely it is that changes to the inside of a module will effect the original behaviour of another one [CGVG98]. As a result, more tests with higher complexity are demanded. There are several levels of coupling, as listed below in increasing order of malignity:

- **No coupling.** All the modules do not communicate with each other at all. In other words, they are completely independent of each other [NOM10]. However, no coupling is only feasible in very small and simple applications [YSCO04]. That means, it is impossible in a product family.

- **Data coupling.** The communication between modules is limited, i.e., via scalar parameters, in which only simple arguments are allowed to pass directly, e.g., variable and array [MM88]. The passed data is always used for an invocation or a return of control [Kin84]. Data coupling is the loosest and best coupling [NOM10].

- **Stamp coupling.** Like data coupling, the communication between modules is also limited. But it passes composite data item, which usually is a entire data structure. Thus, sometimes a data structure may contain pieces of data that are unnecessary to the recipient module [MM88]. As a result, the computation unit of the recipient module may be more complicated than it is needed to handle the superfluous data [Mar02]. As the second loosest coupling, stamp coupling is usually inevitable in object-oriented programming [HM95].

- **Control coupling.** It occurs when a module sends control data to another with the aim of controlling the internal logic of the recipient module [Mar02]. The control data is regarded as a condition flag, which is used to direct the execution order of program instructions [MM88]. That means, the activity of recipient module depends on the value of the control data [Kin84]. In object-oriented programming, control couplings are usually necessary.

- **External coupling.** Modules communicates through an external medium that they can all access, such as a file or a specified object for temporary data storage [JO95, JO98]. Notably, only one copy of common external data is allowed at a time [Lea95]. Consequently, the occurrence of external coupling makes the integration testing more complicated.

- **Common coupling.** In some scenarios, a software system contains a global
5.3. DOMAIN INTEGRATION TESTING

area, in which data can be accessed and modified by every module of the system [SVMH90, DK93]. The usage of common coupling should be careful in software engineering due to it is difficult to determine which module may have made an erroneous change to the common data [Mar02].

- **Content coupling.** A module is actually refers to the inside of another, i.e., branch into the code [YTB05, Lea95]. For example, one method in a component can directly access the implementation of another method in another component [Li01]. This form of coupling forces the developer to understand semantics implemented in another module [Lee99]. On the other hand, with content coupling, a module cannot be modified, or tested, independently. It is a design defect that should be avoided as much as possible.

Apparently, no matter in a single system or in a software product line, the higher the coupling, the harder the testing [AO04]. Unfortunately, current PLE approaches inevitably produce some bad couplings. For instance, KobrA is a component-based approach to construct software product lines [ABM00]. A KobrA component is defined by a set of interrelated UML diagrams that describe both its specification and implementation [ABB+08]. As shown in the classic case study of the library systems product line, content couplings and external couplings both exist in the final family model [BMG01]. The former is caused by direct method calls between KobrA components, whereas the latter results from the external database used for communication. Another example is Koala component model, which is invented for consumer electronics software [VOVDLKM00]. Koala defines a module to glue interfaces between components by implementing all functions of all interfaces whose tip is bound to this module. Then it passes parameters to the module to determine which components are selected, which results in control couplings.

On contrary, the FX-MAN component model we used for product family construction only defines loose couplings. The composition mechanism of FX-MAN does not rely on method calls, thus no content coupling is involved. Alternatively, the components are composed by exogenous connectors, whose semantics also describe the interactions between components, thus no control coupling is involved. Moreover, a component transfer data to another one through a data channel. A data channel is established to link a data input port and a data output port [HBB+04]. Thereby, there are no common coupling and external coupling in our PFA. The only two couplings existed in an FX-MAN component are data coupling and stamp coupling [HM95], both are defined by data channels. As a natural result, the domain integration testing
in our PFA only involves the data channels. All we need to check are: (i) source (data port of the sending components), (ii) target (data port of the recipient component) and (iii) directions of the data flows, which can be easily verified in our visual tool (see Chapter 6).

5.4 Domain System Testing

Domain system testing aims at evaluating the family’s compliance with the domain requirements. In other words, after all the variants are verified in domain integration testing, we must investigate whether all the products function correctly. As we introduced earlier, product-based testing is able to validate the behaviour of individual product, but it is impractical in the context of PFE, due to an exponential number of combinations of assets is unmanageable [TAK+15]. Contrariwise, our domain system testing can perform a family-based testing based on a family statechart derived from the PFA.

According to the vending machine PFA (Figure 4.16), after a drink is selected, F-LOP starts a loop. In the first iteration, component Cancel is invoked, in which customer can give up the order. Otherwise, the state of the order changes to ‘pending’. Therefore, in the next move, the F-SEL turns the control flow to Cash, Gift or Card. After the payment, the first iteration ends and the second one starts, from F-LOP again. The customer has the another chance to cancel the order. If no cancel needed, the state of the order changes to ‘paid’, and leads the control flow to Change, or Cashback if Card is included in the product. To verify the behaviour mentioned above, we start with deriving a statechart from the vending machine PFA.

In Section 3.3.2, we have presented the conversion of family statecharts from FX-MAN components with different composition connectors and adaptors. Every component is mapped onto a basic state, while every component set is mapped onto a state set. Eventually, a (sub)family, as a composition of component sets, is mapped onto a compound state. Afterwards, every control transformation specified by composition connectors and adaptors is mapped onto a state transition between states. The construction of family statechart follows a top-down process in PFA, i.e., starts from the top connector. Every state therefore can include regions, which are containers for holding substates and transitions referring to connectors and adaptor at the next level.

Figure 5.3 shows the statechart of the vending machine family, which enables us to use a classic black box testing technique, namely state transition testing (STT)
While the product family specification is modelled by a statechart, STT generates a set of test cases that aim at covering all the transitions and states in the statechart. Each test case corresponds to a sequence of one or more transitions between states [BIMR97]. It is worth noting that in our family statechart, every basic state (e.g., Coffee, Tea, ...) is validated in the domain unit testing, whereas every state set (e.g., Beverage, Payment, ...) is validated in the domain integration testing. Therefore, we only need to deal with the test cases with one transition between two states.

In Figure 3.12, 3.13 and 3.14, we have demonstrated the statements of family connectors and adaptors in the statecharts. If an F-SEQ or F-SEL connects \( n \) component sets, the statechart shows \( n \) transitions between states, whereas an F-GUD yields one transition and an F-LOP yields two. Additionally, the ‘or’ variation generator corresponds to an aggregator (Figure 3.7), which causes \( 2n \) transitions between states where \( n \) components are composed by the ‘or’ variation generator. Therefore, as shown in Figure 5.3, we only need to test 12 state transitions in the vending machine family statechart, due to the PFA contains two F-SEQs (which correspond to transitions No.1 and 2, and No.9 and 10, respectively), an F-SEL (which corresponds to transitions No.11 and 12), an F-LOP (which corresponds to transitions No.7 and 8) and an ‘or’ variation generator (which corresponds to transitions No.3 to 6).

In comparison with family-based testing, product-based testing is apparently cumbersome. For instance, if we use a PLE approach to develop the vending machine
CHAPTER 5. TESTING PRODUCT FAMILIES CONSTRUCTED BY EV

Figure 5.4: Vending machine: Variability-aware testing case (pseudo code).

family, we have to configure, generate and test all products in order to verify and validate the entire family. In our case, we have to test all 30 products, with 232 test cases in total. As a matter of fact, a lot more test cases might be required if the features are not realised in a explicit manner. Some researches have pointed out that the generated products always share a massive amount of common code, but only variable code resulting in additional effort actually has the impact on analysis results [CE00, PBvDL05]. Hence, variability-aware testing [KVRE+12, LvRK+12], as an extension of variability-aware analysis [TAK+12], has been put forward with the aim of only testing the common code once.

Variability-aware testing consists of three major steps: (1) analysing the code base
to identify the variability, i.e., the variable code, (2) creating a variability-aware model by encoding the variability in a structural representation, e.g., abstract syntax tree (AST), (3) implementing a test case, i.e., a single program, with variability context for execution. Indeed it is a complicated approach due to the challenges occurred during the process, such as the choice of locality, granularity and sharing. The outcome of variability-aware testing is a single program parameterised on variable features. For instance, Figure 5.4 shows the pseudo code of a single program, which is the resulting test case of vending machines family. Accordingly, we can draw a corresponding statechart that uses boolean features as a part of the guard conditions, as demonstrated in Figure 5.5. Such a statechart is a FODA-like statechart. As a result, every test case must be design with a configuration of features. Eventually, we still need 30 test cases from Figure 5.5. The number looks acceptable, but it becomes deteriorate seriously with increasing number of features. Moreover, the difficulties of variability-aware testing actually are (i) identifying the variability in the code base and (ii) reconstructing a new variability-aware model, which consume a significant amount of time and effort.

![Vending Machine Statechart](image)

**Figure 5.5: Vending machine: FODA-like family statechart.**

In conclusion, the family-based testing based on the PFA implemented by enumerative variability is the most promising testing approach to verify and validate a product family.
5.5 Summary

Traditional software testing typically attempts to execute an application with the purpose of detecting errors, bugs or defects. Software product line testing is a lot more difficult due to the implementation of product line usually cannot be executed straightforwardly. Moreover, for a large family, testing all products are not feasible. Even though dozens of testing methods and strategies have been proposed, most of them are not practical due to the limitation of PLE itself.

We have presented our family-based testing approach particularly for our feature-oriented component-based software product families implemented by enumerative variability. That means, our testing approach may not be suitable for others. Another shortcoming of our testing approach is the lack of automation tool at the domain level. We plan to develop a plug-in tool for domain integration and system testing incorporated in the current IDE. Furthermore, we will investigate more testing techniques for our product family.
Chapter 6

Tool Support

“If a craftsman wants to do good work, he must first sharpen his tools.”
— Confucius

6.1 Introduction

We have presented the new FX-MAN component model in Chapter 3, and an approach for software family construction based on FX-MAN in Chapter 4. In order to experiment with the whole process of constructing product families by modelling and implementing enumerative variability, we have developed a web-based integrated development environment (IDE), called Enumerative Variability Modelling Tool (EVMT). The IDE consists of several tools, including feature modelling tool, component modelling tool and family modelling tool. It also enables the user to operate testing and download final products (in form of complete executable code). Figure 6.1 shows the toolset in our IDE. Each tool is designed for an activity in the PFE life cycle. Thus, our IDE becomes a closed loop development system.

The graphical user interface (GUI) of our IDE is realised using HTML5 and CSS3, whereas the computation is developed using JavaScript. In particular, we adopt the latest edition of ECMAScript as JavaScript specification since its significant new syntax, including classes and modules, supports complex applications. Additionally, we import jQuery, the most widely deployed JavaScript library, to improve code quality and enhance system extensibility. Moreover, our IDE offers a client-side repository, which is a NoSQL database for massive amounts of structured data. Every component,
no matter atomic or composite, is capable of deposition and retrieve, so that it can be reused for further construction. For the purpose of user-friendliness, all building blocks, including constraints and interaction, can be easily added through buttons and dialogue boxes.

The technological details will be elaborated in the following sections.

### 6.2 Web-based Application

Unlike traditional desktop applications, web-based applications are installed or deployed on remote servers. Users can access a web-based application via a browser. Since the web technologies become more and more advanced, almost any client software can be replaced by its web-based version. For example, Microsoft launched *Office 365*, as an alternative of the classic desktop software *Microsoft Office*.

In comparison with desktop applications, web-based applications have many advantages, including but not limited to:

- **Flexible Access.** A web-based application can be accessed anywhere via any device with a suitable browser with internet connection. Thus, a web-based applications is naturally a cross-platform software, i.e., it does not matter what kind of operating system is installed on local computer. Moreover, it seems to provide a prospect of remote collaboration, e.g., global teams, home working and real-time processing. Contrariwise, with a client software, team members must stay together, and only one member can use the software at a time. For
instance, *Google Docs* allows multiple users to modify one file simultaneously, whereas desktop *Microsoft Office* only allows one user to operate.

- **Easy Installation.** Web-based applications are installed on servers by software companies or developers. It is unnecessary to buy installation discs, or download installation files. Instead, clients only need to setup a new user on the correct website. In addition, at present, it is very common that a user account is granted permission to access various web-based applications. For example, a legit Google account can access all services provided by Google (e.g., *Gmail*, *Google Maps*, *Google Drive*) and numerous third-party sites and applications (e.g., *Dropbox*, *Youtube*, *Quora*).

- **Always Up-to-date.** Software update is inevitable because of the covered bugs and new requirements. For a desktop application, users have to keep tracking its latest version, and install the update patches, or worse, reinstall the entire application. However, everyone accesses the latest version of a web-based application via URL, due to only one version on the server is actually released.

- **Security.** Many people believe that web-based applications are less secure than traditional software installed on local computer. However, it is the other way around. For instance, some web-based applications require valid credentials, otherwise it remains unavailable, even if a user’s computer is in the wrong hands. Furthermore, users can be granted different levels of authority in order to restrict the functions and data that they can access. On the other hand, current cyber security technologies provide many ways to reduce the risk of cyber attacks, such as SSL enforcement, i.e., accessing a web-based application via a HTTPS connection.

Taking into account the aforementioned advantages, we decide to develop a web-based IDE for modelling enumerative variability and hence constructing product families.
6.3 Technologies

6.3.1 HTML5

_Hypertext Markup Language_¹ (HTML) is a standard markup language that describes the elements of a webpage. More precisely, each kind of element is annotated by a unique ‘markup’, called _tag_, such as ‘<div>’ (defines a division), ‘<img>’ (defines an image), ‘<p>’ (defines a paragraph), etc.

A HTML element is formed as:

```
<tag attribute1="value1" attribute2="value2"> CONTENT </tag>
```

The attributes inside a tag control its properties and functionalities. Some attributes modify HTML elements (e.g., ‘id’, ‘style’), whereas others add functions by running specific scripts (e.g., ‘onclick’, ‘onkeypress’). The text content of the tag is placed between a pair of label: a ‘start tag’ and ‘end tag’. However, not every label contains content, for example, ‘<br>’ defines a link break without content.

HTML also defines the structure of a webpage in form of nested HTML elements. The HTML code in Figure 6.2 shows an example. It present a simple webpage with a

```
1   <html>
2     <head>
3       <title>Page Title</title>
4     </head>
5     <body>
6       <div>
7         <p>This is a paragraph.</p>
8         <div>
9           <img src="avatar.png" alt="Avatar">
10       </div>
11     </div>
12     </body>
13   </html>
```

Figure 6.2: HTML example.

The content of this page contains a big division. Inside this division, there are a text paragraph and another division which again contains an image.

As the latest version of HTML, HTML5 becomes the cornerstone of today’s open web platform. The major difference between HTML5 and previous versions of HTML is

¹https://www.w3.org/TR/html/
is that the older ones require a number of proprietary plugins and APIs in order to foster the full potential the web offers. By providing new elements, attributes and behaviours, HTML5 supports a much larger collection of technologies that aims at building more complex and powerful web site and web-based applications with common interfaces and simpler semantics. For example, HTML5 integrates new elements for graphic ('<svg>', '<canvas>') and multimedia ('<audio>', '<video>'). Consequently, plugins like Flash Player are no longer needed.

Our web-based IDE uses many HTML5 new features, such as canvas, (offline) storage and MIME (Multipurpose Internet Mail Extensions). Moreover, the IDE can be accessed via mobile devices, due to HTML5 is also a strong candidate for cross-platform mobile applications.

### 6.3.2 CSS3

Cascading Style Sheets\(^2\) (CSS) is a language used for defining the style of an HTML document. In other words, CSS is able to describe how an HTML element should be rendered on the browser by controlling the presentation of layouts, colours and fonts.

In order to specify the target elements and their style properties, CSS defines a list of rules using a simple syntax. Figure 6.3 shows an example of CSS.

```
1  div {
2    height: 100px;
3    width: 100px;
4    background-color: \#yellow;
5    color: \#red;
6    border: 4px solid \#blue;
7    font: italic 10px Arial;
8    margin-top: 10px;
9  }
10
11 div p {
12    line-height: 150%;
13    text-indent: 20px;
14  }
```

**Figure 6.3: CSS example.**

CSS code, the division ('<div>') is a 100×100 square with full yellow background, in which the text is 10 pixels, italic, Arial font in red. In addition, the division is wrapped by 4 pixels wide solid line in blue, and a 10 pixels high space is fixed on its top border.

\(^2\)https://www.w3.org/TR/CSS/
Inside the division, the paragraph (‘<p>’) has a 1.5 times line height. The indent of the first line is 20 pixels in width.

In this example, the division and paragraph are specified by their types. In other cases, an element can be specified by its id (i.e., ‘#id’), or class (i.e., ‘.class’). Notably, only the id is unique, type and class can be duplicated. Therefore, CSS uses inheritance, i.e., ancestor-descendant hierarchy of HTML elements in a document tree based on nesting, to specify elements.

CSS3 is the latest version of CSS. It provides a more powerful selector, which can choose an element based on its (partial) attributes. It also brings a lot of new features for elements, such as rounded corners, shadows, gradients, transitions and animations. Finally, CSS3 defines more styles for layout, like multi-columns, flexible box and grid layouts. Thereby, we decided to use CSS3 to control the presentation of our IDE.

### 6.3.3 JavaScript

JavaScript\(^3\) (often abbreviated as JS) is a high-level scripting language, primarily used on the web pages. By comparison with tradition programming languages, e.g., Java and C#, it has some novel characteristics, such as weakly typed, dynamic and prototype-based and. Below we will elaborate in more detail:

- **Weakly typed.** Traditional software programming languages are strongly typed, such as Java or C++, which defines a strict, static typing rules at compile time. Hence, mismatched types of variable assignment, return values and function calling can cause errors or exceptions during compilation. On contrary, a weakly typed language has loose rules. For example, JavaScript uses the `var` statement to declare a variable with implicit data type. As a result, sometimes unpredictable results are output at runtime.

- **Dynamic.** If a programming language is dynamic, the behaviour it implements is executed directly at the runtime. That means, it does not matter that code is added, modified or deleted, the program can be executed straightforwardly without compilation. The developer can observe the new result immediately, which saves a lot of time. On the other hand, dynamic programming language allows users to develop a program in an incremental manner, like stepwise refinement. As a classic dynamic language, JavaScript also provides an `eval` function that can execute statements provided as strings at run-time.

\(^3\)http://www.ecma-international.org/ecma-262/
• **Prototype-based.** In prototype-based programming, behaviour is reused via delegation, which is a process of reusing existing objects, namely *prototypes*. Therefore, prototype-based programming is considered as a style of object-oriented programming. Almost all prototype-based applications are based on dynamic programming language, e.g., JavaScript. Thus, users are suggested to alter the prototypes during run-time, with the aim of focusing on the behaviour of systems, instead of worrying about the classification of objects.

```html
1 <html>
2 <head>
3   <title>Page Title</title>
4   <link href="code.css" rel="stylesheet" type="text/css">
5 </head>
6 <body>
7   <div>
8     <p>This is a paragraph.</p>
9   </div>
10  <script type="text/javascript">
11     var para = document.querySelector('p');
12     para.addEventListener('click', showMessage);
13     function showMessage() {
14       alert("I click the paragraph!");
15     }
16  </script>
17 </body>
18 </html>
```

Figure 6.4: JS example.

The goal of JavaScript is to “make webpages alive”. Figure 6.4 shows an example of JavaScript in an HTML document. The code is embedded in the `<script>` tag. As we can see, the first JavaScript code line indicates the selection of HTML element `<p>`, and the next line add an event listener for `click` to this element. The rest code defines a new function, which implements an activity (i.e., popping a message) after the click. Figure 6.4 also shows the execution result of this JavaScript code.

In the early days, JavaScript was submitted to *ECMA International*, an industry association that standardises information⁴, for promotion. As a result, a new scripting-language standard has been published, namely *ECMAScript*, or *ES* for acronym. The latest and current edition is *ECMAScript 2018*, also known as *ES9* due to it is the ninth edition. We use the latest ECMAScript as JavaScript specification to develop our IDE since its significant new syntax, including classes and modules, supports complex applications. Moreover, it provides a lot of useful features, which make the programming

⁴http://www.ecma-international.org/
incredibly convenient, such as rest parameter, string interpolation, arrow functions, Map/Set data structure, etc.

6.3.4 jQuery

In order to improve code quality and enhance system extensibility, we import jQuery\(^5\), the most widely deployed JavaScript library, in our IDE development. With the help of jQuery, it becomes very easy to traverse HTML document, select HTML elements, handle events, implement animations and execute AJAX (Asynchronous JavaScript And XML) requests [G+05].

In general, jQuery offers fourfold advantages:

1. In a web-based application, jQuery separates its behaviour (implemented by JavaScript code) from its structural presentation (implemented by HTML markup). For example, event handlers can be added to DOM (Document Object Model) objects, i.e., representation of HTML elements, instead of additional HTML event attributes that call JavaScript functions.

2. jQuery offers better brevity and clarity with features, such as chaining effects and actions or shorthand method names.

3. One of the severe problems of JavaScript is the cross-browser incompatibilities. Even though there is a standard specification known as ECMAScript, JavaScript code works for one browser may not work for another. The reason is that different browsers utilise different JavaScript engines, e.g., V8 for Google Chrome and SpiderMonkey for Firefox. However, jQuery deals with all the cross-browser inconsistencies and provides consistent interfaces that works across all JavaScript engines.

4. The extensibility of jQuery allows new elements, events and methods to be easily added to the original code base. Furthermore, the integration of new items can be registered as a jQuery plugin and therefore be reused in other applications.

Figure 6.5 shows an example of jQuery. We rewrite the JavaScript code in Figure 6.4 using jQuery syntax. Apparently, the new program is shorter and simpler, but the result is still correct.

\(^5\)https://jquery.com/
6.3. TECHNOLOGIES

6.3.5 jTopo

jTopo, acronym of JavaScript Topology Library, is a fully interactive diagramming toolkit for all modern browsers. It provides developer-oriented service with strong extensibility. Developers can display data from any kind of data resource, and easily enable interactive creation, viewing and editing of graphs. jTopo also supports automatic diagram layout in order to help users understand complex graph structures.

jTopo is completely based on HTML5 canvas. It does not rely on any other third-party library. That means, no matter how complex a web page can be, jTopo will not taint its HTML document, DOM structure and program namespace. Developer can draw graphics by accessing the easy-to-use interfaces provided by jTopo.

Figure 6.5: jQuery example.

Figure 6.6: jTopo layers.
As Figure 6.6 shows, for every canvas, jTopo defines a stage, and multiple scenes inside. User can switch a scene to another in the stage. Nodes and links are defined in every scene, or in the containers of every scene.

6.4 Feature Modelling Tool

Feature modelling tool is divided into two parts. The full screenshot of feature modelling tool is shown in Figure A.1. On the left is a canvas that visualises the feature model construction (Figure 6.7), whereas on the right is the variant explorer that enumerates all valid product variants derived from the feature model in the form of feature combinations (Figure 6.9).

![Figure 6.7: Feature model canvas.](image)

In order to create a new feature model, firstly we click the button “New Feature Model”, which automatically create a mandatory feature as root. By double clicking the feature, we can edit its name. Then, we can add more features according to the type (“Mandatory”, “Optional”, “Alternative”, “Or”) by simple clicks on the vertical menu on the left side of canvas. After the features are pinned on the canvas, we need to link them to realise the father-children relationships in the feature model. For that purpose, we use multiple selection, i.e., hold the Ctrl key and click two features (father at first, children at second), and then click “Connection” in the menu. Moreover, to identify the ‘ALTERNATIVE’ or ‘OR’ feature group, we must multiple-select features in the
same group and click “Group” in the menu. Finally, we multiple-select two features and click “Requires” or “Excludes” with the aim of establishing cross-tree constraints.

It is worth noting that we can define a cardinality with explicit lower bound and upper bound for every ‘OR’ variation point in order to restrict the number of sub-features that can be selected. At least one feature is selected if no cardinality explicitly stated.

![Add Cardinality](image)

Figure 6.8: Cardinality for ‘OR’ variation point.

After a feature model is constructed in the canvas, we can observe all valid product variants in the variant explorer by clicking the “Refresh” button, as Figure 6.9 demonstrates. In this chapter, the example is still the vending machines family, but, for the sake of simplicity, we assume feature `CANCEL` is mandatory. Thus, the feature model defines 15 products, as shown in Figure 6.9. The “Resize” enables a wider display of variant explorer. It is convenient to show complex feature combinations.

### 6.5 Component Modelling Tool

#### 6.5.1 Atomic Component Modelling Tool

The component modelling tool consists of two gadgets: one for atomic component, and another for composite component. Figure A.2 shows the full screenshot of atomic component modelling tool.

The first step is to create an atomic component. By clicking “New Component” in the horizontal menu, a dialogue box is open, as shown in Figure 6.10(a). Here we can declare a new atomic component by defining its component name, service name, and input/output data. Notably, the service name can be automatically generated by extending the component name, if the “default” checkbox is ticked. The atomic component is ready to be implemented by clicking “Create” in the dialogue box.
In Figure 6.10(b), there are two text areas. The code template of computation unit is automatically constructed in top text area. We can complete the computation unit by adding functions, i.e., writing execution code. After that, we can write testing code in bottom text area, which can be directly invoked by a simple click on “Execute” in the horizontal menu. The testing result can be observed by web development tools, e.g., Chrome DevTools, Firefox Firebug or Opera Dragonfly. For example, Figure 6.11 shows the testing results in Chrome DevTools.

When a component is implemented, it must be deposited into the repository. Components will be retrieved later for further construction of composite components and families. The repository will be discussed in detail in Chapter 6.6.

6.5.2 Composite Component Modelling Tool

Figure A.2 shows the full screenshot of composite component modelling tool. The layout is similar to atomic component modelling tool. The component canvas is on the

---

6https://developer.chrome.com/devtools
7https://getfirebug.com/
8http://www.opera.com/dragonfly/
6.5. COMPONENT MODELLING TOOL

(a) Creating atomic component.

(b) Implementing atomic component.

Figure 6.10: Atomic component modelling tool.
left side (Figure 6.12(b)), whereas the repository is on the right side (Figure 6.13).

Figure 6.12(a) shows a dialogue box used to define name and service of a new composite component. The name and service will be displayed at the upper-left corner of the canvas, e.g., “Coffee : get” in Figure 6.12(b). We click the name-service pair to select the composite component, and then click “input” or “output” in the vertical menu in order to add data to the new component. The added data will be exposed, along with the service, to enable access from other components.

In order to construct a composite component, we need the building blocks, such as sub-components (atomic or composite) and connectors (including adaptors). The former are retrieved from the repository (see Chapter 6.6), while the latter are created by link texts below “Connector” in the vertical menu. Necessary inputs should be added to particular connectors, e.g., selector (SEL) and guard (GUD) in Figure 6.12(b).

The connections between components and connectors are created by multiple selection and click on “Connection”, similar to how we add relationships among features in Chapter 6.4. Likewise, the data flows are realised by data channels, which are connections added between data. The conditions must be manually defined, such as execution order (‘0’ and ‘1’) for sequencer (SEQ), and ‘this.cmd.includes(‘Cream’)’ for guard (GUD) in Figure 6.12(b).

After the composite component is constructed, by clicking “Generate” on the horizontal menu, a XML file is generated to record the structure of the component. Finally, we deposit it for further development.

---

For the clarity, data channels are omitted in the canvas, but shown in the screenshots in Appendix A.
6.6 Repository

In the previous section, we mentioned that components must be deposited in a repository, so that they can be retrieved later for composite component or family construction. Our IDE incorporates a repository developed by IndexedDB (Figure 6.13).

According to World Wide Web Consortium (W3C)\(^\text{10}\), IndexedDB is a large-scale client-side storage system that provides a low-level API for significant amounts of structured data. We prefer to use IndexedDB because of its new features:

1. IndexedDB is embedded in the browsers. Its API allows websites to directly

\(^{10}\text{https://www.w3.org/TR/IndexedDB/} \)
collect data without server-side scripting. Moreover, the data is permanent saved in the IndexedDB, even if the browser window is closed.

2. IndexedDB is a NoSQL (which mostly stands for “not only SQL”) database organised around the conception of JSON (JavaScript Object Notation) objects. Traditional database is relational, which organise structured data fields into predefined columns of tables. NoSQL database is non-relational, which does not incorporate the table model [HHLD11]. Instead, data is stored in a single document file. By comparison, NoSQL database is more flexible, efficient and scalable [Pok13].

3. Every user has their own database, due to every IndexedDB database is only responsible for one origin, which can be considered as a site domain. This domain cannot access or be accessed by others. Therefore, although we deploy only one tool on the server, users can work independently.

IndexedDB has three levels. The highest level is database, which contains the object stores, which in turn contain the data object. User can create more than one database with different name, but usually there is one database for a web-based
6.7. FAMILY MODELLING TOOL

application. An object store is an individual bucket for data storage. Every object is marked by an index for retrieve. A mechanism, namely cursor, is presented for traversing the database or iterating over multiple records thereof. Figure 6.13 shows the three levels of our repository, in which the atomic and composite components are stored in two object stores, respectively.

Although IndexedDB has many advantages, it still has some limitations. For example, IndexedDB is a client-side database, which means users cannot share data with each other via it. In practical, a product family is too large to be realised by one developer. Instead, such a task requires an organisation containing many developers and multiple characters. Hence, it is essential to prepare a cloud-based database for component sharing, e.g., MongoDB\(^1\). However, our tool is a proof-of-concept, with no industrial purpose. By weighing the tradeoff between benefit and cost, we eventually choose IndexedDB.

6.7 Family Modelling Tool

Figure A.4 shows the full screenshot of family modelling tool. The layout is similar to composite component modelling tool: the canvas is on the left side (Figure 6.14(b)), while the repository is on the right side (Figure 6.13).

Figure 6.14(a) shows a dialogue box used to define name and service of a new product family. The name and service will be displayed at the upper-left corner of the canvas, e.g., “VendingMachine : run” in Figure 6.14(b). We click the name-service pair to select the composite component, and then click “input” or “output” in the vertical menu in order to add data to the new family. The added data will be exposed, along with the service, to enable access from main method.

In order to construct a product family, we need the building blocks, such as leaf component, variation generators and family connectors. Components are retrieved from the repository (see Chapter 6.6). Variation generators and family connectors are created by link texts below in the vertical menu. Necessary inputs should be added to particular connectors, e.g., family selector (F-SEQ1) in Figure 6.14(b).

The connections between these building blocks are established by multiple selection and click on “Connection”, similar to how we add relationships among features in Chapter 6.4. Likewise, the data flows are realised by data channels, which

\(^1\)https://www.mongodb.com
are connections added between data\textsuperscript{12}. The conditions must be manually defined, such as execution order (‘0’ and ‘1’) for family sequencer (F-SEQ1, F-SEQ2), and ‘this.state==‘cancel’’ for family selector (F-SEL1) in Figure 6.14(b).

\textsuperscript{12}For the clarity, data channels are omitted in the canvas, but shown in the screenshots in Appendix A.
After the family model is structured, we need to realise the cross-tree constraint as specified in the feature model (Figure 6.7). Similarly, we multiply select two related
components and click “Requires” or “Excludes” in the vertical menu. The constraints will be automatically displayed in the middle of the components, as the purple arrow shown in Figure 6.14(b).

In Chapter 4.4, we present that the solution of addressing feature interaction problem in our family model. Our tool also provides a function to deal with it. First, we must retrieve the extra components (i.e., multiple implementation) for the feature interaction. Then, we select the interacting components and click “Interaction” in the vertical menu, which opens a new dialogue box to set up the interaction rules. For instance, Figure 6.15(a) illustrates that the CashBack replaces Change when Card and Change both exist. After that, in Figure 6.15(b), the extra component is connected to the nearest family composition connector that composes the components involved. The interaction filter is formed of a red dot with the text “FI”.

Finally, we generate a XML file to store the structure and elements of the product family by a click on “Generate” on the horizontal menu. Based on the XML file, all valid products can be directly derived in one go.

6.8 Product Explorer

After the product family is generated, we can open the product explorer to observe all products, each of which is executable. There are two major divisions in the product explorer. Their full screenshots are demonstrated in Figure A.5 and A.6, respectively. At the beginning, the product explorer shows a list that enumerates all valid products in the form of variability resulting from each variation generator at any level of nesting. By a click on any of the products, the list is switched to a canvas that graphically expresses the model of selected product. For example, Figure 6.16(a) shows all 15 vending machine products, whereas Figure 6.16(b) shows the model of vending machine No.4. From Figure 6.16(b), we can easily switch back to Figure 6.16(a) by the big green arrow on the left.

Product explorer is a very useful tool. It gives us a chance to compare the enumerative variability implemented in solution space with the one specified in problem space (variant explorer in Figure 6.9). If the results do not match, i.e., unequal total amounts or irregular constitution of a product, some mistake must be made during the construction of product family. Therefore, we can easily spot the errors in the family model. Any product in the product explorer can be directly tested in a built-in testing tool. Moreover, we can select any number of products (from 1 to all), and download
their source code in one go, i.e., batch download within a compressed file. For instance, Figure 6.17 shows all 15 JavaScript code files of vending machine products in a compressed file, which is downloaded from the product explorer.
6.9 Testing Tool

Our IDE provides a tool for testing the components in the repository (via context menu) and the products in the product explorer (via button on the horizontal menu).

As shown in Figure 6.18, the testing tool is comprised of three textareas. The left textarea displays the generated source code, while the upper right textarea is used to write testing code, and the bottom right one shows the testing result after execution (by clicking “Execute” on the top menu).

6.10 Other Functions

Except for the mentioned features, our IDE offers several basic functions across the tools by reusing common HTML controls, such as:

- **Clean.** It resets the tools to initial state, i.e., empty the canvas and remove the content in textarea.
6.10. OTHER FUNCTIONS

Figure 6.18: Testing tool.

- **Zoom In.** It makes the image of model in a canvas appear much larger and nearer.

- **Zoom Out.** It makes the image of model in a canvas appear much smaller and further away.

- **Export Image.** It exports the rendered scene of canvas as images, and saves it as a PNG file in the device.

- **Refresh.** It updates the information displayed in the tool, e.g., reflects the most recent added component to the repository being viewed.

- **Remove.** It deletes an object from the screen or the container, e.g., deletes a component from repository.

These functions make our IDE more user-friendly.
6.11 Summary

Even though numerous PLE approaches have been proposed, only a few of them have tool supports. Moreover, current tools more or less have defects. For example, pure::variants does only supports annotative approaches with positive variability [Beu12]. It allows user to structure a feature model, but it cannot calculate the exact amount of products in the family. In addition, as an eclipse-based application, it more likes a code editor that does not has good compatibility for family design. Similar tools, such as Gears [KC13] and FeatureIDE [TKB+14], barely supports modelling languages, too [SK09]. On the other hand, the weak integration among the constituents of the tool chain becomes a huge issue [STB+04, Jen07, DH09]. For instance, a web-based tool called SPLIT does construct enumerative variability in the problem space, but does not modelling and implementing enumerative variability in the solution space [MBC09].

Our IDE provides comprehensive facilities to model and implement enumerative variability in the solution space for product family engineering. It applies a great integration among the constituents of the tool chain, which consists of a source code editor (for computation units of atomic components), graphic canvases (for modelling feature models, composite components and product families), a repository (for component storage), a code generator (for automatic source code generation), object browsers (for variant and product explorer) and a debugger (for testing components and products).

Finally, our IDE can be accessed via http://www.cs.man.ac.uk/~qianc?EVMT.
Chapter 7

Case Study: External Car Light Family

“Practice is the sole criterion for testing truth.”

— Chinese Proverb

7.1 Introduction

Scalability is a great challenge in PLE [KLD02]. Many PLE approaches look attractive and elegant in the laboratory, but they turn out to be infeasible in the field [SIMA13]. On the contrary, our approach is able to construct a software product family of non-trivial size, i.e., based on a feature model containing dozens of features and hundreds of variants. Our approach can also handle some real world problems such as feature interactions and crosscutting concerns. In this chapter, we present an example of a product family of external car lights (ECL) systems, which is adapted from an industrial use case provided by pure-systems GmbH\(^1\). We will show screenshots (cropped for legibility wherever necessary) for the various stages of the ECL family example, from creating components for leaf features, applying variation operators, to composing the whole family using family composition connectors.

The lighting system of a vehicle is made of a set of external lighting and signalling devices. They not only indicate the intended direction of travel but also inform other

\(^1\)https://www.pure-systems.com/
drivers and pedestrians of the vehicle’s presence. Figure 7.1 shows the different kinds of front lights of a car, including high/low beam headlight, cornering light, fog light and daytime running light. A car headlight produces two types of light beams. High beam headlights are brighter than the low beam ones. High beam is suitable for driving in dark areas or at night, but it is not always safe. Thus, if a vehicle is around other cars or pedestrians, in order to increase the driver’s visibility to other drivers and avoid hazards, it is important to switch the headlights to low beam. Cornering lights offer higher safety when driving in turning corners due to the additional illumination of the area to the side of the car. Fog lights are designed to illuminate the area in front of the vehicle when the visibility conditions that caused by fog or snow are poor. Daytime running lights, as the name implies, help drivers see oncoming traffic easier during daylight conditions.

In the following sections, we will discuss the details of modelling and implementing the ECL family according to its functional requirement.

### 7.2 Requirement and Feature Model

The feature model is shown in Figure 7.2. The features are described by the following requirements:
7.2. REQUIREMENT AND FEATURE MODEL

Figure 7.2: Feature model of ECL family.

- An ECL system supports Beam Configuration that is able to switch the headlights between Low Beam and High Beam. A headlight, no matter high beam or low beam, is filled with either Xenon or Halogen. Fog Lights and Daytime Running Lights are also controlled by the ECL system, if they are installed on the vehicle. Moreover, some ECL systems provide different kinds of Driver Assistsances for the driver.

- Some vehicles install specific separated lights for Daytime Running Lights, which usually use LED lights or Standard Bulbs. Some others achieve the same lighting effect via Reduced Low Beam headlights.

- Driver Assistance aims at increasing the driving safety. It is realised by at least one, or any combination), of the following services: Automatic Light, Automatic High/Low Beam and Cornering Lights.
– **Automatic Light** is a service that controls the lights according to the light sensor. If the environmental light drops to less than 1000 lux, the Low Beam headlights and Cornering Lights are switched on. When the light returns to 3000 lux, both lights are automatically switched off.

– **Automatic High/Low Beam** is a service that automatically switches headlights to low beam if the high beam is turning on whenever an oncoming vehicle is detected by camera. Then, it will switch the headlights back to high beam when the vehicle leaves.

– **Cornering Light** is realised by at least one (or any combination) of the following sub-systems: Static Cornering Light or Adaptive Forward Light.

  * **Static Cornering Light** automatically activates the Daytime Running Lights when the car is moving at a speed of at least 10 m/s and the steering wheel is above 15 degrees. Furthermore, it activates the Fog Lights if the steering wheel angle goes above 25 degrees.
  
  * **Adaptive Forward Light** automatically modifies the beam direction and shape according to road geometry. For example, the system steers the light cone inside the curve when entering a corner.

– **Fog Lights** can be manually activated by the driver, or automatically triggered by Static Cornering Light.

According to the requirement, if an ECL system supports Static Cornering Light, it must supports Fog Light. Otherwise, Static Cornering Light cannot works completely. Therefore, we can define a feature constraint, namely “Static Cornering Light requires Fog Light” in the feature model.

This example also contains feature interactions, which can happen when both Low Beam and Reduced Low Beam are included in a product. In order to mimic a daytime running light, the Reduced Low Beam adjusts the Low Beam by an extra parameter called the pulse duty cycle. As a result, we plan to implement two extra components for Xenon Low Beam and Halogen Low Beam, respectively.

With the above constraints and feature interactions, the ECL feature model defines an enumerative variability with a total of 384 valid variants. Figure 7.3 shows all valid variants in terms of feature combinations.
7.2. REQUIREMENT AND FEATURE MODEL

Figure 7.3: All 384 ECL valid variants.

<table>
<thead>
<tr>
<th>384 Valid Variants</th>
</tr>
</thead>
<tbody>
<tr>
<td>370. Xenon Low Beam, Xenon High Beam, Standard Bulb, Automatic Light, Automatic High Low Beam, Fog Light</td>
</tr>
<tr>
<td>371. Xenon Low Beam, Xenon High Beam, Standard Bulb, Automatic Light, Automatic High Low Beam, Static Cornering Light</td>
</tr>
<tr>
<td>372. Xenon Low Beam, Xenon High Beam, Standard Bulb, Automatic Light, Automatic High Low Beam, Static Cornering Light, Adaptive Forward Light, Fog Light</td>
</tr>
<tr>
<td>373. Xenon Low Beam, Xenon High Beam, Standard Bulb, Automatic Light, Automatic High Low Beam, Static Cornering Light, Fog Light</td>
</tr>
<tr>
<td>374. Xenon Low Beam, Xenon High Beam, Standard Bulb, Automatic Light, Fog Light</td>
</tr>
<tr>
<td>375. Xenon Low Beam, Xenon High Beam, Standard Bulb, Automatic Light, Static Cornering Light</td>
</tr>
<tr>
<td>376. Xenon Low Beam, Xenon High Beam, Standard Bulb, Automatic Light, Static Cornering Light, Adaptive Forward Light, Fog Light</td>
</tr>
<tr>
<td>377. Xenon Low Beam, Xenon High Beam, Standard Bulb, Automatic Light, Static Cornering Light, Fog Light</td>
</tr>
<tr>
<td>378. Xenon Low Beam, Xenon High Beam, Standard Bulb, Fog Light</td>
</tr>
<tr>
<td>379. Xenon Low Beam, Xenon High Beam, Standard Bulb, Static Cornering Light</td>
</tr>
<tr>
<td>380. Xenon Low Beam, Xenon High Beam, Standard Bulb, Static Cornering Light, Adaptive Forward Light, Fog Light</td>
</tr>
<tr>
<td>381. Xenon Low Beam, Xenon High Beam, Standard Bulb, Static Cornering Light, Fog Light</td>
</tr>
<tr>
<td>382. Xenon Low Beam, Xenon High Beam, Static Cornering Light</td>
</tr>
<tr>
<td>383. Xenon Low Beam, Xenon High Beam, Static Cornering Light, Adaptive Forward Light, Fog Light</td>
</tr>
<tr>
<td>384. Xenon Low Beam, Xenon High Beam, Static Cornering Light, Fog Light</td>
</tr>
</tbody>
</table>
From the ECL feature model and the above requirements, we can construct the corresponding family model with the same enumerative variability in solution space, by the following steps described in Chapter 4.

### 7.3 Roadmap of ECL Family Construction

#### 7.3.1 Step 1: Implement Leaves Features

The first step is to create components as the implementations of leaf features, which are listed in Table 7.1, alongside the names of their implementations. If a feature’s behaviour is too simple to be further decomposed, it should be realised as atomic component, e.g., Fog Light is implemented as atomic component FogLight in Figure 7.4(a). Otherwise, we create some atomic components and use them to construct composite components as the implementations of features, e.g., Static Cornering Light is realised by composite component StaticCornerLight in Figure 7.4(b).

<table>
<thead>
<tr>
<th>Leaf Feature</th>
<th>Component</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fog Light</td>
<td>FogLight</td>
<td>Atomic</td>
</tr>
<tr>
<td>Xenon Low Beam</td>
<td>LowBeamXenon</td>
<td>Atomic</td>
</tr>
<tr>
<td>Halogen Low Beam</td>
<td>LowBeamHalogen</td>
<td>Atomic</td>
</tr>
<tr>
<td>Xenon High Beam</td>
<td>HighBeamXenon</td>
<td>Atomic</td>
</tr>
<tr>
<td>Halogen High Beam</td>
<td>HighBeamHalogen</td>
<td>Atomic</td>
</tr>
<tr>
<td>Reduced Low Beam</td>
<td>DRL_LowBeam</td>
<td>Atomic</td>
</tr>
<tr>
<td>LED</td>
<td>DRL_LED</td>
<td>Atomic</td>
</tr>
<tr>
<td>Standard Bulb</td>
<td>DRL_Bulb</td>
<td>Atomic</td>
</tr>
<tr>
<td>Adaptive Forward Light</td>
<td>AdaptiveForwardLight</td>
<td>Composite</td>
</tr>
<tr>
<td>Static Cornering Light</td>
<td>StaticCornerLight</td>
<td>Composite</td>
</tr>
<tr>
<td>Automatic High/Low Beam</td>
<td>AutomaticHighLowBeam</td>
<td>Composite</td>
</tr>
<tr>
<td>Automatic Light</td>
<td>AutomaticLight</td>
<td>Composite</td>
</tr>
</tbody>
</table>

Table 7.1: Leaf features and related components.

For instance, we create three atomic components TurningSpeed, TurningAngle and ActivateSCL using the atomic component modelling tool, respectively. Each of them is deposited in the repository. Then, in the composite component modelling
tool, we retrieve all three components to construct the composite component Static-CornerLight, as shown in Figure 7.4(b). As the implementation of feature Static Cornering Light, this component is also deposited in the repository waiting for the further construction of product family.

```java
class FogLight {
    constructor() {
        this.cmd = null;
        this.next_cmd = null;
    }

    toggleLight() {
        if (this.cmd.get("toggle") == "on") {
            console.log("Turning on fog lights...");
            console.log("-- Fog lights are switched on --");
        } else if (this.cmd.get("toggle") == "off") {
            console.log("Turning off fog lights...");
            console.log("-- Fog lights are switched off --");
        }
        this.next_cmd = new Map().set("light", "stop");
    }
}
```

(a) Implementing FOG LIGHT by atomic component FogLight.

(b) Implementing STATIC CORNERING LIGHT by composite component StaticCornerLight.

Figure 7.4: Implementing components for leaves features.
7.3.2 Step 2: Apply Variation Generators

Now, we turn to family modelling tool for ECL family construction. We retrieve 12 components, which are mapped onto 12 leaf features, from the repository to create

![Diagram of component sets](image)

**Figure 7.5:** Generate component sets by adding variation generators.
7.3. ROADMAP OF ECL FAMILY CONSTRUCTION

new component instances. Then we apply variation generators to them according to the feature model in order to generate variations as component sets.

Figure 7.5 illustrates all component sets created by adding various variation generators to bottom-level components:

(a) An ALT generator applied to the instances of components HighBeamXenon and HighBeamHalogen yields the component set $S_1 = \{\{HBX\}, \{HBH\}\}$, which can be regarded as the realisation of feature HIGH BEAM.

(b) An ALT generator applied to the instances of components LowBeamXenon and LowBeamHalogen yields the component set $S_2 = \{\{LBX\}, \{LBH\}\}$, which can be regarded as the realisation of feature LOW BEAM.

(c) An OPT generator applied to the instance of component FogLight yields the component set $S_3 = \{\{Fog\}, \emptyset\}$.

(d) An ALT generator applied to the instances of components DRL_LED and DRL_Bulb yields the component set $S_4 = \{\{LED\}, \{Bulb\}\}$, which can be regarded as the realisation of feature SEPARATED LIGHT. Another ALT generator applied to the instance of component DRL_LowBeam and component set $S_4$ yields the component set $S_5 = \{\{LB\}, \{LED\}, \{Bulb\}\}$, which can be regarded as the realisation of feature DAYTIME RUNNING LIGHT.

(e) An OR generator applied to the instances of components StaticCornerLight and AdaptiveForwardLight yields the component set $S_6 = \{\{SCL\}, \{AFL\}, \{SCL,AFL\}\}$, which can be regarded as the realisation of feature CORNERING LIGHT. Another OR generator applied to the instances of components AutomaticHighLowBeam, AutomaticLight and component set $S_6$ yields the component set $S_7 = \{\{AHLB\}, \{SCL\}, \{AFL\}, \{AL\}, \{AHLB,SCL\}, \{AHLB,AFL\}, \{AHLB,AL\}, \{SCL,AFL\}, \{SCL,AL\}, \{AFL,AL\}, \{AHLB,SCL,AFL\}, \{AHLB,SCL,AL\}, \{AHLB,AFL,AL\}, \{SCL,AFL,AL\}, \{AHLB,SCL,AFL,AL\}\}$, which can be regarded as the realisation of feature DRIVE ASSISTANCE.

7.3.3 Step 3: Apply Family Composition Connectors

The previous step aims at capturing the variability expressed in the feature model (Figure 7.2) by the component sets. Now we compose them into sub-families of the ECL family using family composition connectors. The choice of family composition connectors is a design decision, however it will not affect the total number of products in the (sub-)family.
As Figure 7.6 shows, we use a family composition connector \textit{F-SEQ} to compose two component sets \( S_1 = \{\{HBX\},\{HBH\}\} \) and \( S_2 = \{\{LBX\},\{LBH\}\} \) into a sub-family. According to the requirement, we know that the high beam and low beam cannot be emitted from the headlight simultaneously. That means, if the driver switches on the low beam, the system must automatically switches off the high beam (if at work), and vice versa. In ECL system, in order to control the toggles of lights, we define an argument, namely \textit{cmd} (abbreviation of \textit{command}), in form of a data structure that holds two key-value pairs. The first pair targets the desired light (e.g., \textit{FogLight}), while the second one determines the action (e.g., switch on or off). The code in Figure 7.4(a) expresses its usage. It is worth noting that if the value of first pair is ‘\textit{stop}’, it means no further move for any light in the system. Consequently, in Figure 7.6, based on the same command, we invoke the component in \( S_1 \) at first to switch on or off the high beam, and than invoke the component in \( S_2 \) to switch on or off the low beam. Hence, we actually toggle the headlight between high beam and low beam by the composition of \textit{F-SEQ}.

When we implement the behaviour of product family, the crosscutting concerns should be taken into account. According to the requirement of ECL family, there are several crosscutting concerns should be handled in this step. Figure 7.7 depicts one of the examples. If a vehicle installs \textit{FogLight}, then it can be manually activated by driver. In addition, it will be automatically switched on when \textit{Static Cornering Light} is turning on. Therefore, it becomes obvious that \textit{FogLight} is a crosscutting feature, and its component \textit{FogLight} must be invoked after \textit{StaticCornerLight} is executed. In Figure 7.7, our design uses a combination of \textit{F-LOP} and \textit{F-SEL} to achieve
the goal. We assume the first iteration invokes StaticCornerLight and returns a new command that targets FogLight as the argument of the next loop. As a result, the second iteration invokes FogLight, which returns a command that ends the loop.

After this stage, the basic structure of ECL family is established, as Figure 7.8 shows.

7.3.4 Step 4: Address Feature Interactions

Unlike crosscutting concerns, feature interaction problems cannot be solved by delicate design of the family composition connectors. Thus, we take an individual step to address the feature interactions occurred in the ECL family.

As presented earlier, the interactions arise when Low Beam (including Xenon Low Beam and Halogen Low Beam) and Reduced Low Beam are both selected in an ECL system. Therefore, we must realise two interactions in the family model. The functionality in component DRL_LowBeam switches on the low beam headlights with an altered pulse duty cycle. That means, the behaviour of LowBeamXenon and
Figure 7.8: Applying family connectors to the ECL family.

ExternalCarLight: activate

- F-LOFI
  - F-SEL
    - F-SEQ1
      - ALT
      - ALT

- OPT
  - OR
    - OR
      - ALT
      - ALT

- ALT
  - (HighBeamXenon)HBX
  - (HighBeamHalogen)HBH
  - (LowBeamXenon)LBX
  - (LowBeamHalogen)LBX
  - (StaticCornerLight)SCL
  - (AutomaticHighLowBeam)AHLB
  - (DRL_LowBeam)LB
  - (DRL_HighBeam)H
  - (DRL_LowBeam)LB
  - (DRL_HighBeam)H
  - (DRL_Led)LED
  - (DRL_Bulb)Bulb

- OPT
  - OR
    - OR
      - ALT
      - ALT

- ALT
  - (FogLight)Fog
  - (AutomaticLight)AL
  - (AdaptiveForwardLight)AFL
  - (DRL_Led)LED
  - (DRL_Bulb)Bulb

- OPT
  - OR
    - OR
      - ALT
      - ALT

- ALT
  - (FogLight)Fog
  - (AutomaticLight)AL
  - (AdaptiveForwardLight)AFL
  - (DRL_Led)LED
  - (DRL_Bulb)Bulb
LowBeamHalogen is effected. So, we implement second components, namely XenonDRL and HalogenDRL, for Xenon Low Beam and Halogen Low Beam, respectively. In comparison with LowBeamXenon and LowBeamHalogen, both new components have an extra input data for the duty cycle, i.e., the percentage of the ratio of pulse duration. The components are implemented and deposited to the repository, as well as the original components.

After the family model is structured, we retrieve XenonDRL and HalogenDRL from the repository, and define the interaction rules for them subsequently. For instance, Figure 7.9 shows how we define the interaction rule for XenonDRL in the family modelling tool. The interaction filter is formed of a red dot with the text “FI” positioned in the $F\text{-}SEL_1$, which is the family composition connector that composes $DRL\_\text{LowBeam}$ and LowBeamXenon. As a result, XenonDRL will automatically replaces LowBeamXenon when generating a product embedding both Xenon Low Beam and Reduced Low Beam.

![Feature Interaction](image)

Figure 7.9: Defining interaction rule for XenonDRL.

7.3.5 Step 5: Add Data Channels and Constraints

The next step is to add data channels (e.g., SCL.cmd $\rightarrow$ Fog.cmd in Figure 7.7) and constraints (e.g., SCL requires Fog). The final product family is shown in Figure 7.10.

---

2For the clarity, we omit the data channels.
CHAPTER 7. CASE STUDY: EXTERNAL CAR LIGHT FAMILY

Figure 7.10: The whole ECL product family.
7.4 Testing of ECL Family and Products

7.4.1 Step 1: Domain Unit Testing of ECL Family

Now we present the testing process of the constructed ECL family. The first step is domain unit testing in order to validate the functionality of every component in the family.

Figure 7.11 shows the FogLight is tested by the testing tool in our IDE. As shown in Figure 7.4(a), FogLight is an atomic component that only contains one class. Its source code is displayed on the left side of Figure 7.11. On the right, we prepare a test case that aims at switching on the fog light, which is satisfied according to the testing result.

Figure 7.12 shows the unit testing of a composite component. StaticCornerLight is composed by TurningSpeed, TurningAngle and ActivateSCL, as constructed in Figure 7.4(b). Therefore, its source code contains four classes, in which three of them are the implementations of sub-components, while the main class StaticCornerLight exposes the service of the component that allows other components to access. Similarly, we create a test case aims at switching on the static cornering light. The testing result not only shows the car speed, the steering wheel angle and the state of static cornering light, but also express the next command that indicates to turn on the fog light.
7.4.2 Step 2: Domain Integration Testing of ECL Family

At this stage, we should examine (1) the mapping between leaf features in ECL feature model and bottom-level components in ECL product family, (2) the mapping between feature relationships in ECL feature model and component relationships in ECL product family, and (3) the validity of data channels in the final ECL family model.

It is not difficult to investigate the mapping between leaf features and bottom-level components. Table 7.1 enumerates the names of every leaf features and related components. Notably, \textit{LowBeamXenon} and \textit{LowBeamHalogen} both have a second implementation to address the feature interactions in the family. As a result, we can match the 12 leaf features with 14 bottom-level components.

Then we need to verify that every variation generator in the product family is mapped onto a variation point in the feature model by means of the type and position. For example, in ECL family, feature \textit{FOG LIGHT} is optional, so we should check whether the component \textit{FogLight} is linked to an \textit{OPT} variation generator above or not. Furthermore, we need to ensure that the constraints between features are realised in the family. For instance, “\textit{STATIC CORNERING LIGHT} requires \textit{FOG LIGHT}” in the ECL feature model is realised by “\textit{SCL.cmd} → \textit{Fog.cmd}” in the ECL product family, as the purple arrow in Figure 7.10 shows.

If the components, variation generators and constraints are all correct, the amount
of products from the product family is equal to the amount of variants from the feature model. For example, Figure 7.3 specifies all 384 valid variants in terms of feature combinations, while Figure 7.13 lists all 384 ECL products formed by composition of components. Any ECL variant in the variant explorer is relevant to a ECL application in the product explorer.

![ExternalCarLight: 384 products](image)

Finally, we need to check the data channels. A data channel with reverse direction is not allowed in our tools. The completeness of data channels should be confirmed by visual inspection.

### 7.4.3 Step 3: Domain System Testing of ECL Family

At this stage, we use a family-based domain system testing approach to test the whole ECL family. As presented earlier in Chapter 5.4, we can derive a STATEMATE statechart from the ECL product family. The statechart is demonstrated in Figure 7.14, which allows us to use examine the behaviour among components by testing the transitions.

There are 18 test cases, i.e., state transitions, in total, which are resulted from a $F_{-LOP}$ (which corresponds to transitions No.1 and No.2), a $F_{-SEL}$ (which corresponds to transitions No.3 to No.6), a $F_{-SEQ}$ (which corresponds to transitions No.7 and No.8),
Figure 7.14: Featured statechart of ECL family.

and two ‘or’ variation generators (which correspond to transitions No.9 to No.14 and No.15 to No.18, respectively).

7.4.4 Step 4: Application Testing of ECL products

After the family-based testing at domain level, now we can generate desired products and test them. All the ECL products are enumerated in the product explorer, as shown in Figure 7.13. Here we choose product No.165 as a sample.

Figure 7.15(a) illustrate the model of ECL product No.165 in form of a composite component. Thus, we can either test this product directly by the testing tool in our IDE, or download its source code and operate a local testing. Figure 7.15(b) shows one of the execution results displayed by the testing tool.
7.5 Summary

In this chapter, we have elaborated how to model and implement a family of 384 external car light (ECL) systems. With an industrial context, this use case has demonstrated...
the distinguishing properties of our approach, i.e., to construct enumerative variability for a product family in a feature-oriented component-based manner.

Another area where enumerative variability can bring advantages is product family testing. We have presented how to test a family of 384 products at different testing levels. The key is that the ECL product family we constructed supports the family-based testing, which is barely practical in other product lines.

In order to test the limitation of our family construction approach, we will try more industrial examples with more features and requirement in future.
Chapter 8

Evaluation

“Knowing others is wisdom, knowing yourself is enlightenment.”
— Lao Tzu

8.1 Introduction

In this chapter, we evaluate the ECL product family modelled and implemented earlier, in order to determine merit, worth, value or significance of our feature-oriented component-based PFE approach [Scr91].

There are numerous methods and techniques of evaluation in PLE, which is either qualitative or quantitative, such as benchmarking [SGB12], metrics [dOJGM08], statistical analysis [tBLLV15], cost-benefit analysis [AC06], etc.

According to IEEE 1601 standards, software quality is the degree to which software possesses a desired combination of attributes\(^1\). In Chapter 8.2, we assesses the ECL product family and vending machines example by several quality attributes, e.g., testability, scalability, maintainability and evolvability.

An PFE evaluation framework is designed for evaluations across different implemented product families. Chapter 8.3 introduces an evaluation framework, namely family evaluation framework (FEF) [vdL05], which assesses product families by benchmarking.

\(^1\)https://standards.ieee.org/standard/1601-2010.html
8.2 Evaluation by Quality Attributes

8.2.1 Testability

Testability refers to the effort and time required to validate the software against its requirement. Thus, a system with better testability can be validated faster [MGM06]. Unfortunately, unlike the traditional software engineering, in the context of PLE, the researches on testing have not made a significant progress. But they do point out that variability, as the primary strength of PLE, has the greatest impact on decreasing testability [KM06]. As a natural result, many researches have attempted to increase the testability by controlling the combinatorial explosion of feature combinations, e.g., sampling products for PLE testing [CDS06, McG01], reusing feature selections at a finer grain [CCR10] and variability-aware testing [KVRE+12]. However, none of these testing approaches actually works in large-scale product families, due to the parametric variability anyway results in exponential number of variants.

In Chapter 5, we have presented a family-based testing technique, which must be performed on the PFA constructed by our feature-oriented component-based PFE approach. Additionally, in Chapter 7, we practice this testing technique on the ECL family. The family statechart (Figure 7.14) generated from the ECL PFA allows us to examine the system behaviour at the family level. Here, we will test the ECL family by other testing methods, in order to evaluate the testability by comparison.

Firstly, we choose the product-based testing with pure application strategy. This strategy is commonly used in current PLE approaches, due to it is straightforward and accessible. Most product lines are formed as code bases, in which the feature implementations are scattered. Apparently, we cannot test the code fragments in isolation. Thereby, essentially, each test case is a configured and generated software product. As a natural consequence, in order to assess the testability of pure application strategy, we have to compute the time complexity of product generation.

As presented in Chapter 2.3, the feature model is defined with a set of variation points. A mandatory point gives one variant, whereas an optional point provides two. An alternative point shows $n$ variants, while an or point outputs $2^n - 1$ variants, where $n$ is the number of features connected to the variation points. Accordingly, we can calculate the total amount of products. Feature model is structured in the form of tree, from which every abstract feature is a node. We use $P(n)$ to denote the number of products of a node $n$. Hence, the root node has $P(r)$ products, i.e., all products in the
family, while for any leaf feature, $P(l) = 1$. Now, when a node $n$ has $s$ children, then depending on the variation point, $P(n)$ can be:

- mandatory: $P(n) = \prod_{i=1}^{s} P(n_i)$
- optional: $P(n) = \prod_{i=1}^{s} P(n_i + 1)$
- alternative: $P(n) = (\prod_{i=1}^{s} P(n_i + 1)) - 1$
- mandatory: $P(n) = \prod_{i=1}^{s} P(n_i)$

It becomes obvious that the time complexity of computing $P(r)$ is quadratic on the number of variation points. If we use $S_k$ to denote the number of products that selects any combination of $k$ children from $s$, we can calculate $P(n)$ as below:

$$P(n) = \sum_{k=1}^{s} S_k$$  \hspace{1cm} (8.1)

Whilst $S_k$ can be calculated by summing the number of products of possible $k$-combinations, as shown below:

$$S_k = \sum_{1 \leq n_1 < n_2 < \ldots < n_k \leq s} P(n_1)P(n_2)\ldots P(n_k)$$  \hspace{1cm} (8.2)

Therefore, based on equation 8.1 and 8.2, we can calculate the time complexity $C_1$ for $P(r)$, which is:

$$O(N2^N) \leq C_1 \leq O(N^22^N)$$  \hspace{1cm} (8.3)

Briefly, if a product line supports automatic generation, i.e., has a clean mapping between features and code fragments, $C_1$ is the time complexity when the product-based testing with pure application strategy is performed. In the ECL family, the total number of products is 384, which can cost a tester months of work.

Next, we will analyse variability-aware testing. As we introduced in Chapter 5, the major goal of variability-aware testing is to test the common code only once. It creates a single program for the product family, which uses variable features as parameters, in order to control the execution path for variabilities. Therefore, if a feature model consists of $n$ variable features, we may need $2^n$ test cases. Here, we demonstrate it on the ECL family. Figure 8.1 shows the source code of the main class of the program. As a matter of fact, according to the program, we can create a FODA-like statechart, which uses features as transition conditions.

Comparing to product-based testing with pure application strategy, variability-aware testing has improvements. For example, it can be used for the product lines
Figure 8.1: Variability-aware testing of ECL family.
without clean feature mappings. But, according to our experience on ECL family, it
takes a lot of time and effort to identify the variability in the code base and reconstruct
a new variability-aware model. Anyway, in the best scenario, the time complexity \(C_2\)
for variability-aware testing is \(O(2^N)\).

Finally, let us have a look at our family-based testing. The related theory has
been presented in Chapter 5, while the ECL family statechart has been demonstrated
in Chapter 7. \(F-SEQ\) or \(F-SEL\) denotes \(n\) transitions between states, where \(n\) is the
number of connected component sets. \(F-GUD\) yields one transition and an \(F-LOP\)
yields two. the ‘or’ variation generator corresponds to an aggregator (Figure 3.7),
which causes \(2n\) transitions between states, where \(n\) components are composed by the
‘or’ variation generator. As Figure 7.14 shows, there are only 14 components needed
to be tested in the domain unit testing, as well as 18 state transitions in the domain
system testing. Therefore, the time complexity \(C_3\) for family-based testing is \(O(N)\).

In conclusion, the comparison result of three time complexities is \(C_3 < C_2 < C_1\).
Therefore, it proves that the product family constructed by our PFE approach has a
much better testability.

### 8.2.2 Scalability

Scalability refers to the impact of code expansion [AG00]. In the context of PLE, it
depends on the complexity of variability modelling, which may significantly increase
to a level where it is uncontrollable in the product families with massive variation
points and variants [CABA09]. For example, overuse of compilation conditions can
lead to the “ifdef hell” problem [FKA+13], whereas overlapped aspects or deltas can
cause fragile pointcut problem [ABKS13b].

In Chapter 3, we present an algebraic and hierarchical component model called
FX-MAN, which is the foundation of our approach. The compositional nature of FX-
MAN means that it can be used to compose families into bigger ones. This is possible
because variation generators and family connectors can be applied at any level of com-
position. For instance, our vending machines family only sells beverages, it can be
‘OR’ed with a street food selling systems family, and then composed by a family con-
nector with another product family, such as a movie ticket booking systems family, to
create a cinema management systems family (of families). Thus FX-MAN potentially
can facilitate hierarchical construction of very large families (of families).

We have compared our PFE approach with a feature-oriented programming ap-
proach by realising the same product family of elevator systems [QL18b]. The family
only contains 3 leaf features. However, in order to test the scalability, we choose to implement a basic elevator system at the beginning, and later extend it with one feature at a time. During the reconstruction, we analyse the working effort, which is directly proportional to the amount of changed code modules or components. Because the FOSD approach establishes one-to-many mapping between features and code fragments, every time a new feature is added, we have to insert multiple code fragments into the code base, as illustrated in Algorithm 1. Accordingly, the time complexity $C_1$ is $O(N^2)$.

Algorithm 1 Extending product family in FOSD.

```plaintext
for each $f \in \text{features}$ do
    implement code fragments $CFs$ for $f$
    for each $c \in CFs$ do
        add $c$ to the code base
    end for
end for
```

On the contrary, in our PFA, we only need to add a component, a variation generator and possibly a family composition connector. Hence, the time complexity $C_2$ is $O(N)$. Apparently, $C_2 < C_1$. So our approach does provide better scalability.

8.2.3 Maintainability

Maintainability is a composition of three main subcharacteristics: analysability, changeability, and understandability [BG11]. We can perform the evaluation based on these three quality attributes, which are interpreted as follows:

- **Analysability.** This attribute describes the capability of model or source code of a software product family to be diagnosed for deficiency. In Chapter 5, we have already revealed the issues of current software product line testing. Meanwhile, we present how to test our product family in detail. Errors at different levels of abstraction can be easily spotted by various levels of testing in domain. In addition, we can trace the detected errors back to the related features.

- **Changeability.** This attribute refers to the possibility and ease of modification in a product line. Current PLE approaches have to face the inevitable scattering and tangling problem. The former results from 1-to-n feature mappings (e.g., FOSD), while the latter caused by n-to-m feature mappings (e.g., FODA). As
a result, if we try to change the functionality of a given feature, we have to revise several artefacts in the solution spaces, e.g., multiple classes in the source files. Moreover, we must take into account couplings related to the targeted artefacts. That means we need to examine more artefacts than the ones we suppose to change. In the worst scenario, we have to traverse the solution space. Contrariwise, in our product family, we only need to upgrade one component for a feature, and check the data channels linked with it. It is worth noting that changeability looks similar to another quality attribute, called evolvability, but there is a subtle difference (see below).

- **Understandability.** This attribute indicates how easy it is to understand a software product line by its developers and users. As a matter of fact, product families of specific domains are usually hard to be comprehended, especially to users without a background of software engineering. Therefore, product line developers create feature models at the beginning of domain engineering. A feature model consists of features, which are not only capabilities of the applications, but also considered from the end-user’s point of view [KCH+90]. The family model we construct is a tree, which indeed is structurally isomorphic to the feature model. That means, the product families constructed by enumerative variability are as easy to understand as the feature models. On contrary, current PLE approaches normally creates a bundle of artefact fragments, which is impossible to help the users to understand.

### 8.2.3 Evolvability

Evolution of software is inevitable in industry, due to the changing requirement must be satisfied during the life cycle. Thus, the cost of software mainly depends on the evolution in long term. In this context, we prefer the definition of evolvability as: “An attribute that bears on the ability of a system to accommodate changes in its requirements throughout the system’s lifespan with the least possible cost while maintaining architectural integrity” [RLL94].

In PLE, evolvability is the capability of a product line to enable its own evolution, which not only includes the added or modified features, but also the update of the relationships and the variation points [DS06]. By comparison, changeability usually refers to the change only happening in solution space, such as refactoring the legacy code and modules, i.e., the feature model and functional requirement are not evolved.
We will assess the evolvability of our product family under various scenarios of evolution as below:

- **Addition of a feature.** This part is highly related to scalability. If we add a new (leaf) feature in the feature model, we need to implement this feature as a component, and insert the component into the family model at the same position as the feature placed in the feature model. If the feature is not mandatory, i.e., a new variation point is also added in the feature model, we must add a corresponding variation generator that connects to the components we created for the feature.

  ![Figure 8.2: Addition of a feature.](image)

  For example, as shown in Figure 8.2, if an optional feature **Beer** is added to the vending machine feature model (Figure 2.3), we can evolve the family model by adding an **OPT** generator and a component **Beer**. Contrariwise, current PLE approaches have to insert several code segments into different places in the code base. However, it is worth noting that sometimes a new feature changes the behaviour at the domain level. For instance, in comparison with Figure 2.3, Figure 4.4 adds an interacting feature **CANCEL**. As a result, we design a new integration of family composition connectors (Figure 4.16) in order to realise the new requirement.

- **Changing of a feature.** Figure 8.3 shows how to evolve the vending machine family when feature **Tea** is replaced by **Beer**. Notably, we should examine the data channels connected to component **Beer**, due to the data couplings or stamp couplings existing between the components. On the other hand, current PLE approaches do not define an explicit mapping between features and components, which results in two issues: (1) it is difficult to trace a feature’s implementations
scattering cross the solution space; (2) it takes more time and effort on integration testing because of the high couplings (e.g., content couplings, common couplings) between code modules.

- **Remove of a feature.** Removing a feature is similar to changing a feature. When we decide to remove a feature, we must remove all the implementations of this feature. Meanwhile, we may need to modify the relevant variation points. For instance, as illustrated in Figure 8.4, feature \text{TEA} is removed from the feature model, as well as the variation point \text{or}. Accordingly, we delete the component \text{Tea} with its variation generator \text{OR}. However, some PLE approaches have code tangling problem, i.e., a module in the solution space may simultaneously interact with several features [SRP06]. Hence, it takes more time and effort to determine that if a code module should be removed.

- **Modification of a variation point.** For example, in the vending machine family in Chapter 4, if \text{or(COFFEE, TEA)} is changed to \text{alternative(COFFEE, TEA)} in the feature model, all we need to do is replacing the OR by ALT in the product family.
However, current PLE approaches do not respond to the modifications of feature relationships. That means, they do not make any change in the code base, due to the variation points in the feature model are not modelled and implemented at all. Simply put, our approach guarantees that the variability in the solution space is always in conformity with the variability in the problem space.

![Modification of a variation point.](image)

To summarise, traceability is a prerequisite for successful evolution of a software product family without neglecting its consistency [DS06]. Therefore, earlier PLE approaches usually do not have a good evolvability, e.g., FODA [KCH+90] and generative programming [CE00]. Just like the scalability, the time complexity is $O(N^2)$. In contrary, the clearer mapping between problem and solution space an approach defines, the better evolvability the product family owns. Thereby, the software product families constructed by enumerative variability has a distinctive evolvability, due to our time complexity is $O(N)$.

### 8.2.5 Reusability

Component-based development (CBD), also known as component-based software engineering (CBSE), is reuse-based paradigm that aims at modelling, implementing and composing loosely-coupled components into systems [KM10]. Our software product family engineering approach is based on FX-MAN component model, as introduced in Chapter 3. Reusability, therefore, is the capability of a previously implemented and deposited FX-MAN component to be used again or used repeatedly in part or in its entirely, with or without modification, for its family or other families [TS14].

In the ECL family, many components can be reused, such as LowBeamXenon, LowBeamHalogen, FogLight, etc. Furthermore, the compositional nature of FX-MAN
component model allows the reuse of components at different levels of granularity. For example, we can create two atomic components called SwitchOn and SwitchOff that can be used repeatedly to compose every component controlling lights, or we can create a composite component called ToggleLight (composed by SwitchOn and SwitchOff) that can also be used repeatedly to compose every component controlling lights. Moreover, because the mapping between features and components is 1-to-1, we can reuse the features in different families, e.g., feature FOG LIGHT can be reused in a family of ship light systems.

On the other hand, current PLE approaches implement a feature as code modules or model fragments, which is hard to be systematically reused. In addition, a feature’s implementation relies on (part of) the base program. Thereby, features are specified for a particular domain, i.e., they cannot be reused for other families.

### 8.3 Evaluation by Framework

#### 8.3.1 Family Evaluation Framework

Family evaluation framework (FEF) [vdL05] is a framework proposed by three European co-operation projects: ESAPS [vdLO00, JRVDL00], CAFÉ [VdL02b] and FAMILIES [PBvDL05]. It has been widely used in industry [VdLSR07] and academia [AOvOvdL00, OMA+, Wij02, OR10] for evaluating software product families (product lines).

FEF considers all BAPO (Business, Architecture, Process and Organisation) concerns. The four dimensions are applied to evaluate product families [AOvOvdL00, Wij02, VDLBK+04]:

- **B** The business dimension concerns the business relationships between domain and application engineering, such as costs, profits and market value.

- **A** The architecture dimension refers to the variability mechanisms, and how the whole product family architecture effects the individual product architectures.

- **P** The process dimension assesses the roles, responsibilities and relationships within product family engineering by maturity model, such as CMMI (Capability Maturity Model Integration) [Tea10].
The organisation dimension focuses on the collaboration and coordination between the roles and responsibilities during domain engineering and application engineering.

FEF defines a set of aspects for each dimension, and maturity levels ranging from 1 to 5 for each aspect. Thereby, the quality of a product family can be reflected in grades of aspects. In this chapter, we aim at benchmarking the product family constructed by enumerative variability, as well as the product lines implemented by different kinds of PLE approaches, in technical contexts only. Therefore, business, process and organisation dimensions are not taken into account. Table 8.1 expresses the five levels of four aspects within the architecture dimension. The details of the four aspects are discussed below:

<table>
<thead>
<tr>
<th>Level</th>
<th>PFA</th>
<th>Product Quality</th>
<th>Reuse Levels</th>
<th>Variability Management</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lv 1: Independent Product Development</strong></td>
<td>not established</td>
<td>ignored / ad hoc</td>
<td>no institutionalised reuse</td>
<td>absent</td>
</tr>
<tr>
<td><strong>Lv 2: Standardised Infrastructure</strong></td>
<td>specified external components</td>
<td>partially supported</td>
<td>only external components</td>
<td>limited variation points</td>
</tr>
<tr>
<td><strong>Lv 3: Software Platform</strong></td>
<td>common features captured</td>
<td>inherited from the platform</td>
<td>internal platform components</td>
<td>at the platform level</td>
</tr>
<tr>
<td><strong>Lv 4: Variant Products</strong></td>
<td>fully specified</td>
<td>key priority</td>
<td>managed</td>
<td>many variation points</td>
</tr>
<tr>
<td><strong>Lv 5: Self-Configurable Products</strong></td>
<td>enforced</td>
<td>embedded in PFA</td>
<td>automatic generation</td>
<td>full variation points</td>
</tr>
</tbody>
</table>

Table 8.1: Architecture dimension: aspects and levels.

- **Software product family architecture (PFA):** PFA can be formed at different levels. The level depends on what artefacts are shared among the generated products. The higher the level of PFA, the more impact the PFA on development of products. Thereby, a low-level PFA may only make a distinction between infrastructure, and components specified for products. Contrariwise, a high-level PFA enforces the architecture of produced products.

- **Product quality:** Product quality refers to the correct functionality provided by the PFA. Quality is more and more explicitly managed within the product architecture with increasing levels.

- **Reuse levels:** The reuse level depends on how much effort is spent on generating
shared and reusable artefacts in domain engineering, and the extent of the reuse of solution space artefacts in application engineering.

- **Variability management**: The management of software variability refers to the use of variation points assembled in the PFA, and what kinds of variability mechanisms are adopted during the development of product family.

Now, we elaborate the maturity levels of architecture dimension of family evaluation framework (FEF-A).

**Level 1 - Independent Product Development**

The software product family engineering is not utilised. Products are developed independently.

1. **Software product family architecture** (PFA). PFA is not established.

2. **Product quality**. Due to the products are developed independently, the product quality is ignored, or managed in an ad hoc fashion.

3. **Reuse levels**. The reuse may happen, but not in an institutionalised manner.

4. **Variability management**. Because the developments of products are in isolation, variability in the family is impossible to be realised. So the variability management is absent.

In general, the problem space may be ready, but there is no solution space at all. For example, a very small family is analysed via a feature model, in which every product variant is assigned to a software engineer, whose programming skill determines the product quality.

**Level 2 - Standardised Infrastructure**

The software product family is implemented by incorporating third-party components into a standardised infrastructure.

1. **Software product family architecture** (PFA). The PFA is specified by external components.
2. **Product quality.** A standardised infrastructure certainly improves the product quality, but the overall quality still depends on the third-party components, which are probably not designed for the domain.

3. **Reuse levels.** The external components can be reused.

4. **Variability management.** A well-defined infrastructure may include some variation points. However, the external components do not have any.

In conclusion, the product family is modelled and implemented using traditional software engineering. Developers use a standardised infrastructure to reduce some duplicated code of common functionality. Therefore, there is no mapping between features and domain artefacts, and products cannot be automatically produced from the PFA.

**Level 3 - Software Platform**

The software product family is developed based on a software platform.

1. **Software product family architecture (PFA).** The PFA defines a software platform, which implements all common features.

2. **Product quality.** The product quality is inherited from the platform, i.e., the higher quality the platform owns, the higher quality the products get.

3. **Reuse levels.** Specific components suitable for the platform are implemented for the non-mandatory features. These components can be reused.

4. **Variability management.** The variation points are defined in the platform. Thus the variability management is at the platform level.

The major different between this level and previous level is that a platform is specifically developed for the software product family, while a standardised infrastructure is a generic base program for all kinds of further developments. The components are also specifically modelled that can be composed into the platform. As a result, the mapping exists between variable features and internal platform features.

**Level 4 - Variant Products**

The family architecture defines the construction of variant products.
1. **Software product family architecture (PFA).** The PFA is constructed and fully specified for a product family.

2. **Product quality.** As the key priority for development, the product quality is managed explicitly by the PFA.

3. **Reuse levels.** The reuse is managed in a systematic fashion that is based on an asset repository.

4. **Variability management.** The variation points, as well as the dependencies between them, are explicitly realised in the PFA.

In the previous level, the common features between all products are implemented in a platform, which brings many benefits to the organisation. The consequent development is to integrate variable features, which can be considered as common features between several products, into a platform. It arrives a new level, namely variant products. As the name indicates, the PFA determines the variant products in the family. In other words, all products are generated from the PFA directly.

**Level 5 - Self-Configurable Products**

The family architecture is implemented with pervasive rules, which support automatic selection of assets when products are derived.

1. **Software product family architecture (PFA).** The PFA is enforced, which means all commonality and variability is captured and realised. Hence, products do not need to change the architecture in their own architectures, i.e., all products use the architecture as-is and exploit the variation points to realise their own functionality [Bos02].

2. **Product quality.** Quality attributes are implemented within the PFA.

3. **Reuse levels.** The reuse of artefacts are automatically performed based on a shared repository when products are generated.

4. **Variability management.** The variability is verified at variation points. The automated selection of variants has been optimised.

This level describes an ideal PFA, which is suitable for an organisation who wants large amount of products from a stable domain. The architecture is enforced for all
valid products, and it establishes rules for automatic derivation. Massive customisation becomes realistic by the PFA.

8.3.2 Evaluation Result

In Chapter 2.3, we categorise current PLE approaches into three groups: annotative, compositional and transformational approach. We hereby use FEF-A to evaluate the ECL product family constructed in Chapter 7.

The PFA of ECL family is formed by an FX-MAN component, as shown in Figure 7.10. The components build in the ECL family model are mapped onto all features specified in the feature model. That means, all commonality and variability is captured and realised in the PFA. Moreover, the ECL family model integrates variation generators, each of which is modelled for a variation point specified in the feature model. Based on the variation generators, a product architecture can be directly derived from the PFA without any modification. Meanwhile, the composition connectors in the product architecture are inherited from the family connectors in the PFA. As a result, products use the architecture as-is and exploit the variation generators to satisfy their own requirement. In conclusion, the PFA of ECL family reaches level 5.

The ECL products are generated from the product family, since the architecture of a product is modelled as an X-MAN. Therefore, the product quality actually depends on the properties of X-MAN component model, as well as the FX-MAN component model. Chapter 8.2 measures the component by quality attributes. Accordingly, the ECL products have many advantages, such as high scalability, maintainability and reusability. In a nutshell, the quality attributes are implemented within the PFA. So, the product quality also reaches level 5.

The ECL family is developed by our IDE, which comprises a client-side repository. The components deposited in the repository can be retrieved for reuse. On the other hand, the exogenous composition mechanism of FX-MAN applying explicit variation generators at the family level allows a systematic reuse of components. Thus, the reuse level is level 4 in ECL family. If we ameliorate our IDE by using a server-side repository, the reuse level can be improved to level 5.

We model and implement enumerative variability to construct the whole ECL product family. The variability is verified at variation generators in the PFA, as the variants are defined at variation points in the feature model. All ECL variant products can be automatically produced in one go. Hence, the variability management of ECL family reaches level 5.
In summary, by FEF, the architecture dimension of ECL family reaches level 5, i.e., self-configurable products.

8.4 Summary

In Chapter 8.2, we choose several quality attributes to evaluate the performance of the product family constructed by enumerative variability. Notably, these attributes are used to realise the non-functional requirements. As a matter of fact, a number of evaluation methods have been proposed to assess different types of product line quality, e.g., reliability [MAKSC03, Imm06, KM04] and modifiability [GL00, Mac02, RDR03]. However, most of the methods are scenario-based, which are difficult to be straightforwardly transplanted to the use cases we designed.

FEF is a useful tool of evaluating product families. In Chapter 8.3, we only use it to evaluate the ECL family implemented by our approach. If the ECL family is implemented by other PLE approaches, we can predict the quality of its product family based on FEF-A maturity levels. Annotative approaches can implement an ECL product family that reaches level 3, i.e., developed based on a software platform, at most. A platform, e.g., CIDe and pure::variants, can implement all common features, while annotate variable code modules (classes, files, etc.) for non-mandatory features. The code modules may be reusable. On the other hand, compositional and transformational approaches are possible to implement a product family that reaches level 4 of FEF-A. Aspects or deltas are modelled and implemented for the non-mandatory features, whose composition mechanisms result in better reusability and quality. The variation points, as well as their usage constraints, can be determined by using some of the approaches.

By comparison, our feature-oriented component-based approach of product family engineering have more advantages than current PLE approaches.
Chapter 9

Discussion and Conclusion

“If a man keeps cherishing his old knowledge, so as continually to be acquiring new, he may be a teacher of others.”

— Confucius

9.1 Discussion

According to [Kru02], in the real world, many organisations adopt product families using three methods: proactive, reactive and extractive. A proactive approach implies that a product family is modelled from scratch. In contrast, a reactive approach begins with a small, easy to handle product family, which can be incrementally extended with new features and artefacts. An extractive approach starts with a portfolio of existing products and gradually refactor them to construct a product family. At present, it is apparent that our work is proactive. However, recent research [AL17] in reverse engineering suggests that our work also has potential to support reactive and extractive methods.

Our approach assumes that the variants defined by a feature model are always enumerable. However, some research has suggested that this assumption may not always hold; or at least it may be difficult to ascertain. For instance, in a cardinality-based feature model [CHE05] some features can be copied an unlimited number of times in the family, thus the variants are not enumerable in both problem and solution spaces.
9.2. ACHIEVEMENTS

On the other hand, as long as the feature model is fixed, our approach can model enumerative variability in the solution space.

In the case of features with feature attributes, i.e., measurable characteristics [BMAC05], e.g., the horsepower attribute of the engine starter feature in some automotive systems [WY05], the space of possible attribute values is seldom specified [Cue07], making it impossible to enumerate the variants.

The FX-MAN component model can be regarded as an architecture description language (ADL), due to it possesses the 5 characteristics of ADL [Cle96]:

1. FX-MAN supports the tasks of creating, refining and validating an architecture. It defines what constitutes a complete architecture.
2. An architecture style is defined as it “determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined” [GS93]. FX-MAN has the ability to represent the common architecture styles.
3. FX-MAN component is able to provide views of a family of systems that express the architectural information, but not leak the implementation information at the same time.
4. FX-MAN supports specification of families of components that all satisfy a common architecture.
5. FX-MAN can be analysed based on the architectural information. It can generate executable applications in form of X-MAN component.

Like any other PLE approach, our product family engineering approach also has limitations. For example, [MTS+17] shows how to implement an elevator family by using feature-oriented programming (FOP) in FeatureIDE. Indeed our approach can realise the behaviour of the elevator, but it cannot develop the user interface (UI) of simulation, which can be achieved by FOP. The reason it that FOP is a low-level programming language, which is capable of overwriting any code directly, including the UI logic of the application.

9.2 Achievements

This research has made the following contributions:
1. We define a new version of FX-MAN, which determines the 1-to-1 mapping between features and components, variation points and variation generators.

2. We define what enumerative variability is, and elaborate how it is formed in problem space and solution space in product family engineering.

3. We put forwards an approach to constructing the whole product families by enumerative variability in a feature-oriented, component-based manner.
   - We address the feature interaction problem during the family construction.
   - We address the crosscutting concern during the family construction.
   - We resolve the cross-tree constraint problem by family filters.

4. We propose domain testing methods at different levels to test the product families we constructed in a family-based fashion.

5. We develop an IDE that enables modelling and implementing product families from scratch. The IDE consists of feature modelling tool, atomic component modelling tool, composite component modelling tool and family modelling tool.

6. We implement an industrial use case using our IDE to demonstrate our approach.

7. We evaluate our product families through quality attributes and family evaluation framework, respectively.

### 9.3 Future Work

Our approach and tool offers a cornerstone for future work. There are many new possible directions:

- Our IDE integrates a local repository implemented by indexedDB, which allows user to work independently. But it is not beneficial to an organisation due to the components cannot be shared between development teams. Therefore, we will implement the repository with a server-side NoSQL database, such as MongoDB.

- We have proposed a family-based testing method to test the product families at different domain levels. However, we only develop a domain unit testing tool embedded in the IDE, domain integration testing and domain system testing
must be performed by hand. Thus, we will develop two more testing tools. The first tool can check the completeness of data channels for domain integration testing, whereas the second one can derive a family functional model.

- FX-MAN component model defines four family connectors, each of which refers to a composition connector or adaptor defined in X-MAN component model. The connectors have semantics that are mapped onto statements in the programming languages. For instance, a selector SEL is implemented as a if-else conditional statement. That is why X-MAN and FX-MAN component can directly derive a statechart as functional model. However, usually not all state transitions are caused by statement, e.g., they can be event-driven or data-driven from external source. Hence, we will propose a void connector, which allows user to define the statement. For example, Figure 9.1 shows a connector that responses the external event, i.e., click on actual buttons of a vending machine. As a result, the system will serve coffee or tea depends on the selection in real world. With the new connector, our approach can deal with more complex families of applications, e.g., systems with human-machine interactions.

![Figure 9.1: Event-driven connector.](image)

- Industry 4.0, also known as the fourth industrial revolution, is a label given to the current trend of combination of cyber-physical systems (CPS), the Internet of things (IoT), cloud computing and cognitive computing in manufacturing technologies [LFK14]. CPSs for a specific domain share common functionalities, but also variable ones due to different physical elements which constitute a specific CPS product, which all together compose an CPS family [ILA17]. Moreover, product identification is a key task in Industry 4.0, so the traceability is extremely important [Bas17]. Therefore, our approach shows a promising potential, due to the product family we constructed can offer a 1-to-1 mapping between variability in problem and solution space. In future, we will investigate if our family construction approach can help with Industry 4.0.
Bibliography


[CA05] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A


21st International Systems and Software Product Line Conference-


Appendix A

Screenshots of Our IDE

In this appendix, we present the full screenshots of different tools in our IDE.

- Figure A.1 shows the feature modelling tool.
- Figure A.2 shows the atomic component modelling tool.
- Figure A.3 shows the composite component modelling tool.
- Figure A.4 shows the family modelling tool.
- Figure A.5 shows the product explorer enumerating all products.
- Figure A.6 shows the product model in the product explorer.
Figure A.1: Screenshot of feature modelling Tool.
Figure A.2: Screenshot of atomic component modelling tool.
Figure A.3: Screenshot of composite component modelling tool.
Figure A.4: Screenshot of family modelling tool.
Figure A.5: Screenshot of product explorer (all valid products).
Figure A.6: Screenshot of product explorer (Product No.4).